Lecture Notes in Computer Science 4276

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
University of California, Los Angeles, CA, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Robert Meersman Zahir Tari et al. (Eds.)

On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE

OTM Confederated International Conferences
CoopIS, DOA, GADA, and ODBASE 2006
Montpellier, France, October 29 – November 3, 2006
Proceedings, Part II

Springer

Volume Editors

Robert Meersman
Vrije Universiteit Brussel (VUB), STARLab
Bldg G/10, Pleinlaan 2, 1050 Brussels, Belgium
E-mail: meersman@vub.ac.be

Zahir Tari
RMIT University, School of Computer Science and Information Technology
Bld 10.10, 376-392 Swanston Street, VIC 3001, Melbourne, Australia
E-mail: zahirt@cs.rmit.edu.au

Library of Congress Control Number: 2006934986

CR Subject Classification (1998): H.2, H.3, H.4, C.2, H.5, I.2, D.2.12, K.4

LNCS Sublibrary: SL 3 – Information Systems and Application, incl. Internet/Web and HCI

ISSN 0302-9743
ISBN-10 3-540-48274-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-48274-1 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11914952 06/3142 5 4 3 2 1 0

Volume Editors

Robert Meersman
Zahir Tari

CoopIS

Mike Papazoglou
Louiqa Raschid
Rainer Ruggaber

DOA

Judith Bishop
Kurt Geihs

GADA

Pilar Herrero
María S. Pérez
Domenico Talia
Albert Zomaya

ODBASE

Maurizio Lenzerini
Erich Neuhold
V.S. Subrahmanian

OTM 2006 General Co-chairs' Message

Dear OnTheMove Participant or Reader of these Proceedings,

The General Chairs of OnTheMove 2006, Montpellier, France, are happy to observe that the conference series that was started in Irvine, California in 2002 and subsequently held in Catania, Sicily in 2003 and in Cyprus in 2004 and 2005 clearly continues to attract a growing representative selection of today's worldwide research on the scientific concepts underlying distributed, heterogeneous and autonomous yet meaningfully collaborative computing, with the Internet and the WWW as its prime epitomes.

Indeed, as such large, complex and networked intelligent information systems become the focus and norm for computing, it is clear that there is an acute and increasing need to address and discuss in an integrated forum the implied software and system issues as well as methodological, theoretical and application issues. As we all know, e-mail, the Internet, and even video conferences are not sufficient for effective and efficient scientific exchange. This is why the OnTheMove (OTM) Federated Conferences series has been created to cover the increasingly wide yet closely connected range of fundamental technologies such as data and Web semantics, distributed objects, Web services, databases, information systems, workflow, cooperation, ubiquity, interoperability, mobility, grid and high-performance. OnTheMove aspires to be a primary scientific meeting place where all aspects of the development of Internet- and Intranet-based systems in organizations and for e-business are discussed in a scientifically motivated way. This fifth 2006 edition of the OTM Federated Conferences event therefore again provided an opportunity for researchers and practitioners to understand and publish these developments within their individual as well as within their broader contexts.

The backbone of OTM was originally formed by the co-location of three related, complementary and successful main conference series: DOA (Distributed Objects and Applications, since 1999), covering the relevant infrastructure-enabling technologies, ODBASE (Ontologies, DataBases and Applications of SEmantics, since 2002) covering Web semantics, XML databases and ontologies, CoopIS (Cooperative Information Systems, since 1993) covering the application of these technologies in an enterprise context through, for example, workflow systems and knowledge management. For the 2006 edition, these were strengthened by a fourth conference, GADA (Grid computing, high-performAnce and Distributed Applications, a successful workshop at OTM since 2004), covering the large-scale integration of heterogeneous computing systems and data resources with the aim of providing a global computing space. Each of these four conferences encourages researchers to treat their respective topics within a framework that incorporates jointly (a) theory , (b) conceptual design and development, and (c) applications, in particular case studies and industrial solutions.

Following and expanding the model created in 2003, we again solicited and selected quality workshop proposals to complement the more "archival" nature of the main conferences with research results in a number of selected and more "avant garde" areas related to the general topic of distributed computing. For instance, the so-called Semantic Web has given rise to several novel research areas combining linguistics, information systems technology, and artificial intelligence, such as the modeling of (legal) regulatory systems and the ubiquitous nature of their usage. We were glad to see that several earlier successful workshops (notably WOSE, MIOS-INTEROP, AweSOMe, CAMS, SWWS, SeBGIS, ORM) re-appeared in 2006 with a second, third or sometimes fourth edition, and that not less than seven new workshops could be hosted and successfully organized by their respective proposers: IS (International Workshop on Information Security), COMINF (International Workshop on Community Informatics), KSinBIT (International Workshop on Knowledge Systems in Bioinformatics), MONET (International Workshop on MObile and NEtworking Technologies for social applications), OnToContent (Ontology content and evaluation in Enterprise), PerSys (International Workshop on Pervasive Systems), and RDDS (International Workshop on Reliability in Decentralized Distributed Systems). We know that as before, their audiences will mutually productively mingle with those of the main conferences, as is already visible from the overlap in authors! The OTM organizers are especially grateful for the leadership and competence of Pilar Herrero in managing this complex process into a success for the second year in a row.

A special mention for 2006 is again due for the third and enlarged edition of the highly attractive OnTheMove Academy (formerly called Doctoral Consortium Workshop). Its 2006 Chairs, Antonia Albani, Gábor Nagypál and Johannes Maria Zaha, three young and active researchers, further refined the original set-up and interactive formula to bring PhD students together: they call them to submit their research proposals for selection; the resulting submissions and their approaches are presented by the students in front of a wider audience at the conference, where they are then independently and extensively analyzed and discussed in public by a panel of senior professors. This year these were Johann Eder, Maria Orlowska, and of course Jan Dietz, the Dean of the OnTheMove Academy, who provided guidance, support and help for the team. The successful students are also awarded free access to all other parts of the OTM program, and only pay a minimal fee for the Doctoral Symposium itself (in fact their attendance is largely sponsored by the other participants!). The OTM organizers expect to further expand the OnTheMove Academy in future editions of the conferences and so draw an audience of young researchers into the OTM forum.

All four main conferences and the associated workshops share the distributed aspects of modern computing systems, and the resulting application-pull created by the Internet and the so-called Semantic Web. For DOA 2006, the primary emphasis was on the distributed object infrastructure; for ODBASE 2006, it became the knowledge bases and methods required for enabling the use of formal semantics; for CoopIS 2006, the topic was the interaction of such

technologies and methods with management issues, such as occur in networked organizations, and for GADA 2006, the topic was the scalable integration of heterogeneous computing systems and data resources with the aim of providing a global computing space. These subject areas naturally overlap and many submissions in fact also treat an envisaged mutual impact among them. As for the earlier editions, the organizers wanted to stimulate this cross-pollination by a shared program of famous keynote speakers: this year we were proud to announce Roberto Cencioni (European Commission), Alois Ferscha (Johannes Kepler Universität), Daniel S. Katz (Louisiana State University and Jet Propulsion Laboratory), Frank Leymann (University of Stuttgart), and Marie-Christine Rousset (University of Grenoble)! We also encouraged multiple event attendance by providing all authors, also those of workshop papers, with free access or discounts to one other conference or workshop of their choice.

We received a total of 361 submissions for the four main conferences and an impressive 493 (compared to the 268 in 2005 and 170 in 2004!) submissions for the workshops. Not only may we indeed again claim success in attracting an increasingly representative volume of scientific papers, but such a harvest of course allows the Program Committees to compose a higher quality cross-section of current research in the areas covered by OTM. In fact, in spite of the larger number of submissions, the Program Chairs of each of the three main conferences decided to accept only approximately the same number of papers for presentation and publication as in 2003, 2004 and 2005 (i.e., average one paper out of four submitted, not counting posters). For the workshops, the acceptance rate varies but was much stricter than before, about one in two to three, to less than one quarter for the IS (Information Security) international workshop. Also for this reason, we separated the proceedings into two books with their own titles, with the main proceedings in two volumes, and we are grateful to Springer for their suggestions and collaboration in producing these books and CDROMs. The reviewing process by the respective Program Committees as usual was performed very professionally and each paper in the main conferences was reviewed by at least three referees, with arbitrated e-mail discussions in the case of strongly diverging evaluations. It may be worthwhile to emphasize that it is an explicit OnTheMove policy that all conference Program Committees and Chairs make their selections completely autonomously from the OTM organization itself. Continuing a costly but nice tradition, the OnTheMove Federated Event organizers decided again to make all proceedings available to all participants of conferences and workshops, independently of one's registration to a specific conference or workshop. Each participant also received a CDROM with the full combined proceedings (conferences + workshops).

The General Chairs are once more especially grateful to all the many people directly or indirectly involved in the setup of these federated conferences who contributed to making it a success. Few people realize what a large number of people have to be involved, and what a huge amount of work, and sometimes risk, the organization of an event like OTM entails. Apart from the persons in the roles mentioned above, we therefore in particular wish to thank our 12 main conference

PC Co-chairs (GADA 2006: Pilar Herrero, María S. Pérez, Domenico Talia, Albert Zomaya; DOA 2006: Judith Bishop, Kurt Geihs; ODBASE 2006: Maurizio Lenzerini, Erich Neuhold, V.S. Subrahmanian; CoopIS 2006: Mike Papazoglou, Louiqa Raschid, Rainer Ruggaber) and our 36 workshop PC Co-chairs (Antonia Albani, George Buchanan, Roy Campbell, Werner Ceusters, Elizabeth Chang, Ernesto Damiani, Jan L.G. Dietz, Pascal Felber, Fernando Ferri, Mario Freire, Daniel Grosu, Michael Gurstein, Maja Hadzic, Pilar Herrero, Terry Halpin, Annika Hinze, Skevos Evripidou, Mustafa Jarrar, Arek Kasprzyk, Gonzalo Méndez, Aldo de Moor, Bart De Moor, Yves Moreau, Claude Ostyn, Andreas Persidis, Maurizio Rafanelli, Marta Sabou, Vitor Santos, Simao Melo de Sousa, Katia Sycara, Arianna D'Ulizia, Eiko Yoneki, Esteban Zimányi).

All, together with their many PC members, did a superb and professional job in selecting the best papers from the large harvest of submissions.

We also heartily thank Zohra Bellahsene of LIRMM in Montpellier for the considerable efforts in arranging the venue at their campus and coordinating the substantial and varied local facilities needed for a multi-conference event such as ours. And we must all also be grateful to Mohand-Said Hacid of the University of Lyon for researching and securing the sponsoring arrangements, to Gonzalo Méndez, our excellent Publicity Chair, to our extremely competent and experienced Conference Secretariat and technical support staff Daniel Meersman, Ana-Cecilia Martinez Barbosa, and Jan Demey, and last but not least to our hyperactive Publications Chair and loyal collaborator of many years, Kwong Yuen Lai, this year bravely assisted by Peter Dimopoulos.

The General Chairs gratefully acknowledge the academic freedom, logistic support and facilities they enjoy from their respective institutions, Vrije Universiteit Brussel (VUB) and RMIT University, Melbourne, without which such an enterprise would not be feasible.

We do hope that the results of this federated scientific enterprise contribute to your research and your place in the scientific network... We look forward to seeing you again at next year's edition!

August 2006

Robert Meersman, Vrije Universiteit Brussel, Belgium
Zahir Tari, RMIT University, Australia
(General Co-chairs, OnTheMove 2006)

Organization Committee

The OTM (On The Move) 2006 Federated Conferences, which involve CoopIS (Cooperative Information Systems), DOA (distributed Objects and Applications), GADA (Grid computing, high-performAnce and Distributed Applications), and ODBASE (Ontologies, Databases and Applications of Semantics), are proudly supported by CNRS (Centre National de la Researche Scientifique, France), the City of Montpellier (France), Ecole Polytechnique Universitaire de Montepellier, Université de Montpellier II (UM2), Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), RMIT University (School of Computer Science and Information Technology), and Vrije Universiteit Brussel (Department of Computer Science).

Executive Committee

OTM 2006 General Co-chairs:	Robert Meersman (Vrije Universiteit Brussel, Belgium) and Zahir Tari (RMIT University, Australia).
GADA 2006 PC Co-chairs:	Pilar Herrero (Universidad Politécnica de Madrid, Spain), María S. Pérez (Universidad Politécnica de Madrid, Spain), Domenico Talia (Universitá della Calabria, Italy), and Albert Zomaya (The University of Sydney, Australia).
CoopIS 2006 PC Co-chairs:	Mike Papazoglou (Tilburg University, Netherlands), Louiqa Raschid (University of Maryland, USA), and Rainer Ruggaber (SAP Research Center, Germany).
DOA 2006 PC Co-chairs:	Judith Bishop (University of Pretoria, South Africa) and Kurt Geihs (University of Kassel, Germany).
ODBASE 2006 PC Co-chairs:	Maurizio Lenzerini (Universitá di Roma "La Sapienza," Italy), Erich Neuhold (Darmstadt University of Technology, Germany), and V.S. Subrahmanian (University of Maryland College Park, USA).
Publication Co-chairs:	Kwong Yuen Lai (RMIT University, Australia) and Peter Dimopoulos (RMIT University, Australia).
Organizing Chair:	Zohra Bellahsene (LIRMM CNRS/University of Montpellier II, France).
Publicity Chair:	Mohand-Said Hacid (Université Claude Bernard Lyon I, France).

Secretariat: Ana-Cecilia Martinez Barbosa, Jan Demey, and Daniel Meersman.

CoopIS 2006 Program Committee

Marco Aiello
Alistair Barros
Boualem Benatallah
Salima Benbernou
Arne Berre
Elisa Bertino
Klemens Böhm
Alex Borgida
Luc Bouganim
Stephane Bressan
Laura Bright
Fabio Casati
Mariano Cilia
Vincenzo D'Andrea
Umesh Dayal
Alex Delis
Marlon Dumas
Schahram Dustdar
Johann Eder
Klaus Fischer
Avigdor Gal
Fausto Giunchiglia
Paul Grefen
Mohand-Said Hacid
Manfred Hauswirth
Willem-Jan van den Heuvel
Martin Hepp
Carsten Holtmann
Nenad Ivezic
Paul Johannesson
Manolis Koubarakis
Bernd Krämer
Winfried Lamersdorf
Steven Laufmann
Qing Li
Tiziana Margaria
Marta Mattoso
Massimo Mecella
Nikolay Mehandjiev
Brahim Medjahed
Michele Missikoff
Michael zur Muehlen
Jörg Müller
David Munro
Wolfgang Nejdl
Cesare Pautasso
Mourad Ouzzani
Manfred Reichert
Stefanie Rinderle
Uwe Riss
Timos Sellis
Anthony Tomasic
Farouk Toumani
Patrick Valduriez
Wil van der Aalst
Maria Esther Vidal
Mathias Weske
Jian Yang
Vladimir Zadorozhny

GADA 2006 Program Committee

Akshai Aggarwal
Alan Sussman
Alastair Hampshire
Alberto Sanchez
Alvaro A.A. Fernandes
Angelos Bilas
Antonio Garcia Dopico
Artur Andrzejak
Azzedine Boukerche
Beniamino Di Martino
Bhanu Prasad
Bing Bing Zhou

Carmela Comito
Carole Goble
Costin Badica
Dana Petcu
Daniel S. Katz
David Walker
Domenico Laforenza
Eduardo Huedo
Elghazali Talbi
Enrique Soler
Fatos Xhafa
Felix Garcia
Francisco Luna
Franciszek Seredynski
Gregorio Martinez
Hamid Sarbazi-Azad
Heinz Stockinger
Ignacio M. Llorente
Jack Dongarra
Jan Humble
Jemal Abawajy
Jesús Carretero
Jinjun Chen
Jon Maclaren
Jose L. Bosque
Jose M. Peña
Juan A. Botá Blaya
Kostas Karasavvas
Kurt Stockinger
Laurence T. Yang
Manish Parashar
Manuel Salvadores
Marcin Paprzycki
Maía Eugenia de Pool
Maria Ganzha
Mario Cannataro
Marios Dikaiakos
Mark Baker
Mirela Notare
Mohamed Ould-Khaoua
Neil P. Chue Hong
Omer F. Rana
Panayiotis Periorellis
Pascal Bouvry
Rainer Unland
Rajkumar Buyya
Reagan Moore
Rizos Sakellariou
Rosa M. Badia
Ruben S. Montero
Santi Caballé Llobet
Sattar B. Sadkhan Almaliky
Savitri Bevinakoppa
Stefan Egglestone
Thierry Priol
Toni Cortes
Valdimir Getov
Víctor Robles

ODBASE 2006 Program Committee

Sibel Adali
Maristella Agosti
Bill Andersen
Juergen Angele
Franz Baader
Sonia Bergamaschi
Alex Borgida
Christoph Bussler
Marco Antonio Casanova
Silvana Castano
Tiziana Catarci
Giuseppe De Giacomo
Stefan Decker
Rainer Eckstein
Johann Eder
Mohand Said Hacid
Jeff Heflin
Jim Hendler
Edward Hung
Arantza Illarramendi
Vipul Kashyap
Larry Kerschberg
Ross King
Roger

Harumi Kuno
Georg Lausen
Michele Missikoff
John Mylopoulos
Wolfgang Nejdl
Christine Parent
Thomas Risse
Heiko Schuldt
Peter Schwarz
Peter Spyns
York Sure
Sergio Tessaris
David Toman
Guido Vetere
Chris Welty

DOA 2006 Program Committee

Cristiana Amza
Matthias Anlauff
Mark Baker
Guruduth Banavar
Gordon Blair
Harold Carr
Geoff Coulson
Francisco "Paco" Curbera
Frank Eliassen
Tomoya Enokido
Patrick Eugster
Pascal Felber
Jeff Gray
Stefan Gruner
Mohand-Said Hacid
Franz Hauck
Naohiro Hayashibara
Hui-Huang Hsu
Mehdi Jazayeri
Eric Jul
Bettina Kemme
Fabio Kon
Joe Loyall
Peter Loehr
Frank Manola
Keith Moore
Francois Pacull
Simon Patarin
Peter Pietzuch
Joao Pereira
Arno Puder
Rajendra Raj
Andry Rakotonirainy
Luis Rodrigues
Isabelle Rouvellou
Rick Schantz
Heinz-W. Schmidt
Douglas Schmidt
Richard Soley
Michael Stal
Jean-Bernard Stefani
Stefan Tai
Hong Va Leong
Steve Vinoski
Norbert Voelker
Andrew Watson
Torben Weis
Doug Wells
Michael Zapf

OTM Conferences 2006 Additional Reviewers

Carola Aiello
Reza Akbarinia
Nicolas Anciaux
Samuil Angelov
Fulvio D'Antonio
Philipp Baer
Gabriele Barchiesi
Carlo Bellettini
Domenico Beneventano
Jesus Bermudez

Devis Bianchini
Steffen Bleul
Ralph Bobrik
Silvia Bonomi
Abdelhamid Bouchachia
Jose de Ribamar Braga
Vanessa Braganholo
Lars Braubach
Gert Brettlecker
Andrea Cali
Ken Cavanaugh
Emmanuel Coquery
Felix J. Garcia Clemente
Fabio Coutinho
Georges DaCosta
Antonio De Nicola
Fabien Demarchi
Henry Detmold
Christoph Dorn
Viktor S. Wold Eide
Hazem Elmeleegy
Mohamed Y. ElTabakh
Michael Erdmann
Rik Eshuis
Katrina Falkner
Alfio Ferrara
Ilya Figotin
Anna Formica
Nadine Froehlich
Jochen Fromm
Mati Golani
Gangadharan Gr
Tim Grant
Francesco Guerra
Pablo Guerrero
Yuanbo Guo
Peter Haase
Hakim Hacid
Christian Hahn
Bjorn-Oliver Hartmann
Martin Henkel
Eelco Herder
Edward Ho
Thomas Hornung
Sergio Ilarri
Markus Kalb
Marijke Keet
Kyong Hoon Kim
Mirko Knoll
Natallia Kokash
Iryna Kozlova
Christian Kunze
Steffen Lamparter
Christoph Langguth
Marek Lehmann
Domenico Lembo
Elvis Leung
Mario Lezoche
Baoping Lin
An Liu
Hai Liu
Andrei Lopatenko
Carsten Lutz
Linh Thao Ly
Jurgen Mangler
Vidal Martins
Pietro Mazzoleni
Massimo Mecella
Michele Melchiori
Eduardo Mena
Marco Mesiti
Tommie Meyer
Hugo Miranda
Jose Mocito
Thorsten Moeller
Stefano Montanelli
Cristian Madrigal Mora
Francesco Moscato
Anan Mrey
Dominic Müller
Sharath Babu Musunoori
Meenakshi Nagarajan
Dirk Neumann
Johann Oberleitner
Nunzia Osimi
Zhengxiang Pan
Paolo Perlasca
Illia Petrov
Horst Pichler
Christian Platzer

Antonella Poggi
Alexander Pokahr
Konstantin Pussep
Abir Qasem
Michael von Riegen
Francisco Reverbel
Haggai Roitman
Kurt Rohloff
Dumitru Roman
Florian Rosenberg
Nicolaas Ruberg
Kai Sachs
Martin Saternus
Monica Scannapieco
Daniel Schall
Sergio Serra
Kai Simon
Esteban Lean Soto
Stefano Spaccapietra
Michael Springmann
Iain Stalker
Nenad Stojanovic
Umberto Straccia
Gerd Stumme
Francesco Taglino
Robert Tairas
Wesley Terpstra
Eran Toch
Anni-Yasmin Turhan
Martin Vasko
Salvatore Venticinque
Maurizio Vincini
Johanna Voelker
Jochem Vonk
Denny Vrandecic
Andrzej Walczak
Ting Wang
Thomas Weise
Thomas Weishupl
Christian von der Weth
Karl Wiggisser
Hui Wu
Jianming Ye
Sonja Zaplata
Weiliang Zhao
Uwe Zdun
Ingo Zinnikus

Table of Contents – Part II

Grid Computing, High Performance and Distributed Applications (GADA) 2006 International Conference

GADA 2006 International Conference (Grid Computing, High-Performance and Distributed Applications) PC Co-chairs' Message 1117

Keynote

Data-Oriented Distributed Computing for Science: Reality and Possibilities 1119
Daniel S. Katz, Joseph C. Jacob, Peggy P. Li, Yi Chao, Gabrielle Allen

From Intelligent Content to Actionable Knowledge: Research Directions and Opportunities Under the EU's Framework Programme 7, 2007-2013 1125
Stefano Bertolo

Resource Selection and Management

Resource Selection and Application Execution in a Grid: A Migration Experience from GT2 to GT4 1132
A. Clematis, A. Corana, D. D'Agostino, V. Gianuzzi, A. Merlo

A Comparative Analysis Between EGEE and Grid*W*ay Workload Management Systems 1143
J.L. Vázquez-Poletti, E. Huedo, R.S. Montero, I.M. Llorente

Grid and HPC Dynamic Load Balancing with Lattice Boltzmann Models 1152
Fabio Farina, Gianpiero Cattaneo, Alberto Dennunzio

P2P-Based Systems

Trust Enforcement in Peer-to-Peer Massive Multi-player Online Games 1163
Adam Wierzbicki

A P2P-Based System to Perform Coordinated Inspections in Nuclear Power Plants 1181
C. Alcaide, M. Díaz, L. Llopis, A. Márquez, E. Soler

Grid File Transfer

Grid File Transfer During Deployment, Execution, and Retrieval 1191
Françoise Baude, Denis Caromel, Mario Leyton, Romain Quilici

A Parallel Data Storage Interface to GridFTP 1203
Alberto Sánchez, María S. Pérez, Pierre Gueant, Jesús Montes, Pilar Herrero

Parallel Applications

Parallelization of a Discrete Radiosity Method Using Scene Division 1213
Rita Zrour, Fabien Feschet, Rémy Malgouyres

A Mixed MPI-Thread Approach for Parallel Page Ranking Computation 1223
Bundit Manaskasemsak, Putchong Uthayopas, Arnon Rungsawang

Scheduling in Grid Environments

A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments 1234
George V. Iordache, Marcela S. Boboila, Florin Pop, Corina Stratan, Valentin Cristea

Solving Scheduling Problems in Grid Resource Management Using an Evolutionary Algorithm 1252
Karl-Uwe Stucky, Wilfried Jakob, Alexander Quinte, Wolfgang Süß

Integrating Trust into Grid Economic Model Scheduling Algorithm 1263
Chunling Zhu, Xiaoyong Tang, Kenli Li, Xiao Han, Xilu Zhu, Xuesheng Qi

Autonomous and Autonomic Computing

QoS-Driven Web Services Selection in Autonomic Grid Environments... 1273
Danilo Ardagna, Gabriele Giunta, Nunzio Ingraffia, Raffaela Mirandola, Barbara Pernici

Autonomous Layer for Data Integration in a Virtual Repository 1290
Kamil Kuliberda, Radoslaw Adamus, Jacek Wislicki, Krzysztof Kaczmarski, Tomasz Kowalski, Kazimierz Subieta

Grid Infrastructures for Data Analysis

An Instrumentation Infrastructure for Grid Workflow Applications 1305
Bartosz Balis, Hong-Linh Truong, Marian Bubak, Thomas Fahringer, Krzysztof Guzy, Kuba Rozkwitalski

A Dynamic Communication Contention Awareness List Scheduling Algorithm for Arbitrary Heterogeneous System 1315
Xiaoyong Tang, Kenli Li, Degui Xiao, Jing Yang, Min Liu, Yunchuan Qin

Access Control and Security

Distributed Provision and Management of Security Services in Globus Toolkit 4..... 1325
Félix J. García Clemente, Gregorio Martínez Pérez, Andrés Muñoz Ortega, Juan A. Botía Blaya, Antonio F. Gómez Skarmeta

A Fine-Grained and X.509-Based Access Control System for Globus 1336
Hristo Koshutanski, Fabio Martinelli, Paolo Mori, Luca Borz, Anna Vaccarelli

Programming Aspects for Developing Scientific Grid Components

Dynamic Reconfiguration of Scientific Components Using Aspect Oriented Programming: A Case Study 1351
Manuel Díaz, Sergio Romero, Bartolomé Rubio, Enrique Soler, José M. Troya

MGS: An API for Developing Mobile Grid Services..... 1361
Sze-Wing Wong, Kam-Wing Ng

Databases and Data Grids

Using Classification Techniques to Improve Replica Selection in Data Grid 1376
Hai Jin, Jin Huang, Xia Xie, Qin Zhang

Searching Moving Objects in a Spatio-temporal Distributed Database Servers System 1388
Mauricio Marín, Andrea Rodríguez, Tonio Fincke, Carlos Román

Distributed Applications

A Generic Deployment Framework for Grid Computing and Distributed Applications 1402
Areski Flissi, Philippe Merle

CBIR on Grids 1412
Oscar D. Robles, José Luis Bosque, Luis Pastor, Ángel Rodríguez

Evaluation

Performance Evaluation of Group Communication Architectures in Large Scale Systems Using MPI 1422
Kayhan Erciyes, Orhan Dagdeviren, Reşat Ümit Payli

Distributed Objects and Applications (DOA) 2006 International Conference

DOA 2006 International Conference (Distributed Objects and Applications) PC Co-chairs' Message 1433

Keynote

Everyobjects in the Pervasive Computing Landscape 1434
Alois Ferscha

Services

Using Selective Acknowledgements to Reduce the Memory Footprint of Replicated Services 1435
Roy Friedman, Erez Hadad

Modularization of Distributed Web Services Using Aspects with Explicit Distribution (AWED) 1449
Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bart Verheecke

ANIS: A Negotiated Integration of Services in Distributed Environments 1467
Noha Ibrahim, Frédéric Le Mouël

Communications

Towards a Generic Group Communication Service 1485
Nuno Carvalho, José Pereira, Luís Rodrigues

Optimizing Pub/Sub Systems by Advertisement Pruning 1503
Sven Bittner, Annika Hinze

A Specification-to-Deployment Architecture for Overlay Networks 1522
Stefan Behnel, Alejandro Buchmann, Paul Grace, Barry Porter, Geoff Coulson

Searching Techniques

Distributed Lookup in Structured Peer-to-Peer Ad-Hoc Networks 1541
Raphaël Kummer, Peter Kropf, Pascal Felber

A Document-Centric Component Framework for Document Distributions 1555
Ichiro Satoh

Shepherdable Indexes and Persistent Search Services for Mobile Users 1576
Michael Higgins, Dominic Widdows, Magesh Balasubramanya, Peter Lucas, David Holstius

Types and Notations

Distributed Abstract Data Types 1594
Gian Pietro Picco, Matteo Migliavacca, Amy L. Murphy, Gruia-Catalin Roman

Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services 1613
Frank Alexander Kraemer, Peter Herrmann, Rolv Bræk

Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model 1633
Nour Ali, Carlos Millán, Isidro Ramos

Adaptivity

Component QoS Contract Negotiation in Multiple Containers 1650
Mesfin Mulugeta, Alexander Schill

RIMoCoW, a Reconciliation Infrastructure for CORBA Component-Based Applications in Mobile Environments 1668
Lydialle Chateigner, Sophie Chabridon, Guy Bernard

A Component-Based Planning Framework for Adaptive Systems 1686
Mourad Alia, Geir Horn, Frank Eliassen, Mohammad Ullah Khan, Rolf Fricke, Roland Reichle

Middleware

A Case for Event-Driven Distributed Objects 1705
Aliandro Lima, Walfredo Cirne, Francisco Brasileiro, Daniel Fireman

MoCoA: Customisable Middleware for Context-Aware Mobile Applications 1722
Aline Senart, Raymond Cunningham, Mélanie Bouroche, Neil O'Connor, Vinny Reynolds, Vinny Cahill

A Framework for Adaptive Mobile Objects in Heterogeneous Environments 1739
Rüdiger Kapitza, Holger Schmidt, Guido Söldner, Franz J. Hauck

Distribution Support

A Novel Object Pool Service for Distributed Systems 1757
Samira Sadaoui, Nima Sharifimehr

A Java Framework for Building and Integrating Runtime Module Systems 1772
Olivier Gruber, Richard S. Hall

Transparent and Dynamic Code Offloading for Java Applications 1790
Nicolas Geoffray, Gaël Thomas, Bertil Folliot

Self-organisation

Self-organizing and Self-stabilizing Role Assignment in Sensor/Actuator Networks 1807
Torben Weis, Helge Parzyjegla, Michael A. Jaeger, Gero Mühl

Towards Self-organizing Distribution Structures for Streaming Media 1825
Hans Ole Rafaelsen, Frank Eliassen, Sharath Babu Musunoori

Bulls-Eye – A Resource Provisioning Service for Enterprise Distributed Real-Time and Embedded Systems 1843
Nilabja Roy, Nishanth Shankaran, Douglas C. Schmidt

Author Index 1863

Table of Contents – Part I

Cooperative Information Systems (CoopIS) 2006 International Conference

CoopIS 2006 International Conference (International Conference on Cooperative Information Systems) PC Co-chairs' Message 1

Keynote

Workflow-Based Coordination and Cooperation in a Service World 2
Frank Leymann

Distributed Information Systems I

Distributed Triggers for Peer Data Management 17
Verena Kantere, Iluju Kiringa, Qingqing Zhou, John Mylopoulos, Greg McArthur

Satisfaction-Based Query Load Balancing 36
Jorge-Arnulfo Quiané-Ruiz, Philippe Lamarre, Patrick Valduriez

Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System 54
Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, Ji Wu

Distributed Information Systems II

Views for Simplifying Access to Heterogeneous XML Data 72
Dan Vodislav, Sophie Cluet, Grégory Corona, Imen Sebei

SASMINT System for Database Interoperability in Collaborative Networks 91
Ozgul Unal, Hamideh Afsarmanesh

Querying E-Catalogs Using Content Summaries 109
Aixin Sun, Boualem Benatallah, Mohand-Saïd Hacid, Mahbub Hassan

Workflow Modelling

WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL 127
Kristian Bisgaard Lassen, Wil M.P. van der Aalst

Let's Dance: A Language for Service Behavior Modeling 145
Johannes Maria Zaha, Alistair Barros, Marlon Dumas, Arthur ter Hofstede

Dependability and Flexibility Centered Approach for Composite Web Services Modeling 163
Neila Ben Lakhal, Takashi Kobayashi, Haruo Yokota

Aspect-Oriented Workflow Languages 183
Anis Charfi, Mira Mezini

Workflow Management and Discovery

A Portable Approach to Exception Handling in Workflow Management Systems 201
Carlo Combi, Florian Daniel, Giuseppe Pozzi

Methods for Enabling Recovery Actions in Ws-BPEL 219
Stefano Modafferi, Eugenio Conforti

BPEL Processes Matchmaking for Service Discovery 237
Juan Carlos Corrales, Daniela Grigori, Mokrane Bouzeghoub

Evaluation of Technical Measures for Workflow Similarity Based on a Pilot Study 255
Andreas Wombacher

Dynamic and Adaptable Workflows

Evolution of Process Choreographies in DYCHOR 273
Stefanie Rinderle, Andreas Wombacher, Manfred Reichert

Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows 291
Michael Adams, Arthur H.M. ter Hofstede, David Edmond, Wil M.P. van der Aalst

Change Mining in Adaptive Process Management Systems 309
Christian W. Günther, Stefanie Rinderle, Manfred Reichert, Wil van der Aalst

Services Metrics and Pricing

A Link-Based Ranking Model for Services 327
Camelia Constantin, Bernd Amann, David Gross-Amblard

Quality Makes the Information Market 345
B. van Gils, H.A. (Erik) Proper, P. van Bommel, Th. P. van der Weide

Bid-Based Approach for Pricing Web Service 360
Inbal Yahav, Avigdor Gal, Nathan Larson

Formal Approaches to Services

Customizable-Resources Description, Selection, and Composition: A Feature Logic Based Approach 377
Yacine Sam, François-Marie Colonna, Omar Boucelma

Defining and Modelling Service-Based Coordinated Systems 391
Thi-Huong-Giang Vu, Christine Collet, Genoveva Vargas-Solar

Web Service Mining and Verification of Properties: An Approach Based on Event Calculus 408
Mohsen Rouached, Walid Gaaloul, Wil M.P. van der Aalst, Sami Bhiri, Claude Godart

Trust and Security in Cooperative IS

Establishing a Trust Relationship in Cooperative Information Systems 426
Julian Jang, Surya Nepal, John Zic

A Unifying Framework for Behavior-Based Trust Models 444
Christian von der Weth, Klemens Böhm

A WS-Based Infrastructure for Integrating Intrusion Detection Systems in Large-Scale Environments 462
José Eduardo M.S. Brandão, Joni da Silva Fraga, Paulo Manoel Mafra, Rafael R. Obelheiro

P2P Systems

An Adaptive Probabilistic Replication Method for Unstructured P2P Networks 480
Dimitrios Tsoumakos, Nick Roussopoulos

Towards Truthful Feedback in P2P Data Structures 498
Erik Buchmann, Klemens Böhm, Christian von der Weth

Efficient Peer-to-Peer Belief Propagation 516
Roman Schmidt, Karl Aberer

Collaborative Systems Design and Development

Designing Cooperative IS: Exploring and Evaluating Alternatives 533
Volha Bryl, Paolo Giorgini, John Mylopoulos

Natural MDA: Controlled Natural Language for Action Specifications on Model Driven Development 551
Luciana N. Leal, Paulo F. Pires, Maria Luiza M. Campos, Flávia C. Delicato

Managing Distributed Collaboration in a Peer-to-Peer Network 569
Michael Higgins, Stuart Roth, Jeff Senn, Peter Lucas, Dominic Widdows

Collaborative Systems Development

Developing Collaborative Applications Using Sliverware 587
Seth Holloway, Christine Julien

A Framework for Building Collaboration Tools by Leveraging Industrial Components 605
Du Li, Yi Yang, James Creel, Blake Dworaczyk

Evaluation of a Conceptual Model-Based Method for Discovery of Dependency Links 625
Darijus Strasunskas, Sari Hakkarainen

Cooperative IS Applications

Advanced Recommendation Models for Mobile Tourist Information 643
Annika Hinze, Saijai Junmanee

Keeping Track of the Semantic Web: Personalized Event Notification ... 661
Annika Hinze, Reuben Evans

A Gestures and Freehand Writing Interaction Based Electronic Meeting Support System with Handhelds ... 679
Gustavo Zurita, Nelson Baloian, Felipe Baytelman, Mario Morales

Ontologies, Databases and Applications of Semantics (ODBASE) 2006 International Conference

ODBASE 2006 International Conference (Ontologies, DataBases, and Applications of Semantics) PC Co-chairs' Message ... 697

Keynote

SomeWhere: A Scalable Peer-to-Peer Infrastructure for Querying Distributed Ontologies ... 698
M.-C. Rousset, P. Adjiman, P. Chatalic, F. Goasdoué, L. Simon

Foundations

Querying Ontology Based Database Using OntoQL (An Ontology Query Language) ... 704
Stéphane Jean, Yamine Aït-Ameur, Guy Pierra

Description Logic Reasoning with Syntactic Updates ... 722
Christian Halashek-Wiener, Bijan Parsia, Evren Sirin

From Folksologies to Ontologies: How the Twain Meet ... 738
Peter Spyns, Aldo de Moor, Jan Vandenbussche, Robert Meersman

Transactional Behavior of a Workflow Instance ... 756
Tatiana A.S.C. Vieira, Marco A. Casanova

Metadata

An Open Architecture for Ontology-Enabled Content Management Systems: A Case Study in Managing Learning Objects ... 772
Duc Minh Le, Lydia Lau

Ontology Supported Automatic Generation of High-Quality Semantic Metadata ... 791
Ümit Yoldas, Gábor Nagypál

Brokering Multisource Data with Quality Constraints 807
Danilo Ardagna, Cinzia Cappiello, Chiara Francalanci, Annalisa Groppi

Design

Enhancing the Business Analysis Function with Semantics 818
Sean O'Riain, Peter Spyns

Ontology Engineering: A Reality Check 836
Elena Paslaru Bontas Simperl, Christoph Tempich

Conceptual Design for Domain and Task Specific Ontology-Based Linguistic Resources 855
Antonio Vaquero, Fernando Sáenz, Francisco Alvarez, Manuel de Buenaga

Ontology Mappings

Model-Driven Tool Interoperability: An Application in Bug Tracking ... 863
Marcos Didonet Del Fabro, Jean Bézivin, Patrick Valduriez

Reducing the Cost of Validating Mapping Compositions by Exploiting Semantic Relationships 882
Eduard Dragut, Ramon Lawrence

Using Fuzzy Conceptual Graphs to Map Ontologies 891
David Doussot, Patrice Buche, Juliette Dibie-Barthélemy, Ollivier Haemmerlé

Formalism-Independent Specification of Ontology Mappings – A Metamodeling Approach 901
Saartje Brockmans, Peter Haase, Heiner Stuckenschmidt

Information Integration

Virtual Integration of Existing Web Databases for the Genotypic Selection of Cereal Cultivars 909
Sonia Bergamaschi, Antonio Sala

SMOP: A Semantic Web and Service Driven Information Gathering Environment for Mobile Platforms 927
Özgür Gümüs, Geylani Kardas, Oguz Dikenelli, Riza Cenk Erdur, Ata Önal

Integrating Data from the Web by Machine-Learning Tree-Pattern Queries 941
Benjamin Habegger, Denis Debarbieux

Agents

HISENE2: A Reputation-Based Protocol for Supporting Semantic Negotiation 949
Salvatore Garruzzo, Domenico Rosaci

An HL7-Aware Multi-agent System for Efficiently Handling Query Answering in an e-Health Context 967
Pasquale De Meo, Gabriele Di Quarto, Giovanni Quattrone, Domenico Ursino

PersoNews: A Personalized News Reader Enhanced by Machine Learning and Semantic Filtering 975
Evangelos Banos, Ioannis Katakis, Nick Bassiliades, Grigorios Tsoumakas, Ioannis Vlahavas

Contexts

An Ontology-Based Approach for Managing and Maintaining Privacy in Information Systems 983
Dhiah el Diehn I. Abou-Tair, Stefan Berlik

Ontology-Based User Context Management: The Challenges of Imperfection and Time-Dependence 995
Andreas Schmidt

Moving Towards Automatic Generation of Information Demand Contexts: An Approach Based on Enterprise Models and Ontology Slicing 1012
Tatiana Levashova, Magnus Lundqvist, Michael Pashkin

Similarity and Matching

Semantic Similarity of Ontology Instances Tailored on the Application Context 1020
Riccardo Albertoni, Monica De Martino

Finding Similar Objects Using a Taxonomy: A Pragmatic Approach 1039
Peter Schwarz, Yu Deng, Julia E. Rice

Towards an Inductive Methodology for Ontology Alignment Through Instance Negotiation 1058
Ignazio Palmisano, Luigi Iannone, Domenico Redavid, Giovanni Semeraro

Combining Web-Based Searching with Latent Semantic Analysis to Discover Similarity Between Phrases 1075
Sean M. Falconer, Dmitri Maslov, Margaret-Anne Storey

A Web-Based Novel Term Similarity Framework for Ontology Learning 1092
Seokkyung Chung, Jongeun Jun, Dennis McLeod

Author Index 1111

GADA 2006 International Conference (Grid Computing, High-Performance and Distributed Applications) PC Co-chairs' Message

This volume contains the papers presented at GADA 2006, the International Conference on Grid Computing, High-Performance and Distributed Applications. The purpose of the GADA series of conferences, held in the framework of the OnTheMove Federated Conferences (OTM) federated conferences, is to bring together researchers, developers, professionals and students in order to advance research and development in the, areas of grid computing and distributed systems and applications. This years conference was in Montpellier, France, from November 2 to November 3.

Within the OTM framework, the GADA workshop arised in 2004, as a forum for researchers in grid computing, whose aim was to extend their background on this area and more specifically for those who used grid environments for managing and analyzing data. Both GADA 2004 and GADA 2005 were very successful events, due to the large number of high-quality papers received in both editions as well as the brain-storming of experiences and ideas interchanged in the associated forums. As final reward for all this hard work, GADA was upgraded as a conference within the OTM 2006 Federated Conferences and Workshops.

GADA 2006 covered a broader set of disciplines in the field of distributed and high-performance computing, although grid computing kept a key role in the set of main topics of the conference. A grid is a collection of processing resources and people performing cooperative tasks. By pooling federated assets, grids provide a single point of access to powerful distributed resources. Users can literally submit thousands of jobs at a time without knowing where they will run. Innovative applications continue to spread both in research and business. The objective of grid computing is the complete integration of heterogeneous distributed computing systems and data resources with the aim of providing a global and decentralized computing space. Researchers working to solve many of the most difficult scientific problems have long understood the potential of such shared distributed computing systems. The achievement of this goal involves revolutionary changes in the field of computation, because it enables resource-sharing across networks, data being one of the most important. Thus, data access, management and analysis within grid and distributed environments were also a main part of the symposium.

Besides the traditional set of topics of previous GADA meetings, high-performance and distributed applications were tackled in an explicit manner within GADA 2006. These research areas and grid computing have many commonalities which can and must be dealt with. Therefore, the main goal of GADA 2006 was to provide a framework in which a community of researchers, developers and users can exchange ideas and works related to grids, high-performance and distributed applications and systems.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1117–1118, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The 25 revised full papers presented were carefully reviewed and selected from a total of 76 submissions with an acceptance rate of 33%. Each paper has been reviewed by 2 reviewers and totally 71 reviewers were involved in the review process of GADA 2006. Topics of the accepted papers include computational and data grids, cluster computing, parallel applications, collaboration technologies, agent architectures for grid and distributed environments, Semantic Grid, and security in distributed systems.

We would like to thank the members of the Program Committee for their hard and expert work. We would also like to thank the OTM General Co-chairs, the workshop organizers, the external reviewers, the authors, and the local organizers for their contributions to the success of the conference.

August 2006

Pilar Herrero, Universidad Politécnica de Madrid, Spain
María S. Pérez, Universidad Politécnica de Madrid, Spain
Domenico Talia, Universitá della Calabria, Italy
Albert Zomaya, The University of Sydney, Australia

Data-Oriented Distributed Computing for Science: Reality and Possibilities

Daniel S. Katz[1,2,⋆], Joseph C. Jacob[2], Peggy P. Li[2], Yi Chao[2], and Gabrielle Allen[1]

[1] Center for Computation & Technology, Louisiana State University
d.katz@ieee.org
[2] Jet Propulsion Laboratory, California Institute of Technology

Abstract. As is becoming commonly known, there is an explosion happening in the amount of scientific data that is publicly available. One challenge is how to make productive use of this data. This talk will discuss some parallel and distributed computing projects, centered around virtual astronomy, but also including other scientific data-oriented realms. It will look at some specific projects from the past, including Montage[1], Grist[2], OurOcean[3], and SCOOP[4], and will discuss the distributed computing, Grid, and Web-service technologies that have successfully been used in these projects.

1 Introduction

This talk will explore a pair of related questions in computer science and computational science: "What is a Grid?" and "How can the concepts that are sometimes described as a Grid be used to do science, particularly with large amounts of data?" The reason these two questions are interesting is that the amount of scientific data that is available is exploding, and as both the data itself and the computing used to obtain knowledge from it are distributed, it is important that researchers understand how other researchers are approaching these issues.

2 Definitions and Meanings

A recently written paper [1] that attempts to understand what Grid researchers mean when they say "Grid" mentions that the term Grid was introduced in 1998 and says that since that time, many technological changes have occurred in both hardware and software. The main purpose of the paper was to take a snapshot of how Grid researchers define the Grid by asking them to:

⋆ Corresponding author.
[1] http://montage.ipac.caltech.edu/
[2] http://grist.caltech.edu/
[3] http://OurOcean.jpl.nasa.gov/
[4] http://scoop.sura.org/

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1119–1124, 2006.
© Springer-Verlag Berlin Heidelberg 2006

> Try to define what are the important aspects that build a Grid, what is distinctive, and where are the borders to distributed computing, Internet computing etc.

More than 170 researchers were contacted, and more than 40 contributed material that considered distinct enough to be of value to the paper. The conclusion of the paper is that the research community has a fairly homogeneous view of grids, with a few outlying opinions, but with far less diversity of opinions than that expressed in a similar survey among industrial information technology (IT) leaders. One interesting section of this paper (3.2.1) discusses the fact that some of the people surveyed considered Grids a general form of distributed computing while other considered Grids a special form of distributed computing, and some said that there is no line between Grids and distributed computing. The next section (3.2.2) states that Grid services are Web services with additional features. To explore these possible distinctions, this talk will examine how some real applications make use of the Grid and of Web services.

3 Projects

3.1 Montage

Montage [2] is a set of software packages that can be used in building astronomical image mosaics. Users see Montage either as a set of software executables that can be run on a single processor or on a parallel system or as a portable, compute-intensive service. In either case, Montage delivers science-grade custom mosaics. *Science-grade* in this context requires that terrestrial and instrumental features are removed from images in a way that can be described quantitatively; *custom* refers to user-specified parameters of projection, coordinates, size, rotation and spatial sampling. Fig. 1 shows examples of two mosaics, one without and one with background rectification.

This talk will discuss the performance of the parallel Montage application, and will compare it with the performance of the Montage portal on the same problems. This will show that at least some Grid software is both sufficiently mature and sufficiently high-performance to be useful to a set of real scientific applications.

3.2 Grist

The Grist project [3] aims to establish a mechanism whereby grid services for the Astronomy community may be quickly and easily deployed on the NSF TeraGrid, while meeting the requirements of both the service users (the astronomy virtual observatory community) and the grid security administrators. In collaboration with the TeraGrid and National Virtual Observatory (NVO), the Grist project is building the NVO Extensible Secure Scalable Service Infrastructure (NESSSI), a service oriented architecture with the following characteristics:

- Services are created, deployed, managed, and upgraded by their developers, who are trusted users of the compute platform where their service is deployed.
- Service jobs may be initiated with Java or Python client programs run on the command line or with a web portal called Cromlech.
- Service clients are authenticated with "graduated security" [4], which scales the size of jobs that are allowed with the level of authentication of the user. The Clarens service infrastructure [5] serves as the "gatekeeper" by managing user certificates.
- Access to private data, such as that from the Palomar-QUEST survey, is restricted via a proposed "Visa" system, which examines user certificates to determine who is authorized to access each dataset supported by a service.

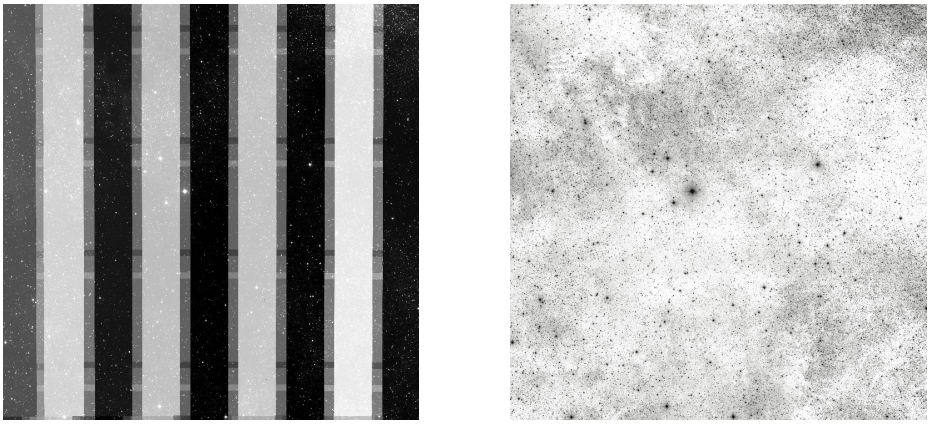


Fig. 1. A Montage-produced 2MASS mosaic of a square degree of sky near the Galactic center (left, without background rectification; right, with background rectification)

Grist is using NESSSI to build the NVO Hyperatlas, which supports multi-wavelength science via the construction of standard image plates at various wavelengths and pixel sizes, and associated services to construct and visualize these plates. In support of this objective, the Grist team is deploying services on the TeraGrid for computing image mosaics and image cutouts. The image cutout service will be scaled up to compute massive numbers of cutouts in a single request on the TeraGrid. This multi-cutout service will provide input data for a galaxy morphology study to be conducted with a science partner.

This talk will discuss NESSSI and how it can be used to build services, including Montage, that use TeraGrid resources while properly handling authentication and authorization issues, either from a portal or from a client application.

3.3 OurOcean

OurOcean [6] is a JPL project that has built a portal to enable users to easily access ocean science data, run data assimilation models, and visualize both data

and models. The concept of OurOcean is to allow users with minimal resource requirements to access data and interact with models. Currently, OurOcean provides both real-time and retrospective analysis of remote sensing data and ocean model simulations in the Pacific Ocean. OurOcean covers the U.S. West Coastal Ocean with focused areas around Southern California, Central and Northern California, and Prince William Sound in Alaska.

OurOcean consists of a data server, a web server, a visualization server, and an on-demand server, as shown in Fig. 2. The data server is in charge of real-time data retrieval and processing. Currently, our data server manages a MySQL database and a 5 terabyte RAID disk. The web server is an apache2 server with Tomcat running on a Linux workstation. The web server dispatches user requests to the visualization server and the on-demand server to generate custom plots or invoke on-demand modeling. The visualization server consists of a set of plotting programs written in GMT and Matlab. In addition, Live Access Server (LAS) [7] is used to provide subsetting and on-the-fly graphics for 3D time-series model output. Finally, the on-demand server manages the custom model runs and the computing resources. OurOcean has a 12-processor SGI Origin 350 and a 16-processor SGI Altix cluster as modeling engines.

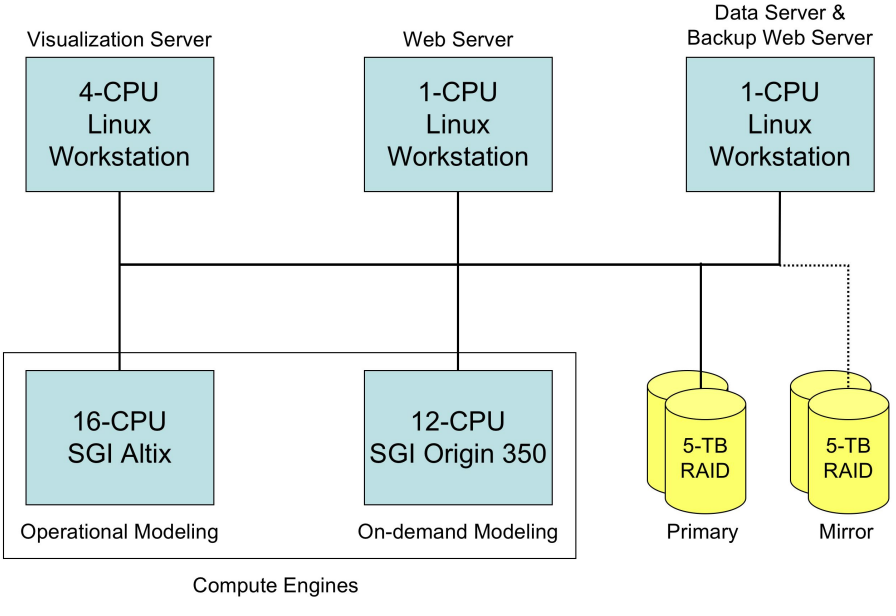


Fig. 2. The OurOcean hardware architecture

3.4 SCOOP

Similarly to OurOcean, the SURA Coastal Ocean Observing and Prediction (SCOOP) program [8] is developing a distributed laboratory for coastal research and applications. The project vision is to provide tools to enable communities

of scientists to work together to advance the science of environmental prediction and hazard planning for the Southeast US coast. To this end, SCOOP is building a cyberinfrastructure using a service-oriented architecture, which will provide an open integrated network of distributed data archives, computer models, and sensors. This cyberinfrastructure includes components for data archiving, integration, translation and transport, model coupling and workflow, event notification and resource brokering.

SCOOP is driven by three user scenarios, all of which involve predicting the coastal response to extreme events such as hurricanes or tropical storms: (i) ongoing real-time predictions, (ii) retrospective analyses, (iii) event-driven ensemble predictions. From these use cases, event-driven ensemble predictions provides the most compelling need for Grids, and provides new challenges in scheduling and policies for emergency computing.

Following an initial hurricane advisory provided by the National Hurricane Center, the SCOOP system will construct an appropriate ensemble of different models to simulate the coastal effect of the storm. These models, driven by real time meteorological data, provide estimates of both storm surge and wave height. The ensemble covers different models (e.g. surge and wave), areas (e.g. the entire southeast or higher resolution regions), wind forcing (e.g. NCEP, GFDL, MM5, and analytically generated winds), and other parameters. The ensemble of runs needs to be completed in a reliable and timely manner, to analyze results and provide information that could aid emergency responders.

Core working components of the SCOOP cyberinfrastructure include a transport system built primarily on Local Data Manager (LDM) [9], a reliable data archive [10], a catalogue with web service interfaces to query information about models and data locations, client tools for data location and download (getdata) [11] and various coastal models (including ADCIRC, CH3D, ELCIRC, SWAN, WAM, WW3) which are deployed using Globus GRAM. An application portal built with GridSphere [12] provides user interfaces to the tools, and the results are disseminated though the OpenIOOS site[5]. Current development in SCOOP is focused on model scheduling, deployment and monitoring, incorporating on-demand priority scheduling using SPRUCE [13].

4 Conclusion

As the projects that have been discussed show, there are many alternative methods to effectively perform scientific calculations in a distributed computing environment. Common aspects of these projects is that they have involved a mix of computer scientists and application scientists. The combined actions of members of these communities working toward a common vision seems to often lead to a successful project. To a large extent, the names and specific technologies used to define the environment are not important, except in how they allow multiple people to effectively discuss and extend distributed scientific computing.

[5] http://www.openioos.org/

References

1. Stockinger, H.: Defining the Grid: A Snapshot on the Current View. J. of Super-Computing (Spec. Issue on Grid Computing). Submitted June 26, 2006.
2. Jacob, J. C., Katz, D. S., Berriman, G. B., Good, J., Laity, A. C., Deelman, E., Kesselman, C., Singh, G., Su, M.-H., Prince, T. A., Williams, R.: Montage: A Grid Portal and Software Toolkit for Science-Grade Astronomical Image Mosaicking. Int. J. of Computational Science and Engineering. (to appear)
3. Jacob, J. C., Williams, R., Babu, J., Djorgovski, S. G., Graham, M. J., Katz, D. S., Mahabal, A., Miller, C. D., Nichol, R., Vanden Berk, , D. E., Walia, H.: Grist: Grid Data Mining for Astronomy. Proc. Astronomical Data Analysis Software & Systems (ADASS) XIV (2004)
4. Williams, R., Steenberg, C., Bunn, J.: HotGrid: Graduated Access to Grid-based Science Gateways. Proc. IEEE SC|04 Conf. (2004)
5. Steenberg, C., Bunn, J., Legrand, I., Newman, H., Thomas, M., van Lingen, F., Anjum, A., Azim, T.: The Clarens Grid-enabled Web Services Framework: Services and Implementation. Proc. Comp. for High Energy Phys. (2004)
6. Li, P., Chao, Y., Vu, Q., Li, Z., Farrara, J., Zhang, H., Wang, X.: OurOcean - An Integrated Solution to Ocean Monitoring and Forecasting. Proc. MTS/IEEE Oceans'06 Conf. (2006)
7. Hankin S., Davison, J.,Callahan, J. , Harrison, D.E., O'Brien, K.: A Configurable Web Server for Gridded Data: a Framework for Collaboration. Proc. 14th Int. Conf. on IIPS for Meteorology, Oceanography, and Hydrology (1998) 417–418
8. Bogden, P., Allen, G., Stone, G., Bintz, J., Graber, H., Graves, S., Luettich, R., Reed, D., Sheng, P., Wang, H., Zhao, W.: The Southeastern University Research Association Coastal Ocean Observing and Prediction Program: Integrating Marine Science and Information Technology. Proc. MTS/IEEE Oceans'05 Conf. (2005)
9. Davis, G. P., Rew, R. K.: The Unidata LDM: Programs and Protocols for Flexible Processing of Data Products. Proc. 10th Int. Conf. on IIPS for Meteorology, Oceanography, and Hydrology (1994) 131–136
10. MacLaren, J., Allen, G., Dekate, C., Huang, D., Hutanu, A., Zhang, C.: Shelter from the Storm: Building a Safe Archive in a Hostile World. Lecture Notes in Computer Science **3752** (2005) 294–303
11. Huang, D., Allen, G., Dekate, C., Kaiser, H., Lei, H., MacLaren, J.: getdata: A Grid Enabled Data Client for Coastal Modeling. Proc. High Performance Comp. Symp. (2006)
12. Zhang, C., Dekate, C., Allen, G., Kelley, I., MacLaren, J.: An Application Portal for Collaborative Coastal Modeling. Concurrency and Computation: Practice and Experience, **18** (2006) 1–11
13. Beckman, P., Nadella, S., Beschastnikh, I., Trebon, N.: SPRUCE: Special PRiority and Urgent Computing Environment. University of Chicago DSL Workshop (2006)

From Intelligent Content to Actionable Knowledge: Research Directions and Opportunities Under the EU's Framework Programme 7, 2007-2013

Stefano Bertolo

European Commission, DG INFSO/E2
stefano.bertolo@ec.europa.eu

Abstract. Since many human activities depend on the creation, use and transmission of symbolic information advances in our ability to produce and use such information (semi)automatically can be expected to have great impact on society and economic development. For this reason, this objective has been proposed as one of the main activities under Framework programme 7 (FP7), the next cycle of EU research and development activities and funding to run through 2007-2013. This paper gives a broad overview of the place of content and knowledge in FP7 and discusses several specific lines of research that have been identified as particularly important and promising.

Motivation

Even an extremely superficial look at the history of technology shows that as soon as a human activity becomes of economic or societal significance it becomes important to make it more efficient. Depending on the circumstances this might mean either allowing for greater output to result from the same amount of activity (in a given amount of time you could move more rocks with a lever than you could with your hands) or by allowing more people to participate in the activity by reducing its physical or mental demands (in the not so distant past a professional typesetter would have been required to typeset the paper you are reading: contributors to this volume however just used the supplied template after importing it in their word processor or typesetting program). This sounds almost too trivial to mention. Consider however that in developed countries, and in the EU in particular, a large and growing number (and proportion) of people is daily engaged in the process of creating, storing, publishing, accessing and reasoning about what one could in the most general terms describe as content (music, pictures, videos, …) and knowledge (descriptions of reality that can be used to summarise past experiences and/or draw conclusions that go beyond them). But while we have efficient tools to, say, lift crates or cut metal to a desired shape and infrastructure to extract and transport energy, our existing tools for managing content and knowledge (semi)automatically are, by comparison, still extremely primitive (with the notable exception of textual indexing and search). When you come back from a trip and your digital camera disgorges on your hard disk dozens of sequentially numbered pictures you still have to manually inspect them to

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1125 – 1131, 2006.
© Springer-Verlag Berlin Heidelberg 2006

make better sense of their content. What is a minor inconvenience to an amateur photographer becomes a major bottleneck in the work of a professional reporter or newsroom editor. Similarly, while doctors dutifully keep records of their patients' background and conditions, today only highly trained medical professionals are able to compare them meaningfully to draw appropriate diagnostic conclusions. If some of these comparisons could be carried out automatically and brought to the attention of the physician when appropriate she could devote more of her time to other aspects of her practice. These two simple examples show how different our activities would be if we could create a global infrastructure to do for human knowledge and content what early 20th century industrialization did for electricity, allowing it to be produced, stored, transmitted and used reliably, efficiently and according to standards that could be relied upon to build applications and tools. This vision is the motivation behind FP7 plans for content and knowledge research and development activities.

A Brief Overview of Framework Programme 7

Multi-year programmes called Framework Programmes have since 1984 been the European Union's main instrument for funding research and development, as per article 166 of the Treaty establishing the European Community. As the current Framework Programme 6[1] (2002-2006) draws to an end, the European institutions have been at work defining Framework Programme 7[2] with the Commission's proposals going through the co-decision procedure for approval and adoption by the European Parliament and Council. At the date of writing the co-decision procedure is still in process and is expected to come to a conclusion at the end of 2006, at which time FP7 will be formally launched and its work programme published and organised into a series of objectives that will form the object of calls for proposals. According to the most recent *indicative* figures[3], in FP7 €9110M will be devoted to cooperation activities in information and communication technologies.[4] It is anticipated that approximately €250M will be devoted specifically to content and knowledge themes for the period 2007-2008, with three calls for proposals likely to be published in 2007 and covering topics such as digital libraries, technology-enhanced learning, sensing and cognition, networked multimedia processing and distribution, knowledge representation and induction and inference. The topics selected are the result of articulating the vision described in the previous section against the background of the Commission's i2010 policy initiatives[5] with their strong emphasis on growth and development, the recommendations received by various bodies of experts such as the

[1] http://ec.europa.eu/research/fp6/index_en.cfm
[2] http://cordis.europa.eu/fp7/
[3] See page 8 of the Council of the European Union's political agreement of 24 July 2006 available at http://www.consilium.europa.eu/ueDocs/cms_Data/docs/pressData/en/intm/90654.pdf
[4] FP7 also contains a programme called "Ideas" to support "investigator-driven" research projects. Investigators in content and knowledge seeking individual (as opposed to cooperative project) grants are certainly encouraged to familiarise themselves with the programme. Details are available at http://cordis.europa.eu/fp7/ideas.htm.
[5] http://europa.eu.int/information_society/eeurope/i2010/index_en.htm

IST Advisory Group (ISTAG)[6], very extensive consultations with researchers and technology developers from all over the European Union[7] and constant monitoring of global technology trends to identify opportunities and gaps.

FP7 Research Directions in Content and Knowledge

While at the time of writing they have not yet been finalised in a formal work programme, it is safe to anticipate that FP7 directions in content and knowledge can be seen as an organic evolution of the FP6 directions[8] with continuity assured for hard problems that are still considered open, acceleration in areas in which external developments and dynamics invite more decisive action and new lines of activities for topics whose importance and promise could not be fully appreciated just a few months ago. These are not necessarily in alternative to one another and indeed novel lines of activity often suggest new lines of attack for long standing problems as is the case for social network analysis and knowledge management.

Acceleration and novelty are due to the enormous changes in the production and consumption of content and knowledge that have taken place from just a few years ago. While past Framework Programme efforts have concentrated either on manually encoded formal knowledge or knowledge extracted from textual sources, it has become evident that these are not going to be the only or even the prevalent forms in which content and knowledge will exist and be consumed.

The first trend that can be very readily observed is the explosion in the availability of multimedia content[9] and the fact that much content is produced and remixed by non-professionals and accessed/consumed on devices of great variability in terms of networking capability, display resolution and controls. This brings about problems of scale, usability and democratisation of the tool chain that need to be addressed.

The second trend, which we are already witnessing in the most advanced scientific laboratories but can soon be expected to spread to many other environments is the proliferation of data that has been produced by instruments as opposed to humans. Such data are in need of interpretation and integration and, for reasons discussed below, they create novel demands on knowledge representation and reasoning.

A final clear trend that has guided the formulation of the FP7 content and knowledge objectives is the speed at which distributed (e.g. peer to peer) and socially enhanced content management applications have established themselves as successful solutions[10], throwing into sharper relief than it was possible in the past issues such as trust and provenance, personalisation and contextualisation.

---

[6] http://cordis.europa.eu/ist//istag.htm

[7] See http://cordis.europa.eu/ist/kct/fp7_consul.htm for a complete record of consultations in the specific domain of content and knowledge.

[8] See http://cordis.europa.eu/ist/kct/strategic.htm for a description of the FP6 objectives in this domain and http://cordis.europa.eu/ist/kct/projects.htm for a list of the projects that resulted from the corresponding calls for proposal.

[9] See issue 15.04 or Wired Magazine for a recent survey: http://www.wired.com/wired/archive/14.05/guide.html

[10] See http://www.alexaholic.com/sethgodin for an up-to-date survey and interesting quantitative analysis of more than a thousand web-based companies that operate according to these principles, often collectively referred to as "web2.0" companies.

Before delving into the specific lines of research that have been identified as central to FP7, a few methodological remarks that apply to all of them are in order. Two of the corner stones of the scientific method are accuracy of measurement and replicability. In the context of upcoming FP7 activities accuracy of measurement means that proposed development activities should come with clear plans as to what aspects of the system's performance will be measured and how the measurements will provide evidence that the benefits of the system outweigh its costs. This is to cover not only algorithmic performance but also issues of usability. Replicability means that developers are expected to demonstrate their system's performance in conditions that can in principle be replicated elsewhere by other research groups so as to invite comparison and competition and provide the means for progress tracking and cost-benefit analyses. Additionally, those conditions will need to be realistic, i.e. address data collections or user populations of the same order of magnitude expected in the eventual deployment of the technology developed. This means that in FP7 projects in content and knowledge, securing access to realistic data collections and relentlessly benchmarking and testing against those collections can be assumed to be as important as working on ideas that are scientifically or technologically novel. Indeed, there will be opportunities for projects whose *main* goal is sophisticated benchmarking and testing of technologies developed by others.

We are now ready to give a short description of some specific research themes, their motivation and their expected outcome. They are listed in no particular order.

Advanced Multimedia Authoring Environments

As mentioned above, one of the obvious trends of the last couple of years is the enormous growth of user produced multimedia content that is presumably the effect of both the increased availability of digital (video)cameras and the ease with which the materials so produced can be uploaded and shared on the internet. Between creating/capturing and sharing there are however various forms of planning and editing that need to be better supported. These go from various forms of image correction functionalities to enhance the appearance of people or objects, to novel forms of storyboarding, to the remixing of content from previously existing materials, to the creation of animated models from real life images. Recognizing objects of interest in digital content and making them available for manipulation in an editing environment according to their specific type (much as programming constructs can be recognised and manipulated in an integrated development environment) would certainly benefit professional creators of content. Activities under this theme however will also be required to be mindful of two social dimensions. The first one is the democratisation of the tool chain: bringing these advanced content editing functionalities within the technical reach of casual users can be expected to have a positive effect on the widespread creation of content of better quality. Once this content is created, it is important that it be shared and searched effectively both locally by the user as a form of personal data management and in a creative social environment. The second dimension is thus the creation of open standards and tools for (semi)automatically metadata annotation in support of more sophisticated forms of both symbolic and similarity based multimedia search.

Collaborative Multimedia Workflow Environments

More directly targeted in support of organisations and professional creators of content will be activities to develop collaborative workflow environments in support the entire lifecycle of media and enterprise content. These should go from the planned acquisition of raw materials (especially from legacy collections) to the versioning, packaging and repurposing of complex products allowing for content and annotations produced on a given application to be saved and stored according to open standards to be reused by other applications in the workflow. The technology developed should be able to address the needs of extremely large multimedia archives, appropriately addressing physical storage and indexing schemes. This is considered essential to ensure a transition of the technology from the research to the actual deployment stage in a commercial environment or a digital library. One of the dimensions of repurposing will naturally be the automatic preparation of content for various target audiences with widely varying bandwidth and display capabilities. Under this dimension, attention will be given to efficient techniques for selecting and executing multimedia summarisation and encoding schemes based on the properties of the target device and the psychology of human attention and perception. This will allow for salient features of multimedia segments to be identified and for the non-salient features to be compressed with no loss in the overall experience at the point of consumption.

Advanced Publishing and Consumption of Content

The success of the two previous lines of activities produces a scenario in which content objects will come into existence and be distributed with a considerable amount of actionable knowledge about themselves (what they represent, when and where they have been captured/created, …). This creates two significant opportunities at the point of consumption that will be explored in FP7. The first one is to use the knowledge embedded into the object in interaction with knowledge of the user and emergent ambient intelligence in support of the most effective interactive multimodal experience. In this scenario the user would select a content object and the object would determine how to best display itself and expose its controls, potentially co-opting various forms of hardware detected in its environment based on its understanding of the user (her goals in the interaction, her language and cultural preferences, …) and her current circumstances (location, time, …). The second opportunity is to allow user interactions to flow back into the object and add to its intelligence in a sort of ecological validation. The object will be able to unobtrusively detect and record if and how it is being interacted with (looked at, clicked at, resized, skipped …) or even what emotions it has elicited. This information, when made available to the content creators, will allow them to edit the object for improved effect, just as web designers today improve the usability of their sites after analysing click-through logs or eye-tracking heatmaps. Both opportunities clearly depend on accessing or collecting a non-trivial amount of personal information on content consumers and their behaviour. Projects addressing the delivery of smart content objects will thus need to appropriately address consumer privacy issues, as discussed in the next section.

Social and Economic Studies

The great increase of user produced content available on the internet has been one of the most noticeable trends of the past few months but it is fair to say that it has been rather serendipitous with even the largest commercial players reacting to it rather than breaking new ground. The time seems ripe to undertake a systematic study of what economic and social factors act as catalyst or inhibitors to the production and distribution of content and what information technologies would be needed to enhance the effects of the catalysts and reduce the effects of the inhibitors. For example, as seen in the previous section, enormous benefits could be reaped from systematically collecting user feedback on the consumption of content but at the same time users cannot legitimately be expected to surrender this information indiscriminately. This line of research will thus encourage work on privacy preserving data-mining algorithms for collecting this feedback from an *aggregate* analysis of social and user-device interactions. Similarly, users may decide to produce more content, or more content of a certain valuable type, if they could rely on certain forms of reward (not necessarily financial) or feel that their repurposing or remixing of pre-existing content is fully legitimate. Determining the appropriate mixture of incentives will require empirical studies in social psychology and economics. Finally, this line of activities will also foster community building to encourage multi-disciplinary approaches and a more effective dialogue between suppliers and users of technology for a faster and more effective uptake of research results.

Semantic Foundations

Formal knowledge representation remains very much central to the Framework Programme's research efforts but its emphasis will be focused towards demands that have been identified from an analysis of the types and scale of the data that it is expected to integrate. The massive amounts of structured information coming on line as the valuable output of government bodies (e.g. census and cadastral data) research labs (a trend most obvious in the life sciences) or any other source, will make it ever more important to build and manage the semantic 'glue' that will allow reasoning over data stored in different databases, whose semantics are not directly aligned. This is indeed one of the central insights of the *Semantic Web* vision. This requires methods for aligning ontologies and support reasoning over distributed knowledge sources. These large amounts of structured data moreover offer an obvious opportunity for theory induction, i.e. the ability to reach formally represented general conclusions from the analysis of a large number of individual observations. The FP7 goal for semantic integration and theory induction is to produce programs that routinely outperform trained professionals in their ability to reach important conclusions and produce insightful and novel hypotheses from the data at hand. The very heterogeneity of those sources however will mean that noise and inconsistencies will be inevitable: novel techniques in reasoning and machine learning will be needed to overcome them, likely leading to an integration of symbolic and probabilistic knowledge representation. Existing knowledge representation formalisms will also need to be extended to be able to represent and reason about objects that exist and change in time, and processes that unfold and branch over time. A second source of

massive amounts of data that is just behind the horizon and for which appropriate knowledge representation integration solutions don't yet exist are sensors and physical objects endowed with various form of radio identifiers and locators. In this scenario too we will face massive streams of structured data that can be queried for the occurrence of certain events and from which much can be induced. Developing the technology for doing this efficiently and on a massive scale will be an important objective in FP7.

Advanced Knowledge Management Systems

The previous section addresses the knowledge representation problems that need to be solved in environments where data, while potentially noisy, are also highly structured and typically interpretable against the background of mature sciences or well defined systems. Most organisations however operate in environments where those characteristics do not apply and where knowledge is typically stored in textual or multimedia documents of arbitrary format and content. These organisations need technologies capable of bootstrapping unstructured knowledge to ever more structured integrated repository by making effective use of lightweight annotations made by humans (as done today by 'folksonomies' in successful social bookmarking applications) and by observing and data-mining their interactions (the same concerns on personal privacy discuss above also apply here). The goal is to progressively integrate an organisation's implicit knowledge into its formal business processes and to be able to expose both to third parties in the dynamic creation of virtual organisations as required by common business objectives. The associated security concerns will be addresses by the definition, verification and automatic implementation of formal policies designed to regulate access to data sources or other kinds of organisational resources. Formal policies are indeed a prime example of the need for extending existing knowledge representation formalisms to include rules and temporal reasoning. Progress in such advanced knowledge management systems will have to be tested in real-life settings with particular attention to the integration of legacy systems and usability.

Conclusion

All of the research directions described above can be seen as related strands of a simple goal: identifying human activities where the constraining factor is the availability of human intelligence and implementing technologies capable of amplifying it or replacing it when appropriate as physical machines amplify or replace human strength or dexterity. If these goals are met people will be able to concentrate on activities of ever greater depth and creativity and great productivity gains will materialise in those very information bound domains (the sciences, organisational management) from which productivity gains have traditionally come, hopefully triggering a virtuous circle of global proportions.

Resource Selection and Application Execution in a Grid: A Migration Experience from GT2 to GT4

A. Clematis[1], A. Corana[2], D. D'Agostino[1,3], V. Gianuzzi[3], and A. Merlo[2,3]

[1] IMATI-CNR, Via De Marini 6, 16149 Genova, Italy
[2] IEIIT-CNR, Via De Marini 6, 16149 Genova, Italy
[3] DISI, Universitá Genova, Via Dodecaneso 35, 16146 Genova, Italy

Abstract. The latest Globus Toolkit 4 (GT4) version, fully OGSA compliant and almost completely based on Grid Services, is expected to become the reference version of the Globus Toolkit. Therefore, the necessity to migrate from previous versions, especially the widespread GT2, to GT4 is becoming a relevant issue for many Grid systems.

We present a migration experience from GT2 to GT4 of a simple tool for resource discovery and selection, and execution of parallel applications in a computational Grid environment.

The proposed tool provides a simple and intuitive user interface, a structured view of Grid resources, a simple mechanism for resource selection, and support for application staging and execution.

We discuss some relevant differences between GT2 and GT4, particularly for the Monitoring and Discovering subsystem. GT4 turns out to be more powerful and flexible.

Just because we address the implementation of basic functionalities, the proposed discussion may be of general interest.

1 Introduction

Grid platforms are increasingly used to share computation resources, data, instrumentation and knowledge. Grid systems and virtual organizations are no longer used only in scientific advanced projects, but are now of interest for a wider potential user's community, including business, industry and medium/small research groups. Such users need powerful but simple tools to set-up, administrate and exploit their own Grid platforms, often obtained by middleware customization. In the last few years standards for Grid systems evolved, and particularly there was a convergence between Grid and web services; indeed the Open Grid Services Architecture (OGSA) specifies the requirements for a Grid architecture based on web services [4]. This approach is general and powerful since it is based on existing standards, that are extended only when needed. In particular, Grid Services are improved web services, with some generalizations, e.g. the introduction of the state [7].

The Globus Toolkit is the most used middleware for Grid, since its introduction in the late '90. In particular GT2 [3] has been the most widespread version and still represents the basic middleware in many Grid platforms. Also

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1132–1142, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the Globus Toolkit evolved following the OGSA specifications; however, the first service oriented GT version (GT3) experienced a diffusion lower than expected, also owing to the cost of migration of existing Grid applications.

The latest GT4 version (released in 2005) [8] is fully OGSA compliant and almost completely based on Grid Services and it is expected to became the reference version of the Globus Toolkit, also because the GT2 version is no longer supported. Therefore, the necessity to migrate from previous versions to GT4 is becoming a relevant issue for many Grid systems [5,9].

In the present work we are interested in computational Grids. In particular, we propose a tool that keeps in consideration the basic requirements for Grid resource monitoring and application deployment and provides a simple and intuitive user interface. The tool permits a structured view of Grid resources, the monitoring and selection of resources, and the staging and execution of parallel applications (both simple parameter sweep and message passing applications).

We present a migration experience of such tool from Globus Toolkit 2 to Globus Toolkit 4, pointing out the basic differences between the two versions of the structures involved in the Monitoring and Discovery Service (MDS).

2 A Simple Application Deployment Tool for a Grid

Since computational Grids comprise various machines, as a rule heterogeneous and belonging to different administrative domains, a simple tool able to classify nodes following their actual affinity degree, and to organize them into subclasses on the basis of suitable common criteria like cpu speed, RAM memory, network connection, and so on, is very useful.

The proposed tool consists of two parts: the first, for the Grid administrator, allows to set up a logical Grid structure (GSC, Grid Structure Creator); the second one (called GRSAE, Grid Resource Selection & Application Execution) permits the view of the structure created by GSC, the selection of a suitable set of resources and the staging and execution of the parallel application on the selected machines. The user interacts with the Grid through an ad-hoc GUI.

The tool has been designed with the following requirements: 1) it uses the mechanisms available in Globus Toolkit but it must be open to integration with other tools; 2) it must be modular; 3) it must be able to execute both simple parameter sweep as well message passing applications (eg. MPI applications).

2.1 A Structured View of Available Machines

In order to describe in a simple but effective way the complexity of the Grid, we choose to divide the computational resources potentially available in a Grid into three classes, namely the *single machine*, the *homogeneous cluster* and the *heterogeneous cluster*.

The single machine is the base unit, with a set of resources such as: node name, IP address, model and clock of the processor, number of available processors on the motherboard, total and free RAM, total and free disk space, type and release of O.S.

A homogeneous cluster is a set of machines having the same set of properties (i.e. a set of machines with the same hardware and software connected by a homogeneous network).

A heterogeneous cluster is a set of machines with different characteristics. It can be thought as a set of different machines that can be convenient to organize into a logical structure, for some kind of reason (e.g. all machines belong to the same department or to the same research group).

We suppose that a cluster (both homogeneous and heterogeneous) must have a distributed file system and is composed of nodes belonging to the same administrative domain. It needs three additional attributes: number of nodes, type of the distributed filesystem, and description of the network connection. In the current tool version, the homogenous cluster is seen as a unique computational resource, whereas in a heterogeneous cluster we can view each component node.

2.2 The Logical Grid Structure

The three models we have just described permit to create using GSC a logical Grid structure on which GRSAE works on. Such resulting topology has a tree structure [1] and is composed by a hierarchical disposition of the instances of classes we have presented.

The root of such tree structure is a single machine to which are logically connected by a path all the available machines. Each son of this root can be the root of a cluster (homogenous or heterogeneous) or a single machine. Each heterogeneous cluster is itself a tree with a root node and single machines as leaves.

The Grid middleware (GT in our case) is installed on each single machine, on the root of the homogeneous cluster, and on each node of heterogeneous clusters. We employ a test Grid composed of about ten machines (which can be grouped into heterogeneous clusters) plus a 16-nodes homogeneous cluster. All processors are Intel Pentium or Xeon.

2.3 Resource Selection, Node Allocation and Program Execution

The tool GRSAE helps the user in the execution of a parallel application. It is composed by three different modules, anyone with a well-defined role.

Connection Module. This module discovers and shows the entire structure of the Grid, the logical connection of the nodes and the state of the resources.

Selection Module. It interacts through a GUI with the user, and allows him to analyze the available resources and choose a subset of them (single machines or clusters) on which to run the parallel program. The user can filter resources on the basis of suitable criteria (e.g. processor type, frequency, RAM, disk space).

Execution Module. This module automatically creates the configuration files to launch the application on the selected machines of the Grid. The configuration files contain the executable path, the working directory for temporary files, the names of machines selected for the execution, the number of processes, the information for the staging of remote input and output files.

3 Deployment in Globus Toolkit 2.4

The tool, whose functionalities have been described in the previous section, has been originally deployed into Globus Toolkit 2.4 [2].

Such Grid middleware is composed by different independent subsystems that manage different aspects of the Grid. To set up our structured view of available resources, we are interested specifically at one of them, the MDS.

MDS is the information service that keeps track of all information about the Grid resources and their state. MDS has a hierarchical structure that consists of three main components. Every resource (e.g. CPU, RAM, storage) is controlled by a logical structure (GRIS, Grid Resource Information Service), that checks/measures the resource attributes through a suitable set of meters (IP, Information Provider). A set of resources (i.e. a set of GRIS) is then collected in another structure (GIIS, Grid Index Information Service): typically, there is a GIIS for every node of the Grid, collecting information about all resources of that machine. Various GIISs can be then organized hierarchically, creating groups of structures able to control many machines. A GIIS can be inquired about the set of resources it manages; information is supplied through a database using the standard LDAP (Lighweight Directory Access Protocol).

Deploying our structure on GT2 MDS is quite simple. In fact, we can map the root of the structure into a GIIS that manages a groups of other GIIS: one for each homogenous cluster, one for each heterogeneous cluster and one for each single machine, independent or contained in a heterogeneous cluster (Fig. 1).

The Grid structure is previously created by the Grid administrator using the tool GSC. Each cluster is registered in the root GIIS. Each machine is registered directly in the root GIIS, if it is a single machine, or in the GIIS of a cluster (if the machine belongs to a cluster).

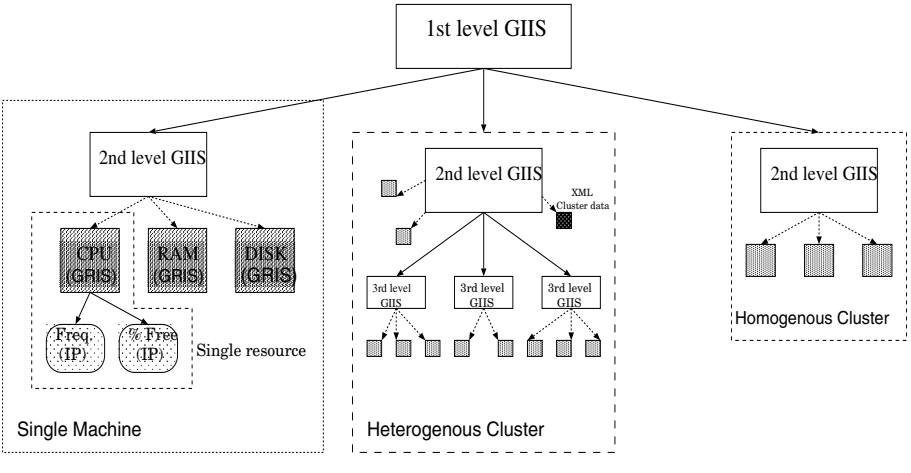


Fig. 1. The structure of the Grid under GT2; grey boxes denote GRIS

3.1 Modifying LDAP and MDS

In Globus Toolkit 2.4 data on resources of machines are kept on a distributed database (LDAP): a user that needs to know the state of a certain machine has to query the LDAP database to retrieve the attributes describing the resources of such machine. Unfortunately, in GT2 MDS and LDAP are able to keep track only of the resources on single machines: by default, there is no way to search and maintain collective information about a set of machines. For this reason, we modified the MDS and LDAP database in the following way:

- the new fields for the collective information needed for cluster are added to LDAP database;
- such information is kept in a XML file, one per cluster, on the cluster root; if characteristics of a cluster change we need only to modify the related XML file;
- in this way, a new IP for the cluster is created associated with the XML file: at every request of information about the cluster, it parses the XML file to find the relevant data that are stored in the new fields of LDAP database.

We denote by *'modified MDS'* the resulting MDS.

3.2 Interacting with the User

After the modification of MDS, the GRSAE tool is a Java application, implemented through Java CogKit, a Globus-oriented Java package, allowing modularity and easy interface with GT2. The modules of GT2 GRSAE behave as follows.

Connection Module. This module starts with enquiring the modified MDS to retrieve information about the structure; we query the root GIIS, we visit the tree and store all intermediate GIIS, until the leaves (single machines). Information about clusters is collected by a query of the cluster root GIIS and by reading the information stored in the corresponding XML file.

Selection Module. The structure and the resources are then shown to the user through the GUI (Fig. 2). From the whole set of resources found by the previous module, are returned the machines which fulfil the requisites given by the user. If the case, the user can further check the state of the resources, through a new query of the connection module.

Execution Module. After the choice has been made, GRSAE proceeds, through the Create RSL button, to create a Globus RSL executable file ($< file.rsl >$) with the parameters chosen by the user. Then the file is submitted to Globus Toolkit through the command '$globusrun < file.rsl >$', starting execution on the selected machines. Remote file transfer and management are performed by the module GASS (Global Access to Secondary Storage) of GT2.

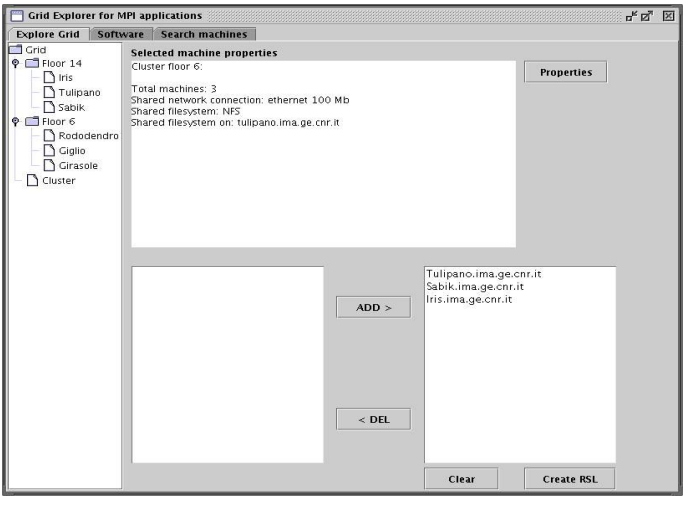


Fig. 2. An example of resource discovering and selection: on the left the whole tree, on the right the info about cluster Floor 6

4 Grid Information Service: Differences Between MDS2 and MDS4

Globus Toolkit evolution from the widely-adopted GT2 to the new-come GT4 passed through significant structural changes [10]. The new way of conceiving every resource on the Grid as an "improved" Web Service (GRID Service) and the need to access such resources in a common and standard way through different organizations required radical changes in the Globus architecture, specially in the Information Service [8]. We show in this section the main differences between MDS2 and MDS4, pointing out the advantages of the new implementation.

GRIS & GIIS vs WS Index Service. MDS2 manages information about one or more resources through a hierarchical system: every resource is controlled by a logical structure (GRIS) that uses a group of meters (IP) to measure different properties of the resource. Such properties are represented in LDAP-standard form. A set of GRISs can be organized into a GIIS that can be inquired about the set of resources it manages; moreover, different GIISs can be organized hierarchically.

In the MDS4 approach, every single resource (e.g. CPU, RAM, storage) is mapped into a Grid Service: in such a situation, accessing the state of a resource means querying an appropriate interface of the Grid Service that controls the state of the resource. In this case, the Grid Service replaces exactly the GRIS in MDS2 approach. Information is now inquired through standard Web Services queries and is returned in a XML document.

The GIIS structure has been replaced with Index Service (that is "de facto" another Grid Service) on which all the "resource" Grid Services can be registered on. By default, there is an Index Default Service on every host, that manages

all the resources of the host, but there is the possibility to create personalized Index Services.

Index Services can be organized in a hierarchical structure like GIIS (Fig. 3). The main difference between the two approaches is that MDS4 is more general and highly configurable. The Index System offers also a quicker way to configure homogeneous and heterogeneous clusters, allowing to create hierarchy of Indexes and to add information about the state of the cluster in a simpler way.

LDAP Database vs. XML Document. To fulfil OGSA requirements, GT4 replaced the LDAP database with a XML based description. XML is more general and doesn't depend on a particular database architecture. Moreover, it is platform-free and can be read by any proprietary system.

In XML approach information about resources is presented through a well-formed document based on standard schemas, creating a common way to describe the same kind of resource through different Grids and virtual organizations. Such schemas can easily be modified to permit the addition of new information about a resource: any organization that adopts ad-hoc schemas to describe its own resources, has only to publish and share them to allow Information Service from other organizations to retrieve information.

LDAP Architecture vs. Plugin Architecture. Another big difference is in the way the two systems practically collect information at the lowest level. In MDS2 information about a certain property of a resource is obtained through queries to a LDAP database, distributed among the GIIS. On the other hand, MDS4 doesn't support a particular system to retrieve such information but has a plug-in based architecture that allows the Grid administrator to use the preferred mechanism to collect information. Such approach stimulates the production of new information collectors.

LDAP Browser vs. WebMDS. In MDS2, information about a resources could be graphically shown to the user through one of available LDAP browsers, without any default. In order to achieve a higher standardization, GT4 adopted a single standard browser (WebMDS, based on Apache Tomcat) to access information in a XML document.

Pulling vs. Pushing. In MDS2 the only way to be aware of the state of a resource is to perform an explicit query to the LDAP database, and to keep up-to-date the state of a resource we have to repeatedly query the LDAP database, without any possibility to be informed automatically when a particular situation occurs. MDS4 breaks such limitation through a particular component, called Trigger Service, that allows to inform the user when a certain situation happens through a particular signal called Notification.

SAML vs. HTTPS. The security protocol that manages the authorizations and the certificates changed from SAML used in GT2 to HTTPS in GT4, because of the wider use of HTTPS on the Web, even if the way to authenticate and manage the security, based on proxies and certificates, remains quite the same in GT2 and GT4.

5 GSC and GRSAE Migration from GT2 to GT4

The differences between MDS2 and MDS4 we have shown before, make interesting the migration of GT2-based tools like GSC and GRSAE to GT4. It is clear that due to the significant architectural differences between MDS2 and MDS4, such operation could require a quite total rewriting of the application.

In particular, using GT4 we don't need to modify MDS to manage collective information about the cluster (FS Type, Connection Type, etc.), but we use an ad-hoc Grid Service on every node that is the root of a cluster.

The GT4 version of the tool is maintained in Java, adapting the GT2 one, and relying on the Java Core of the GT4 suite [11].

5.1 Organizing the Structure of the Grid

The creation of a Grid structure from a set of hosts and clusters can be performed on MDS4 through the Index Service. Every hosts with GT4 has a Default Index Service that keep track of the information about all the resources (as Grid Services) of the machine. Any of this Default Index Service can be placed in relation of hierarchical dependence with a second Default Index Service, simply modifying a XML configuration document (**hierarchy.xml**) on the machine that must have the lower level in the hierarchy.

For example, to make a machine A "father" of a machine B, we must only modify the hierarchy.xml file in B, inserting a tag labeled "upstream" with the address of the Default Index Service of A. After this modification has been done, the Default Index Service on machine A collects information about resources on B directly through Default Index Service of B.

In this way, the Grid Administrator can easily design a Grid structure (Fig. 3) using an ad hoc GSC tool for GT4. The hierarchy creation results simpler and more automatic than in GT2. Moreover, we can add in a simple way information about the Grid or the clusters (e.g. Total memory available on all the machines of a cluster) different from that offered by the Default Index Service of the single host. To obtain such result, GSC simply creates a new schema from the default ones, adding the new meters and mapping them to an appropriate collector of data.

5.2 Interacting with the User

With GT4 the three components of GRSAE perform the following tasks.

Connection Module. This module interacts only with the MDS4 components of GT4. It takes care of inquiry the Default Index Service of the host that is the root of the topology, in order to know the state of all the resources of the Grid. It receives a hierarchically-organized XML and parses it, taking out information about the Grid structure, and the state of every Grid Service on every nodes. Then, it starts a communication to the Selection module, passing to it the whole information organized in appropriate Java data structures.

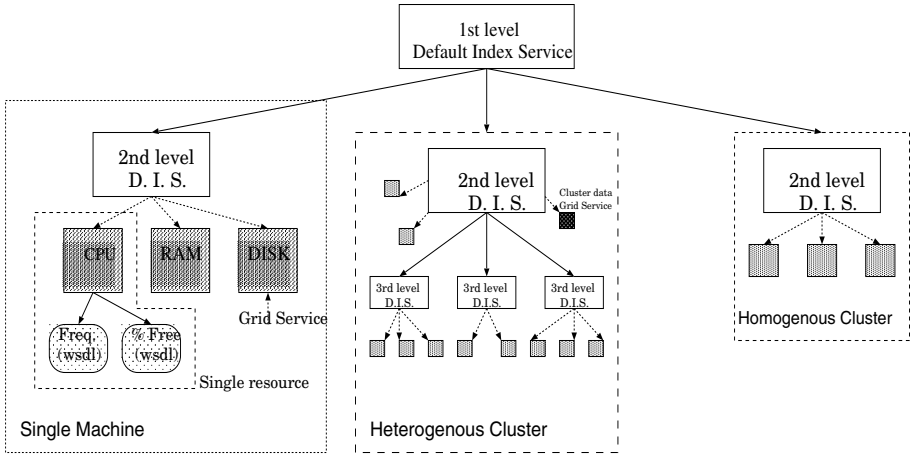


Fig. 3. The structure of the Grid under GT4; grey boxes denote Web Services

Selection Module. This module is the central one because interacts directly with the user, through a GUI quite similar to the one used with GT2. It firstly shows the state of the Grid and its resources (Grid Services of single host and clusters) to the user (from the data structures it receives from the Connection module). Then it gives the user the ability to choose the resources and the options to run correctly his parallel program.

An important improvement of GT4 is the new Trigger Service. The user can chose a set of constraints on one or more resources, and attend that such conditions are fulfilled before the launch of the parallel program. Such constraints are submitted through a particular form to the Trigger Service of the root node (through the Connection module). The connection module comprises an active daemon, able to interface with the Grid root node, that waits for the notifications from the Trigger Service of the root node and quickly transmits such notifications to the selection module that informs the user, by appropriate pop-up windows, that the constraints submitted has been satisfied.

When this happens, a new inquiry to the Default Index Service of the root node is performed and the state of the Grid is updated. Such query can also be performed manually at every moment by the user.

The user chooses the nodes that best fit his requirements and requests to launch the parallel job on that nodes: the Selection module passes such information to the Execution module.

Execution Module. It manages the creation of executables and starts their execution on the Grid, interacting directly only with the GRAM component of GT4. The Execution module receives data from the Selection module, and, for every host involved in the execution of the parallel program, creates an ad-hoc XML file with the running configuration. Such XML files follow an ad-hoc schema that allows to run the program on every GT4 GRAM Service. XML files are created by a sub-module that parses the structure of the schema and creates

the XMLs. Then, every XML file is staged to the corresponding host and the program is launched through the GRAM services of the selected hosts.

During and after the execution, all the output and error messages are reported to the Execution module and then to the user attention through the GUI. Such control of the status of the running job and the staging of all files (XMLs, inputs, outputs) has been obtained taking advantage of the new transfer protocol of GT4 RFT (Remote File Transfer), built on the old GridFTP one. After the execution is correctly finished, a detailed report is sent to the user, together with performance information (e.g. execution time on every node and other statistics that allows the user to understand potential bottlenecks in the set of nodes).

6 Conclusions

We presented a migration experience from Globus Toolkit 2 to Globus Toolkit 4 of a simple tool for the discovery and selection of computational resources and execution of parallel applications in a Grid environment.

We discuss the main differences between GT2 and GT4, particularly for the Monitoring and Discovering subsystem, more directly involved in our porting experience. We try to explain also the impact of some "philosophical" differences between the two Globus versions.

GT4 turns out to be more powerful and flexible, and it makes easier the design and implementation of our tool. In particular, GT4 facilitates the creation of the Grid structure, provides a more direct support to the view and selection of Grid resources, and makes easier the management of collective information about clusters. Moreover, the Trigger Service allows to enhance the capabilities of the Selection module.

The proposed work is a first step towards the implementation of a environment for the efficient execution of parallel structured applications on the computational Grid. There are worldwide several efforts in this direction: among the most important we mention the VGrADS Project [12], which is the prosecution of the earlier GrADS Project [6].

Acknowledgement

This work was supported in part by the Project PRAI-FESR Regione Liguria 2006, Action 28 'Software tools for the development of distributed high performance computing platforms for industrial research'.

References

1. Y. Chen, Y. Li, Z. Gong, Q. Zhu, A framework of a tree-based Grid Information Service, Proc. IEEE Int. Conf. on Services Computing (SCC'05), IEEE Computer Society, 2005.
2. F. Conti, Tools for execution and simulation of structured parallel programs in a GRID environment, Master Thesis, DISI, University of Genova (I), 2005.

3. L. Ferreira et al., Introduction to Grid Computing with Globus, Ibm Redbooks, 2003.
4. I. Foster, C. Kesselman, J.M. Nick, S. Tuecke, Grid services for distributed system integration, Computer, 35, 6, 2002.
5. I. Foster, C. Kesselman, The GRID 2: Blueprint for a new computer infrastructure, Elsevier, 2005.
6. GrADS: http://www.hipersoft.rice.edu/grads/.
7. A. Grimshaw, Grid services extend Web services, SOAWebServices Journal, Jul. 24, 2003 (http://webservices.sys-con.com/read/39829.htm).
8. GT4: http://www.globus.org/toolkit/docs/4.0/.
9. B. Jacob, M. Brown, K. Fukui, N. Trivedi, Introduction to Grid Computing, IBM Redbooks, 2005.
10. Migrating from GT2 to GT 4.1.0: http://www.globus.org/toolkit/docs/development/4.1.0/toolkit-migrating-gt2.html.
11. B. Sotomayor, The Globus Toolkit 4 Programmer's Tutorial, 2005 (http://gdp.2 globus.org/gt4-tutorial/).
12. VGrADS: http://vgrads.rice.edu/.

A Comparative Analysis Between EGEE and Grid *W*ay Workload Management Systems*

J.L. Vázquez-Poletti, E. Huedo, R.S. Montero, and I.M. Llorente

Departamento de Arquitectura de Computadores y Automática. Facultad de Informática, Universidad Complutense de Madrid. 28040 Madrid, Spain

Abstract. Metascheduling is a key functionality of the grid middleware in order to achieve a reasonable degree of performance and reliability, given the changing conditions of the computing environment. In this contribution a comparative analysis between two major grid scheduling philosophies is shown: a semi-centralized approach, represented by the EGEE Workload Management System, and a fully distributed approach, represented by the Grid *W*ay Metascheduler. This comparative is both theoretical, through a functionality checklist, and experimental, through the execution of a fusion plasma application on the EGEE infrastructure.

1 Introduction

The growing computational needs of nowadays projects have permitted the evolution to a new paradigm called Grid Computing. Among the intervening elements of a computional grid, the Metascheduler is gathering most attention as a way to meet these challenging needs. The term Metascheduler can be defined as a grid middleware that discovers, evaluates and allocates resources for grid jobs by coordinating activities between multiple heterogeneous schedulers that operate at local or cluster level [1].

Several implementations of the Metascheduler can be found, such as CSF, Silver, Nimrod/G, Condor-G, GHS, GrADS, MARS, AppLeS and Gridbus. From these, we would like to remark the following: CSF supports advance reservation booking and offers round-robin and reservation based scheduling algorithms. Scheduling characteristics provided by Nimrod/G strive for the equilibrium between resource providers and resources consumers via auctioning mechanisms [2]. Condor-G is not conceived for supporting scheduling policies but in the other hand, it supplies mechanisms, such as *ClassAd* and *DAGMan*, that may be useful for a metascheduler standing above [3]. GrADS and AppLeS support scheduling mechanisms that take into consideration both application and system level environments [4,5]. These solutions provide complementary functionality,

* This research was supported by Consejería de Educación de la Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through BIOGRIDNET Research Program S-0505/TIC/000101, and by Ministerio de Educación y Ciencia, through the research grant TIC2003-01321. The authors participate in the EGEE project, funded by the European Union.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1143–1151, 2006.
© Springer-Verlag Berlin Heidelberg 2006

which focuses on specific application domains and grid middlewares, contributing significant improvement in the field.

The aim of the Grid*W*ay metascheduler [6] is to provide Globus user community with a grid scheduling functionality similar to that found in local DRM (Distributed Resource Management) systems. In a precedent contribution, we showed how a joint testbed of EGEE (LCG-2 middleware) and non-EGEE resources (Globus middleware) can be harnessed by Grid*W*ay with good results [7]. The evaluation of the resulting infrastructure showed that reasonable levels of performance and reliability were achieved. The objective of this contribution is to compare two metascheduling philosophies by means of two representative implementations: The EGEE WMS (Workload Management System) and Grid*W*ay Metascheduler. In both cases, the experimental results are obtained on EGEE computing resources.

The structure of this paper is as follows. In Section 2, a short architecture overview of the EGEE WMS and the Grid*W*ay Metascheduler can be found, along with a comparative analysis of the functionality provided by both solutions is shown. Then, Section 3 evaluates the performance obtained by the two alternatives in the execution of fusion plasma application on the EGEE infrastructure. Finally in Section 4, the paper ends up with some conclusions.

2 Grid Scheduling Infrastructures

The EGEE WMS and Grid*W*ay are representative metascheduling technologies of different strategies to deploy a grid scheduling infrastructure. Grid*W*ay follows the *loosely-coupled* Grid principle [8], mainly characterized by: autonomy of the multiple administration domains, heterogeneity, scalability and dynamism. A Grid*W*ay instance is installed in each organization involved in the partner grid to provide scheduling functionality for intra-organization users (site-level scheduling). On the other hand, the EGEE WMS provides a higher centralized strategy as one or more scheduling instances can be installed in the grid infrastructure, each providing scheduling functionality for a number of VOs (VO-level scheduling).

The EGEE WMS is the result of previous projects (Datagrid, CrossGrid). The present version of the EGEE WMS is based on the LCG-2 middleware and it is migrating to a new one called gLite which inherits many of the former elements and functionalities. The EGEE WMS architecture is highly centralized and each functionality is provided by almost a different machine, as it is conceived as a network service. The EGEE WMS components are the following: The User Interface (UI), which is from where the user sends the jobs; the Resource Broker (RB), which is based in Condor-G and uses most of its functionality; the Computing Element (CE) and the Worker Nodes (WN), which are the cluster frontend and the cluster nodes respectively, as it is established in the fixed configuration dictated by the EGEE WMS architecture; the Storage Element (SE), which is used for job files storage; the Logging and Bookkeeping service (LB), which registers job events [9].

For this contribution's purpose, we have studied the RB, where the user submits the job and its matching is performed. Here, the RB can adopt an eager or lazy policy for scheduling the jobs. While with the eager policy the job will likely end up in a queue, with lazy scheduling the job is held until a resource becomes available. Then the job is sent to the chosen CE and, from there, executed in a corresponding WN. In any case, there are several alternative brokering services to submit the job.

GridWay works on top of Globus services, performing job execution management and resource brokering, allowing unattended, reliable, and efficient execution of jobs, array jobs, or complex jobs on heterogeneous, dynamic and *loosely-coupled* Grids formed by Globus resources. GridWay's modular architecture is conformed by the GridWay Daemon (GWD) and different Middleware Access Drivers (MADs) to access different Grid Services (information, execution and transfer), all of them in just one host, as GridWay is conceived as a client tool. GridWay performs all the job scheduling (using a lazy approach) and submission steps transparently to the end user adopting job execution to changing Grid conditions by providing fault recovery mechanisms, dynamic scheduling, migration on-request and opportunistic migration. GridWay allows the deployment of organization-level meta-schedulers that provide support for multiple intra-organization users in each scheduling instance. There is one scheduling instance for each organization and all instances compete with each other for the available resources.

2.1 Scheduling Capabilities

Both EGEE WMS and GridWay treat jobs in a FIFO way and support dynamic scheduling, providing a way to filter and evaluate resources based on dynamic attributes, by means of requirement and rank expressions. In the EGEE WMS, the names of these attributes are the same as retrieved from the information service. Nevertheless, in GridWay, these expressions are based on common resource attributes, independent from the information service, providing another way of decoupling. While the EGEE WMS RB accesses only the BDII servers and only processes the GLUE Schema, GridWay's different information MADs allow to access the most common information services. Considering execution and transfer functionalities, both EGEE WMS and GridWay support Globus Pre-WS services, but only GridWay allows access to Globus WS services [10].

GridWay supports opportunistic migration. That is, each scheduling cycle evaluates the benefits of migrating submitted jobs to new available resources (recently added or freed) by comparing rank values. In the EGEE WMS, this functionality is not supported and the ranking only affects submission.

Considering performance slowdown detection, GridWay takes count of the suspension time in remote batch systems and requests a migration when it exceeds a given threshold. Moreover, jobs are submitted together with a lightweight self monitoring system. The job will migrate when it doesn't receive as much CPU as the user expected. None of the performance slowdown detection mechanisms given by GridWay are implemented in the EGEE WMS. The

monitoring in the EGEE WMS architecture is provided by the LB, which records only basic job states and mixes them with events originated in other components.

With GridWay, an application can take decisions about resource selection as its execution evolves by modifying its requirement and rank expressions and requesting a migration. In the EGEE WMS RB instead, these expressions are only set at the beginning.

The EGEE WMS supports checkpointing by providing an API to allow applications to be instrumented to save the state of the process (represented by a list of variable and value pairs) at any moment during the execution of a job. Also, it provides the mechanisms to restart the computation from checkpointed data (previously saved state). If a job fails, the WMS automatically reschedules the job and resubmits it to another compatible resource. There, the last state is automatically retrieved and the computation is restarted. The user can also retrieve the saved state for a later manual resubmission, where the user can specify if the job must start from this retrieved checkpoint data. With GridWay, user-level checkpointing or architecture independent restart files managed by the programmer must be implemented. Migration is implemented by restarting the job on the new candidate host. If the checkpointing files are not provided, the job should be restarted from the beginning. These checkpoints are periodically retrieved to the client machine or a checkpoint server.

In the EGEE WMS, the PBS Event Logging (APEL) is employed for distributed accounting [11]. GridWay gives the user local accounting functionalities, standing on the Berkeley Database.

Dependency in job submission is supported in the EGEE WMS RB by the Condor's DAGMan tool [11]. A DAGMan process is locally spawned for each Direct Acyclic Graph (DAG) submitted to Condor-G. GridWay also provides support for job dependencies.

2.2 Fault Detection and Recovery Capabilities

The EGEE WMS RB incorporates error detection mechanisms provided by Condor-G [12]. On the other hand, GridWay detects job cancellation (when the job exit code is not specified), remote system crash and network disconnection (both when the polling of the job fails). In all of these cases, GridWay requests a migration for the job [13].

Also the system running the scheduler could fail. GridWay persistently saves its state in order to recover or restart the jobs when the system is restarted. The EGEE WMS RB relies on Condor-G, which stores persistently all relevant state for each submitted job in the scheduler's job queue [3].

2.3 User Interface Functionality

Both GridWay and the EGEE WMS RB allow single jobs. The EGEE WMS RB can handle jobs with dependencies (DAGMan functionality) and interactive jobs. On the other hand, GridWay allows array jobs, jobs with dependencies and complex jobs. For providing complex job support, GridWay gives the user

a functionality to synchronize jobs. In the case of the EGEE WMS, this must be implemented by a periodical polling (active wait).

Focusing in the command line interface, both Grid*W*ay and the EGEE WMS RB give the user full control of his jobs. Anyway, Grid*W*ay incorporates commands which allow the user to migrate and synchronize jobs, functionalities not provided by the EGEE WMS.

Also, Grid*W*ay offers C and Java implementations of the DRMAA Application Programming Interface, which is a Global Grid Forum (GGF) standard [14]. The EDG WMS API given by the EGEE WMS[1] is not standard.

3 Experimental Conditions and Results

The grid infrastructure used for the experiments is the corresponding to the Test Virtual Organization at the Southwest Federation (SWETEST VO) of the EGEE project (Table 1). All Spanish sites are connected by RedIRIS, the Spanish Research and Academic Network, whose interconnection links of the different nodes are shown in Figure 1.

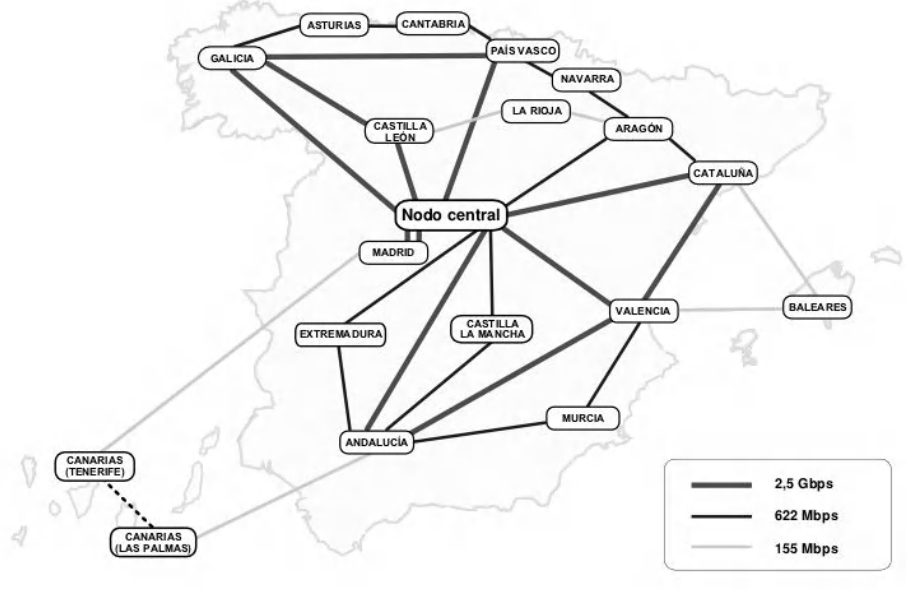


Fig. 1. Topology and bandwidths of RedIRIS-2

The target application, called *Truba*, performs the tracing of a single ray of a microwave beam launched inside a fusion reactor [15]. Each experiment involves the execution of 50 instances of the *Truba* application. The experiments were performed with a development version of *Truba*, whose average execution time

[1] http://www.to.infn.it/grid/workload_management/apiDoc/edg-wms-api-index.html

on a Pentium 4 3.20 GHz is 9 minutes. *Truba*'s executable file size is 1.8 MB, input file size is 70 KB, and output file size is about 549 KB.

For the EGEE WMS experiments, we have developed a framework using the lcg2.1.69 User Interface C++ API, which provides support to submit, monitor and control each single ray tracing application to the grid. This framework works in the following way: First of all, a launcher script generates the JDL files needed. Then, the framework launches all the single ray tracing jobs simultaneously, periodically querying each job's state. And finally, it retrieves the job's output. The scheduling decisions are of course delegated to the EGEE WMS.

Grid*Way* only relies on Globus services, so it could be used in any Grid infrastructure based on the Globus Toolkit, both PreWS and WS [10]. In the case of the EGEE WMS (LCG-2), Globus behaviour has been slightly modified, but it does not loose its main protocols and interfaces, so Grid*Way* can be used in a standard way to access LCG-2 resources [7].

Table 1. EGEE grid resources employed during the experiment

Site	Processor	Speed	Nodes	DRMS
CESGA	Intel Pentium III	1.1 GHz	46	PBS
IFAE	Intel Pentium 4	2.8 GHz	11	PBS
IFIC	AMD Athlon	1.2 GHz	127	PBS
INTA-CAB	Intel Pentium 4	2.8 GHz	4	PBS
LIP	Intel Xeon	2.8 GHz	25	PBS
PIC	Intel Pentium 4	2.8 GHz	172	PBS
USC	Intel Pentium III	1.1 GHz	100	PBS

In both cases, the jobs were submitted from Universidad Complutense de Madrid. The RB employed for the experiments with the EGEE WMS was located at the IFIC site and used an eager scheduling policy.

3.1 Experimental Results

Table 2 shows a summary of the performance exhibited by the two scheduling systems in the execution of the fusion application. As can be seen, Grid*Way* presents a higher transfer time, because of the reverse-server transferring model used for file staging [6] (which has been replaced in version 4.7 for solving this issue). Moreover, the standard deviation of raw performance metrics can be interpreted as an indicator of the heterogeneity in the grid resources and interconnection links [7]. Finally, the lower overhead induced by Grid*Way* shows the benefits of its lighter approach and the functionality for performance slowdown detection.

The EGEE WMS spent 195 minutes (3.25 hours) to execute the 50 jobs, giving a productivity equal to 15.38 jobs/hour. Grid*Way* spent 120 minutes (2 hours) to execute the same workload, giving a productivity equal to 25 jobs/hour. We can conclude that Grid*Way* takes better advantage of the available resources due to its superior scheduling capabilities on dynamic resources. In fact, during the experiments with the EGEE WMS, several problems described before were

Table 2. Performance metrics for both platforms, times are in minutes and productivity is in jobs/hour

	Execution/Job		Transfer/Job		Turnaround	Productivity	Overhead/Job
Framework	Mean	Dev.	Mean	Dev.			
EGEE WMS	30.33	11.38	0.42	0.06	195	15.38	1.82
Grid*W*ay	36.80	16.23	0.87	0.51	120	25.00	0.52

evidenced. The EGEE WMS RB does not provide support for opportunistic migration and slowdown detection, and jobs are assigned to busy resources.

Additionally, the achieved level of parallelism [16] can be obtained by using the following expression:

$$U = \frac{T_{exe}}{T}, \tag{1}$$

being T_{exe} the sum of job execution times and T the turnaround time. The level of parallelism achieved by Grid*W*ay was higher than the level achieved by the EGEE WMS (14.91 and 6.89 respectively).

Not all jobs ended successfully at the first try. In the case of the EGEE WMS, 31 jobs were affected and they had to be resubmitted. However, with Grid*W*ay, only 1 job failed, but there were 21 migrations mostly due to suspension timeouts (too much delay in a queue), and better resource discovery (too much time allocated to a resource when better resources are waiting to be used).

A methodology to analyze the performance of computational Grids in the execution of high throughput computing applications has been proposed in [17]. This performance model enables the comparison of different platforms in terms of the following parameters: asymptotic performance (r_∞), which is the maximum rate of performance in tasks executed per second, and half-performance length ($n_{1/2}$), which is the number of tasks required to obtain half of the asymptotic performance. A first order characterization of a grid by means of these parameters is:

$$n(t) = r_\infty t - n_{1/2}. \tag{2}$$

Then, we can define the performance of the system, jobs completed per second, with a finite number of tasks with:

$$r(n) = n(t)/t = \frac{r_\infty}{1 + n_{1/2}/n}, \tag{3}$$

where n is the number of jobs. The parameters of the model, r_∞ and $n_{1/2}$, are obtained by linear fitting to the experimental results obtained in the execution of the applications.

Figure 2 and Figure 3 show the experimental performance obtained with the two workload management systems, along with that predicted by Eq. (2) and Eq. (3). With the EGEE WMS, r_∞ was 0.0051 jobs/second (18.19 jobs/hour) and $n_{1/2}$ was 8.33. With Grid*W*ay, r_∞ was 0.0079 jobs/second (28.26 jobs/hour) and $n_{1/2}$ was 1.92. From the different values of $n_{1/2}$, we can deduce that Grid*W*ay needs less jobs to obtain half of the asymptotic performance due to an earlier job allocation in the resources.

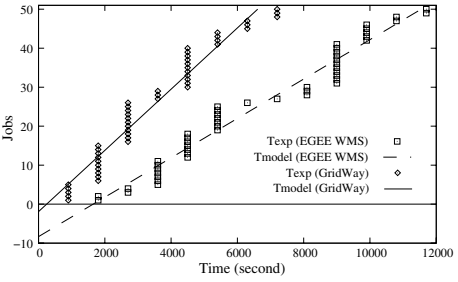


Fig. 2. Measurements of r_∞ and $n_{1/2}$ parameters for both platforms

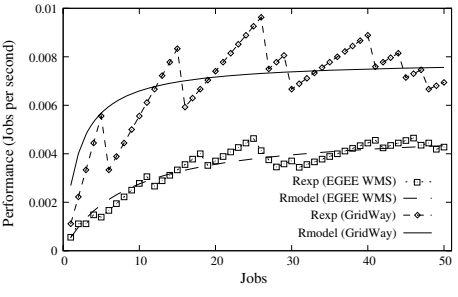


Fig. 3. Experimental and predicted performance

4 Conclusions

We have demonstrated that Grid*W*ay achieves lower overhead and higher productivity than the EGEE WMS. Grid*W*ay reduces the number of job submission stages and provides mechanisms, not given by the EGEE WMS RB, such as opportunistic migration and performance slowdown detection that considerably improves the usage of the resources. Nevertheless, EGEE WMS provides other components that weren't considered in this article, such as data management.

Acknowledgments

We would like to thank all the institutions involved in the EGEE project, in particular those who collaborated in the experiments.

References

1. Yu, J., Buyya, R.: A Taxonomy of Workflow Management Systems for Grid Computing. Journal of Grid Computing **3** (2005) 171–200
2. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. Fourth International Conference on High-Performance Computing in the Asia-Pacific Region **1** (2000) 283

3. Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01), IEEE Computer Society (2001) 55
4. Dail, H., Sievert, O., Berman, F., Casanova, H., YarKhan, A., Vadhiyar, S., Dongarra, J., Liu, C., Yang, L., Angulo, D., Foster, I.: Scheduling in the Grid Application Development Software Project. In: Grid resource management: state of the art and future trends. Kluwer Academic Publishers (2004) 73–98
5. Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D.: Adaptive Computing on the Grid Using AppLeS. IEEE Transactions on Parallel and Distributed Systems **14** (2003) 369–382
6. Huedo, E., Montero, R.S., Llorente, I.M.: A Framework for Adaptive Execution on Grids. Intl. J. Software – Practice and Experience (SPE) **34** (2004) 631–651
7. Vázquez-Poletti, J., Montero, R.S., Llorente, I.M.: Coordinated Harnessing of the IRISGrid and EGEE Testbeds with GridWay. Journal of Parallel and Distributed Computing **66** (2006) 763–771
8. Llorente, I.M., Montero, R.S., Huedo, E.: A Loosely Coupled Vision for Computational Grids. IEEE Distributed Systems Online **6** (2005)
9. Campana, S., Litmaath, M., Sciaba, A.: LCG-2 Middleware Overview. Available at https://edms.cern.ch/document/498079/0.1 (2004)
10. Huedo, E., Montero, R.S., Llorente, I.M.: Coordinated Use of Globus Pre-WS and WS Resource Management Services with GridWay. In: Proc. 2nd Workshop on Grid Computing and its Application to Data Analysis (GADA'05) on the Move Federated Conferences. Volume 3762 of Lecture Notes in Computer Science. (2005) 234–243
11. Avellino, G., Beco, S., Cantalupo, B., et al: The DataGrid Workload Management System: Challenges and Results. Journal of Grid Computing **2** (2004) 353–367
12. Morajko, A., Fernandez, E., Fernandez, A., Heymann, E., Senar, M.A.: Workflow Management in the CrossGrid Project. In: Proc. European Grid Conference (EGC2005). Volume 3470 of Lecture Notes in Computer Science. (2005) 424–433
13. Huedo, E., Montero, R.S., Llorente, I.M.: Evaluating the Reliability of Computational Grids from the End User's Point of View. Journal of Systems Architecture, (2006) (in press).
14. Herrera, J., Montero, R., Huedo, E., Llorente, I.: DRMAA Implementation within the GridWay Framework. In: Workshop on Grid Application Programming Interfaces, 12th Global Grid Forum (GGF12). (2004)
15. Castejon, F., Tereshcenko, M.A., et al.: Electron Bernstein Wave Heating Calculations for TJ-II Plasmas. American Nuclear Society **46** (2004) 327–334
16. Huedo, E., Montero, R.S., Llorente, I.M.: An Evaluation Methodology for Computational Grids. In: Proc. 2005 International Conference on High Performance Computing and Communications. Volume 3726 of Lecture Notes in Computer Science. (2005) 499–504
17. Montero, R.S., Huedo, E., Llorente, I.M.: Benchmarking of High Throughput Computing Applications on Grids. Parallel Computing (2006) (in press).

Grid and HPC Dynamic Load Balancing with Lattice Boltzmann Models*

Fabio Farina[1,2], Gianpiero Cattaneo[1,2], and Alberto Dennunzio[1]

[1] Università degli Studi di Milano–Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione,
Via Bicocca degli Arcimboldi 8, 20126 Milano (Italy)
{farina, cattang, dennunzio}@disco.unimib.it
[2] INFN Milano–Bicocca, Piazza della Scienza 3, 20126 Milano (Italy)

Abstract. Both in High Performance Computing and in Grid computing dynamic load balancing is becoming one of the most important features. In this paper, we present a novel load balancing model based on Lattice Boltzmann Cellular Automata. Using numerical simulations our model is compared to diffusion algorithms adopted on HPC load balancing and to agent-based balancing strategies on Grid systems. We show how the proposed model generally gives better performances for both the considered scenarios.

Keywords: Lattice Boltzmann Models, Dynamic Load Balancing, diffusion algorithm.

1 Introduction

Dynamic load balancing is one of the most challenging features for the next generation of both Grid and High-performance computing (HPC) systems. The main differences in the formulation for the dynamic load balancing (DLB in the following) problem between these scenarios are related to the capabilities associated to each processing element and its network topology. In High-performance systems the computing elements are considered homogeneous and are connected according to highly regular topologies, in general as meshes, fat-trees or hypercubes [2]. On the contrary Grid infrastructures [9] assume that all the computing elements participating in a Virtual Organization can be very different both in terms of network connectivity and in performance capability.

Let us define an abstract distributed formulation for the DLB Problem that is suited for both the Grid and the HPC scenarios. We are given an arbitrary, undirected, connected graph $G = (V, E)$ describing a general distributed processor system. Two real numbers are associated to any processor $p \in V$: $\rho(p, t) \in \mathbb{R}^+$, the current work load at time t for p, and $c(p, t) \in \mathbb{R}^+$ representing the maximal sustainable work load for the i-th node at time t. Obviously for HPC systems

* This work has been supported by M.I.U.R. COFIN project "Formal Languages and Automata: Theory and Applications".

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1152–1162, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the capacity c is assumed as a constant, $\forall p \in V, \forall t \in \mathbb{N}, c(p,t) = \overline{c} \in \mathbb{R}^+$. A node is assumed to be able to communicate bidirectionally with any other node connected to it. The connections for every node are stated in the set $E \subseteq V \times V$. Communication between nonadjacent nodes is not allowed.

The goal is to determine a schedule to move an amount of work load across edges so that finally, the weight on each node is the closest as possible to the node capacity. The performance of a balancing algorithm can be measured in terms of number of iterations it requires to reach a stable state and in terms of the amount of load moved over the edge of the graph. The abstract DLB formulation holds when we associate a node with a processor, an edge with a communication link of unbounded capacity between two processors, and the weight u_i on each node can be divided into a very large amount of independent tasks.

In this paper we present a brand new approach for the DLB problem based on a Cellular Automata model known as Lattice Boltzmann [4]. This local model was originally created to simulate flow phenomena and to solve elliptic and parabolic PDEs [16]. Today Lattice Boltzmann Models are widely used to simulate complex systems such as multiphase turbolent flows and percolation [19].

The rest of the paper is organized as follows. Section 2 presents several commonly used approaches for the solution of the DLB problem. In Section 3 the Lattice Boltzmann model to solve the DLB problem is presented, while in Section 4 some numerical simulations of the new method are reported. In this section we will also show how the model operates efficiently both on Grid and on homogeneous distributed processors with respect to some other efficient DLB solvers. Finally in Section 5 conclusive notes are reported.

2 Related Work

Many different approaches have been used in the recent years to solve efficiently the DLB problem. These techniques take inspiration from numerical approximations and distributed Artificial Intelligence. In this section several successful DLB solvers are described, showing the highlights and drawbacks of each methodology.

2.1 Diffusion Equation Models

Diffusion algorithms assume that a node of the graph is exclusively able to send and receive messages to/from all its neighbors simultaneously. Most of the existing iterative DLB algorithms [7, 10, 17] involve three steps:

- **Diffusion Iteration:** flow balance calculation towards the equilibrium of the system.The diffusion iteration is a preprocessing phase which determines the actual load balance logic and rules the performance of the DLB solver.
- **Flow Calculation:** work load schedule preparation to migrate the amount of exceeding load between neighboring processors.
- **Task Migration:** deciding which particular tasks are to be migrated. This phase is particularly network intensive and is strongly application dependent.

Naturally, these models focus their efforts on the search of an effective way to reach a stable status during the Diffusion Iteration phase.

The original algorithm described in the Cybenko work [6], known as the Diffusion method, assumes the workload vector $\boldsymbol{p}$ over all the nodes $p \in V$ evolves according to the model

$$\boldsymbol{\rho}(t+1) = M\boldsymbol{\rho}(t)$$

where $\boldsymbol{\rho}(0)$ is the initial configuration and M is the diffusion equation governing the iterative process. The optimal matrix M is defined as $M = I - \frac{2}{\lambda_2+\lambda_n}L$ where I is the identity matrix, L is the Laplacian Matrix for the graph G and λ_2, λ_n are the larger and the smaller subdominant eigenvalues of L. Let us remind that $L = \mathrm{diag}(d_{pp}) - A$ where d_{pp} is the degree of node p, and A denotes the adjacency matrix of G. The Flow Calculation for the Diffusion method is performed solving the linear system $L\boldsymbol{d}(t) = \boldsymbol{\rho}(t) - \boldsymbol{c}(t)$, where $\boldsymbol{c}$ denotes the $c(p,t)$ values for each processor $p \in V$ and $\boldsymbol{d}(t)$ defines the schedule for the Task Migration [10]. It can be proved that the linear system solution can be performed online using an iterative schema similar to the one used for $\boldsymbol{\rho}(t)$, in particular $\boldsymbol{d}(t+1) = \boldsymbol{d}(t) + \frac{2}{\lambda_2+\lambda_n}\boldsymbol{\rho}(t)$.

Even in the case of the optimal matrix M, anyhow, the original Diffusion method lacks in performance because of its very slow convergence to the equilibrium.

A huge convergence acceleration is given by the Semi-iterative schema proposed in [11], which is somehow inspired by θ-schemas commonly used in numerical function extrapolations. In this algorithm the Diffusion Iteration is ruled by:

$$\boldsymbol{\rho}(t+1) = \sigma_{t+1}\left(\boldsymbol{\rho}(t) - \frac{2L\boldsymbol{\rho}(t)}{\lambda_2+\lambda_n}\right) + (1-\sigma_{t+1})\boldsymbol{\rho}(t-1) \tag{1}$$

$$\text{with} \quad \sigma_1 = 1, \quad \sigma_2 = \frac{(\lambda_2+\lambda_n)^2}{2\lambda_2\lambda_n}, \quad \text{and} \quad \sigma_{t+1} = \left(1 - \frac{(\lambda_2-\lambda_n)^2}{(\lambda_2+\lambda_n)^2}\cdot\frac{\sigma_t}{4}\right)^{-1} \tag{2}$$

The Flow Calculation step for the Semi-iterative schema requires the evaluation of the following flow potential: $\boldsymbol{d}(t+1) = \sigma_{t+1}\left(\boldsymbol{d}(t) + \frac{2}{\lambda_2+\lambda_n}\boldsymbol{\rho}(t)\right) + (1-\sigma_{t+1})\boldsymbol{\rho}(t-1)$. The use of semi-iterative techniques improves the rate of convergence of an order of magnitude with respect to the classical Diffusion method.

The main lack of Diffusion Models is that they are limited to the HPC systems only, where each node has the same characteristics and the same performances.

2.2 Swarm Intelligences

The Distributed Artificial Intelligence approach for DLB on Grid has been originally proposed in [12], thought they can be easily and effectively adapted to HPC context. The main idea behind this models derives from the artificial ants proposed to approximate efficiently NP-Complete problems solutions [15].

The swarm intelligence for DLB adopts a greedy schema divided in three phases:

- **SearchMax:** a migrating ant locates the most overloaded node;
- **SearchMin:** another other ant locates the less efficiently used node;
- **Transfer:** an ant migrates computational load from the most overloaded node to the most underloaded one.

In HPC or Grid environments a coalition of multiple mobile ants is considered. Each ant takes care of a computational workload granule. Ants have to exploit nodes and federate themselves into small teams with a fixed maximum size. Teams are useful both to save time during the SearchMax/SearchMin phases and to improve the transfer bandwidth. An ant can join or leave teams according the trails left behind by the other ants in the team. Such trails inform the newcomer about the load of the node currently hosting the team. Once the ants are clustered onto the most overloaded and on the most underloaded sites, then the transfer phase can be performed.

The main advantage of the outlined load balancing mechanism is that it requires no global knowledge of the node topology w.r.t. the Diffusion algorithms. Moreover, the direct migration among the nodes does not require a migration schedule calculation. The main drawbacks of swarm intelligence DLB solvers is that the ants may introduce further work load for the system. Such overload could imply a significant overhead for a very high number of agents. The convergence performance of the method is also very sensitive to the work load migration capabilities of ants: a small migration capacity would bring to a large number of ants overcrowding the system. Instead, a too large transfer capacity would imply a very slow convergence rate, with a small number of nodes swapping each other loads repeatedly [20].

3 The Model

Lattice Boltzmann Models (LBM) have been widely used to simulate multiphase fluid flows, porous media and many other complex systems [4] in the last decade. Such Cellular Automata simulate the mesoscopic behavior of the fluid inside an automaton cite. If the evolution rule preserves physical quantities, such as the local mass density and the local momentum density, theoretically it can be proved that macroscopic behavior emerges conformant to PDEs describing the simulated physical phenomena.

The proposed LBM solver assumes that the nodes V in the system graph G introduced in Section 1 are connected according to a regular topology with dimension d defining the set of the edges E (e.g. a ring topology for $d = 1$, a torus for $d = 2$, an n-cube for higher values of d). In HPC systems this configuration can be easily obtained remapping the physical network connections logically among the processing elements. In Grid systems a ring topology can be efficiently obtained constructing a peer-to-peer overlay among the worker nodes. Recently some efficient algorithms for the autonomous organization of nodes in overlays have been proposed [1]. The DLB Lattice Boltzmann solver is an iterative process similar to the Iterated Dynamical Systems adopted by the diffusion equation models described in Section 2. Though, as the multiagent approaches, it is not

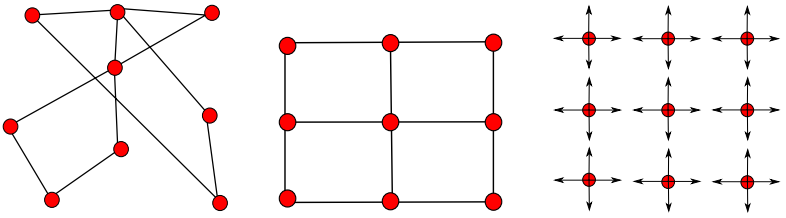


Fig. 1. Nodes arrangement from the physical network topology (a) to a regular overlay network (b) which is used as support to identify the neighbors of each node through the vectors $\{\boldsymbol{e}_i\}$ (c)

necessary to solve a migration scheduling problem because the information in the automaton cells state how the work load has to be transferred.

In general, we assume n automaton cells, corresponding to the computing nodes, are organized in a d-dimensional space so that each node $p \in V$ has exactly degree $2d$. In this way a cell is connected to its $2d$ neighbors along the processor links. The links are modelled as directions corresponding to $2d$ unary link vectors $\boldsymbol{e}_i$. For example, in the case of $d = 2$ the link vectors are

$$\boldsymbol{e}_0 := (0,0) \qquad \boldsymbol{e}_i := \left(\cos\left(\frac{\pi(i-1)}{2}\right), \sin\left(\frac{\pi(i-1)}{2}\right)\right), i \in \{1,2,3,4\} \quad (3)$$

where the vector $\boldsymbol{e}_0$ corresponds to the central position of each cell.

The set $\{\boldsymbol{e}_i\}$ for a generic node $p \in V$ is constructed assigning a node p' s.t. $(p,p') \in E$ to one of the vectors $\boldsymbol{e}_i$. In this way the processor graph can be mapped into a lattice suited for the LBM computation. We also introduce the nearest neighbor function $NN : V \times \{\boldsymbol{e}_i\} \to V$ that associates to each node p the node p' connected to it through a direction vector $\boldsymbol{e}_i$. An example for $d = 2$ of this mapping is reported in Figure 1. A generic cell state is characterized by $2d$ real number quantities modelling the load a processing element is going to move toward a neighbor along its links. More formally, given a processor $p \in V$, at time t its load configuration is defined as $\boldsymbol{f}(p,t) = \{f_i(p,t) : i = 0,1,\ldots,2d\}$ where $f_i(p,t) \in \mathbb{R}^+$ are called load distributions and represent the work load that will be moved along the link i. On the basis of these quantities we evaluate the two variables $\rho(p,t)$ and $c(p,t)$ according to the following equations:

$$\rho(p,t) = \sum_{i=0}^{2d} f_i(p,t) \qquad \text{and} \qquad c(p,t) = \Pi(p,t) \quad (4)$$

where, as stated in Section 1, $\rho(p,t)$ is the total load for a node $p \in V$ at time t and $c(p,t)$ is the maximum node capability with respect to a generic performance measure $\Pi : V \times \mathbb{N} \to \mathbb{R}^+$.

The time-step automaton evolution consists of two phases, according to the common LBM stream-and-collide algorithm:

– *Collision*: the load within a cell is redistributed among the $2d$ directions and the rest load distribution. Original distributions $\boldsymbol{f}(p,t) = \{f_i(p,t) : i = 0, 1, \ldots, 2d\}$ are modified in $\boldsymbol{f}^*(p,t) = \{f_i^*(p,t) : i = 0, 1, \ldots, 2d\}$ applying an instantaneous transition, formalized by the following equation:

$$f_i^*(p,t) = f_i(p,t) + \Omega_i(p,t) \tag{5}$$

where the *collision operators* $\Omega_i(p,t)$ must satisfy the following conservation law:

$$\rho^*(p,t) = \sum_{i=0}^{2d} f_i^*(p,t) = \sum_{i=0}^{2d} f_i(p,t) = \rho(p,t) \tag{6}$$

Equivalently this corresponds to $\sum_{i=0}^{2d} \Omega_i(p,t) = 0$. That is, the work load on each node is conserved during the redistribution. In the classical LBM context the Equation (6) corresponds to the mass conservation principle.

– *Streaming*: the load distributions move along their direction towards adjacent cells/processors in the network. This transfer is described in the automaton by the following equation:

$$f_i(\mathrm{NN}(p, \boldsymbol{e}_i), t+1) = f_i^*(p,t) \tag{7}$$

where the function NN is the previously defined nearest neighbor function, which associates a node p with another node along the direction $\boldsymbol{e}_i$.

Composition of the two Equations. (5) and (7) leads to an evolution rule similar to the *Lattice Boltzmann Equation*:

$$f_i(\mathrm{NN}(p, \boldsymbol{e}_i), t+1) = f_i(p,t) + \Omega_i(p,t) \tag{8}$$

We use a common form of the collision operator Ω_i, known as BGK formulation (see[13, 3, 14]):

$$\Omega_i(f_i(p,t)) = \frac{1}{\tau}\left(f_i^{eq}(p,t) - f_i(p,t)\right) \tag{9}$$

where τ is called the *relaxation time* of the model. It has been proved theoretically that the LBMs with BGK collision operator are unconditionally unstable for $\tau < 1/2$ [18]. Thus, the *evolution equation* of a LBM is

$$f_i(\mathrm{NN}(p, \boldsymbol{e}_i), t+1) = f_i(p,t) + \frac{1}{\tau}\left(f_i^{eq}(p,t) - f_i(p,t)\right) \tag{10}$$

The choice for the specific form for the equilibrium distributions f_i^{eq} is

$$f_0^{eq}(p,t) = (1 - 2d\,\alpha(p,t))\;\rho(p,t) \qquad \text{and} \qquad f_i^{eq}(p,t) = \alpha(p,t)\;\rho(p,t) \tag{11}$$

where $\alpha(p,t)$ is an over-relaxation term [8] introduced to speed up the convergence:

$$\alpha(p,t) = \frac{1}{2d}\left(\rho(p,t) - c(p,t)\right) - b\left(1 - \frac{\rho(p,t)}{c(p,t)}\right) \tag{12}$$

with $b \ll 1$ over-relaxation constant ruling the numerical stability. Using a Chapman-Enskog expansion, it can be proved that the model described above approximates a continuous diffusion equation up to the second order, where the diffusion constant is proportional to the parameter τ [4].

Even though the synchronous automaton evolution is more suited for typical HPC all-to-all optimized communications, it is possible to reproduce the discrete time stepping on Grid systems using the WsNotification standard [5] or analogous Grid messaging systems.

4 Numerical Experiments

In this section we will present numerical experiments results for the LBM solver. These results will be compared to the solutions by both diffusive schemas and swarm intelligence proposed in Section 2. After the comparison we will state our conclusion. For these numerical insights a ring processor graph is taken in account. Ring topology has been chosen because of the lowest connectivity as possible and also because it is the most frequent and natural overlay topology adopted by peer-to-peer Grid systems.

In the first set of simulations the LBM solver performances are compared w.r.t. the Semi-iterative schema on an HPC system. The load balance will be calculated for a number of processors ranging from 32 up to 1024. The maximum allowed number of iterations is 1000. The initial load distribution on the nodes will be generated randomly. The convergence criterion for each test is $err(t) = \sum_{p \in V}(\rho(p,t) - c(p,t)) < \epsilon$, with $\epsilon = 10^{-3}$. For the comparison between the Semi-iterative schema and the LBM solver we considered the following system capacity function $c(p,t) = \overline{c} = \sum_{p \in V} \rho(p,0)/|V|$.

For each test the optimal values of the models parameters are considered. In Table 1 the two iterative schemas are compared by the number of iterations required to reach the steady state $c(p,t)$ according to the criterion stated above. It is particularly interesting to notice how the Lattice Boltzmann approach scales better than the accelerated diffusion model for larger systems. Moreover we have not yet found a formal relation between the network topology and the LBM relaxation time τ, so we had to tune it numerically for each considered graph, while the theoretical optimal parameters for the diffusion models are well-known (see Section 2).

As the LBM solver and the Semi-iterative schema are different numerical approaches to the same class of models, we can focus our attention on the way they damp load distribution frequencies during the diffusion. Studying the dynamical evolution of the distributions toward the equilibrium it has been noticed that the LBM solver results to be faster thanks to its capabilities of damping low wavelengths. In particular, the diffusion process takes advantage from the adaptive parameter $\alpha(p,t)$, defined in Equation (12), that keeps on pushing the exceeding load away from the nodes. This adaptive schema grants, in general, that the convergence ratio does not slow down, even if the nodes are slightly out-of-equilibrium. Instead, the convergence parameter for the Semi-iterative schema of Equation (2),

Table 1. Number of iterations for the LBM and the Semi-iterative schema

# of nodes	LBM	Semi-iterative
32	32	20
64	55	43
128	112	93
256	144	198
512	313	420
1024	783	897

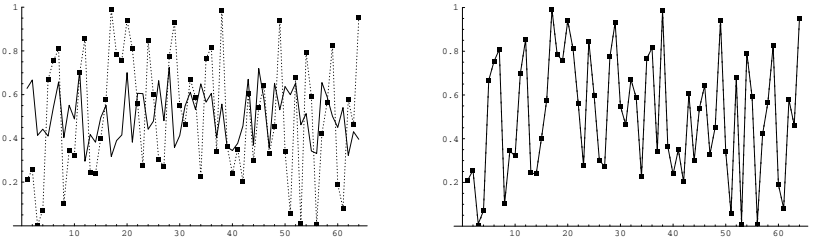
Table 2. LBM performance w.r.t. Semi-iterative schema for strongly biased initial configuration

# of nodes	LBM	Semi-iterative
32	103	20
64	193	39
128	342	77
256	719	152
512	>1000	303
1024	>1000	611

depending on the time only, seems to become less efficient as the load distribution becomes smoother and the low frequencies components become dominant.

The second set of tests compares the performance between the LBM and the Semi-iterative schema in the case of a strongly biased initial condition. In particular, we set only one node with most of the load in the system, while the others are strongly under-loaded. Under this condition the Semi-iterative schema performs much better than the proposed LBM model. More investigation on this singularity will be performed in future, even thought we are quite confident that this abnormal behaviour is related to the poor high frequency damping capabilities exhibited by all the Lattice Boltzmann cellular automata [18, 4]. Some results for these tests are reported in Table 2.

For the Grid scenario we compared the LBM solver approach with the DLB solution proposed by the Swarm intelligence model described in Section 2. Three agents have been assigned to each node in the system: SearchMin, SearchMax and Transfer agents respectively. The agent teams maximum size has been chosen so that it is proportional to the number of nodes in the graph. E.g., for $|V| = 32$ we found, after some empirical tests, that teams of at most 15 agents give best results without introducing agent overcrowding overhead. As a general rule, we noticed that a number of agents of about half of the computational nodes gives the best convergence speed. The network connecting the nodes is still arranged as a ring. The initial load configuration is generated randomly and also the node capacities are chosen randomly. The initial load configuration, in particular, is normalized to the nodes capacities, so that at the steady state the system is fully saturated. In Figure 2 we reported the initial and the final configurations



(a) (b)

Fig. 2. Initial configuration with system capacity (a). Steady state configuration reached in about 50 iterations (b).

Table 3. LBM performance w.r.t. swarm intelligence for a Grid environment

# of nodes	LBM	Swarm
32	43	150
64	52	269
128	113	399
256	210	737
512	404	>1000
1024	795	>1000

for a graph containing 64 nodes. On the abscissa axes the node ID is reported, while on the y axes we reported the work load assigned to each node. In the initial configuration picture we reported also the system capacities (dashed line). Analogous results are obtained with a swarm intelligence schema, but the LBM convergence takes a significantly lower number of iterations. Some results on the convergence speed are reported in Table 3.

The reasons for the different convergence speed could be searched in the LBM solver capabilities of calculating the load transfer without the need of moving agents across the network. Furthermore, fixed payload moved by agents can affect the swarm approach in a significant way: a too small payload can slow down the convergence but transferring a large work load with each agent could jeopardise the whole balancing process. In the worst case a too large payload could bring the balancing to a starvation, with the nodes swapping the work load forever. For these particular tests we tried to tune the payload for each agent so that the better convergence performance are obtained. Further investigation should be performed to better understand the swarm intelligence convergence performances with respect to both the used number of agents and the quantity of computational load each agent can move across the system.

Both the LBM and the Swarm intelligence DLB solver exhibit a proper behavior even in the case of globally overloaded environments. That is, they distribute the exceding load equally even in the case of low system capacity. An example of such behavior is shown in Figure 3. Let us remind that the node IDs are reported on the x axes and that on the y axes the work load is reported.

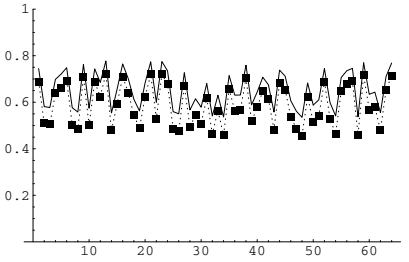


Fig. 3. Steady state reached by the LBM solver in the case of a globally overloaded system

5 Conclusion

In this paper, we have introduced a novel DLB solver which is inspired by the Lattice Boltzmann cellular automata usually used to simulate complex fluids dynamics. In particular, we modified the original LBM to approximate a diffusive phenomenon that suitably solves the DLB problem. We compared the performances of the proposed LBM solver with the ones of an accelerated diffusion equation model usually adopted in DLB on HPC systems. The model has been also compared to swarm intelligence models commonly adopted in Grid context load balancing problems. For both the comparisons it has been shown how the LBM behaves more efficiently than the standard approaches to the DLB problem. Only for ad-hoc initial configurations the LBM always presents worse performances than the other methods.

References

[1] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin, *Fast construction of overlay networks*, Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2005, pp. 145–154.

[2] D.P Bertzekas and J.N. Tsitsiklis, *Parallel and distributed computing, 2nd edition*, Athena Scientific, NH, 1997.

[3] H. Chen, S. Chen, and W. Matthaeus, *Recovery of the Navier Stokes equation using a Lattice Gas Boltzmann method*, Physical Review A **45** (1992), 5339–5342.

[4] B. Chopard and M. Droz, *Cellular automata modelling of physical systems*, Cambridge University Press, Cambridge, 1998.

[5] OASIS Consortium, *Ws-notification specification 1.3 (pubblic draft 2)*, 2006, available at http://www.oasis-open.org/committees/wsn.

[6] G. Cybenko, *Dynamic load balancing for distributed memory multi-processors*, Journal of Parallel and Distributed Computing **7** (1989), 279–301.

[7] R. Diekmann, A. Frommer, and B. Monien, *Efficient schemes for nearest neighbor load balancing*, Parallel Comput. **25** (1999), 789–812.

[8] H. Fang, R. Wan, and Z. Lin, *Lattice boltzmann model with nearly constant density*, Physical Review E **66** (2002), 036314.

[9] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke, *Grid Computing: Making the Global Infrastructure a Reality*, ch. The Physiology of the Grid, pp. 217–249, Wiley, 2003.

[10] Y.F. Hu and R.J. Blake, *An improved diffusion algorithm for dynamic load balancing*, Parallel Computing **25** (1999), 417444.

[11] G. Karagiorgos and N.M. Missirlis, *Accelerated diffusion algorithms for dynamic load balancing*, Information Processing Letters **84** (2002), 61–67.

[12] A. Montresor, H. Meling, and O. Babaoglu, *Messor: Load-balancing through a swarm of autonomous agents*, Tech. report, Dept. of Computer Science, University of Bologna, 2002.

[13] E. P. Gross P. L. Bhatnagar and M. Krook, *A model for collision processes in gases. I. Small amplitude processes in charged and neutral one component systems*, Physical Review **94** (1954), 511–525.

[14] Y. H. Qian, D. D'Humieres, and P. Lallemand, *Lattice BGK models for Navier-Stokes equation*, Europhysics Letters **17** (1992), 479–484.

[15] M. Resnick, *Turtles, termites, and traffic jams: explorations in massively parallel microworlds*, MIT Press, Cambridge, MA, USA, 1994.

[16] D. Rothman and S. Zaleski, *Lattice-gas cellular automata: Simple models of complex hydrodynamics*, Cambridge University Press, UK, 1997.

[17] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel diffusion schemes for repartitioning of adaptive meshes*, Journal of Parallel and Distributed Computing **47** (1997), 109124.

[18] J. D. Sterling and S. Chen, *Stability analysis of Lattice Boltzmann methods*, Journal of Computational Physics **123** (1996), 196–206.

[19] Sauro Succi, *The lattice Boltzmann equation, for fluid dynamics and beyond*, Oxford University Press, UK, 2001.

[20] Y. Wang, J. Liu, and X. Jin, *Modeling agent-based load balancing with time delays*, IAT '03: Proc. of the IEEE/WIC Int. Conf. on Intelligent Agent Technology, IEEE Computer Society, 2003, pp. 189–196.

Trust Enforcement in Peer-to-Peer Massive Multi-player Online Games

Adam Wierzbicki

Polish-Japanese Institute of Information Technology,
ul. Koszykowa 86, Warsaw, Poland
adamw@pjwstk.edu.pl

Abstract. The development of complex applications that use the Peer-to-Peer computing model is restrained by security and trust management concerns, despite evident performance benefits. An example of such an application of P2P computing is P2P Massive Multi-user Online Games, where cheating by players is simple without centralized control or specialized trust management mechanisms. The article presents new techniques for trust enforcement that use cryptographic methods and are adapted to the dynamic membership and resources of P2P systems. The proposed approach to trust management differs significantly from previous work in the area that mainly used reputation. The paper describes a comprehensive trust management infrastructure fore P2P MMO games that enables to recognize and exclude cheating players while keeping the performance overhead as low as possible. While the architecture requires trusted centralized components (superpeers), their role in trust management is limited to a minimum and the performance gains of using the P2P computing model are preserved.

1 Introduction

Apart from file-sharing applications, the Peer-to-peer (P2P) computing model has found practical use in distributed directories for applications such as IP telephony, content distribution or P2P backup. The improved performance and availability of the P2P model has proven useful for applications that have to scale to very large numbers of users. However, creating complex P2P applications still faces several obstacles: it is very difficult to solve coordination, reliability, security and trust problems without the use of centralized control. These problems have been overcome in some respects, however, much work still needs to be done before the P2P computing model can be applied to complex applications that have high security or reliability requirements.

We consider how the P2P model could be applied to build a Massive Multiplayer Online (MMO) game. At present, scalability issues in MMO games are usually addressed with large dedicated servers or even clusters. According to white papers of a popular multi-player online game – TeraZona [17] – a single server may support 2000 to 6000 simultaneous players, while cluster solutions used in TeraZona support up to 32 000 concurrent players. The client-server approach has a severe weakness, which is the high cost of maintaining the central processing point. Such an architecture is too expensive to support a set of concurrent players that is by an order

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1163 – 1180, 2006.
© Springer-Verlag Berlin Heidelberg 2006

or two orders of magnitude larger than the current amounts. To give the impression of what scalability is needed – games like Lineage report up to 180 000 concurrent players in one night.

MMO games can benefit from the application of the P2P model. However, in a P2P MMO game, issues related to trust become of crucial importance, as shall be shown further in this paper. How can a player be trusted not to modify his own private state to his advantage? How can a player be trusted not to look at the state of hidden objects? How can a player be trusted not to lie, when he is accessing an object that cannot be used unless a condition that depends on the player's private state is satisfied? In this paper, we show how all of these questions can be answered, and present the proposed protocols in detail. We also address performance and scalability issues of the proposed trust management mechanisms and give an extended presentation of related work.

The proposed trust management architecture does not use reputation, but relies on cryptographic mechanisms that allow players to verify fairness of moves. Therefore, we call our approach to trust management "trust enforcement". Our trust management architecture for P2P MMO games makes used of trusted central components. It is a hybrid P2P system, the result of a compromise between the P2P and client-server models. A full distribution of the trust management control would be too difficult and too expensive. On the other hand, a return to the trusted, centralized server would obliterate the scalability and performance gains achieved in the P2P MMO game. Therefore, the proposed compromise tries to preserve performance gains while guaranteeing fairness of the game. To this end, our trust management architecture does not require the use of expensive encryption, which could introduce a performance penalty.

In the next section, security and trust issues in P2P MMO games are reported and illustrated by possible attack scenarios. In section 3, some methods of trust management for P2P MMO games will be proposed. Section 4 presents a security analysis that demonstrates how the reported security and trust management weaknesses can be overcome using our approach. Section 5 discusses the performance of the presented protocols. Section 6 describes related work, and section 7 concludes the paper.

2 Security Issues in P2P MMO Games

The attacks described in this section illustrate some of the security and trust management weaknesses of P2P game implementations so far. We shall use a working assumption that the P2P MMO game uses some form of Dynamic Hash Table (DHT) routing in the overlay network, without assuming a specific protocol. In the following section, we describe a trust management architecture that can be used to prevent the attacks described in this section.

2.1 Private State: Self-modification

P2P game implementations that allow player to manage their own private state [3] do not exclude the possibility that a game player can deliberately modify his own private

state (e.g. experience, possessed objects, location, etc.) to gain advantage over other game players. A player may also alter decisions already made in the past during player-player interaction that may affect the outcome of such an interaction.

2.2 Public State: Malicious / Illegal Modifications

In a P2P MMO game, updates of public state may be handled by a peer who is responsible for a public object. The decision to update public state depends then solely on this peer – the coordinator. Furthermore, the coordinator may perform malicious modifications and multicast illegal updates to the group. The falsified update operation may be directly issued by the coordinator and returned back to the group as a legal update of the state. Such an illegal update may also be issued by another player that is in a coalition with the coordinator, and accepted as a legal operation.

2.3 Attack on the Replication Mechanism

When state is replicated in a P2P game, replication players are often selected randomly (using the properties of the overlay to localize replicated data in the virtual network). This can be exploited when the replication player can directly benefit from the replica of the knowledge he/she is storing (i.e. the replication player is in the region of interest and has not yet discovered the knowledge by himself).

2.4 Attack on P2P Overlays

In a P2P overlay (such as Pastry), a message is routed to the destination node through other intermediary nodes. The messages travel in open text and can be easily eavesdropped by competing players on the route. The eavesdropped information can be especially valueable if a player is revealing his own private state to some other player (player–player interaction). In such case, the eavesdropping player will find out whether the interacting players should be avoided or attacked.

The malicious player may also deliberately drop messages that he is supposed to forward. Such an activity will obstruct the game to some extent, if the whole game group is relatively small.

2.5 Conclusion from Described Attacks

Considering all of the attacks described in this chapter, a game developer may be tempted to return to the safe model of a trusted, central server. The purpose of this article is to show that this is not completely necessary. The trust management architecture presented in the next section will require trusted centralized components. However, the role of these components, and therefore, the performance penalty of using them, can be minimized. Thus, the achieved architecture is a compromise between the P2P and client-server models that is secure and benefits from increased scalability due to the distribution of most game activities.

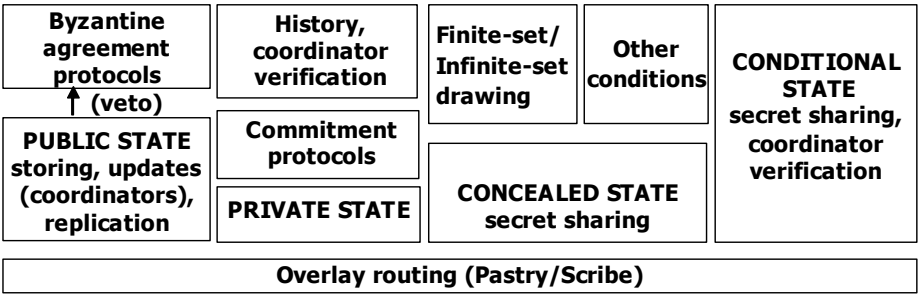


Fig. 1. A trust management architecture for P2P MMO games

3 Trust Enforcement for P2P MMO Games

In this section, we propose a trust management architecture for P2P MMO games. Before the details of the proposed architecture will be described, let us shortly discuss the used concepts of "trust" and "trust management".

3.1 Trust Enforcement

In much previous research, the notion of trust has been directly linked to reputation that can be seen as a measure of trust. However, some authors [20,23] have already defined trust as something that is distinct from reputation. In this paper, trust is defined (extending the definition of [20]) as a subjective expectation of an agent about the behavior of another agent. This expectation relates the behavior of the other (trusted) agent to a set of normative rules of behavior, usually related to a notion of fairness or justice. In the context of electronic games, fair behavior is simply defined as behavior that obeys all rules of the game. In other words, an agent trusts another agent if the agent believes that the other agent will behave according to the rules of the game.

Trust management is used to enable trust. Reputation systems are a type of trust management mechanism that assigns a computable measure of trust to any agent on the basis of the observed or reported history of that agent's behavior. However, any mechanism that allows to determine who can, and who cannot be trusted, is a trust management mechanism.

Our approach to trust management does not rely on reputation, which is usually vulnerable to first-time cheating. We have attempted to use cryptographic methods for verification of fair play, and have called this approach "trust enforcement". To further explain this approach to trust management, consider a simple "real-life" analogy. In many commercial activities (like clothes shopping) the actors use reputation (brand) for trust management. On the other hand, there exist real-life systems that require and use trust management, but do not use reputation. Consider car traffic as an example. Without trust in the fellow drivers, we would not be able to drive to work every day.

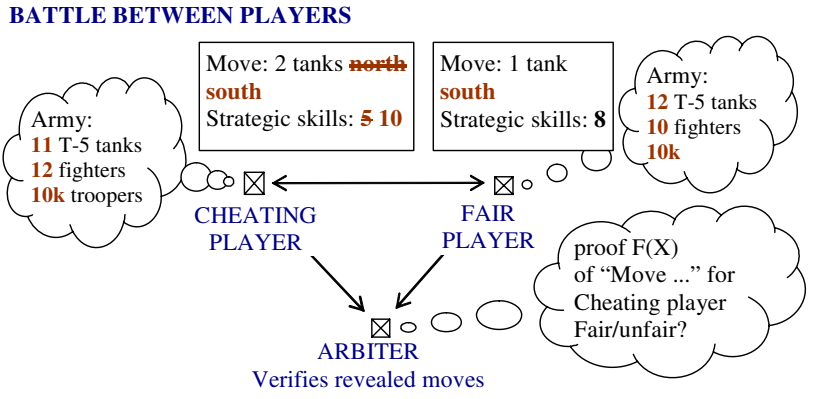


Fig. 2. Preventing self-modification of private state

However, we do not know the reputation of these drivers. The reason why we trust them is the existence of a mechanism (the police) that enforces penalties for traffic law violations (the instinct for self-preservation seems to be weak in some drivers). This mechanism does not operate permanently or ubiquitously, but rather irregularly and at random. However, it is (usually) sufficient to enable trust.

Note that the issue of what to do with those who violate fair rules is beyond the scope of our research. We propose simply that players in a P2P MMO game that have cheated should be excluded from the game, but this is by no means the only possible approach. Another possible approach could be to calculate reputation of players based on the results of fairness verification, and to allow other players to decide whether they want to interact with cheating players.

3.2 The Proposed Trust Management Architecture

The trust management architecture proposed in this paper is visualized on Figure 1. It uses several cryptographic primitives such as commitment protocols and secret sharing. It also uses certain distributed computing algorithms, such as Byzantine agreement protocols. These primitives shall not be described in detail in this paper for lack of space. The reader is referred to [8-11]. The relationships between the components of the trust management architecture will be described in this section.

Our trust management architecture for P2P MMO games will use partitioning of game players into groups, like in the approach of [3]. A group is a set of players who are in the same region. All of these players can interact with each other. However, players may join or depart from a group at any time. Each group must have a trusted coordinator, who is not a member of the group (he can be chosen among the players of another region or be provided by the game managers). The coordinator must be trusted because of the necessity of verifying private state modifications (see below). However, the purpose of the trust management architecture is to limit the role of the coordinator to a minimum. Thus, the performance gains from using the P2P model may still be achieved, without compromising security or decreasing trust.

3.3 Game Play Scenarios Using Proposed Trust Management Architecture

Let us consider a few possible game play scenarios and describe how the proposed trust management mechanisms would operate. In the described game scenarios that are typical for most MMO games, the game state can be divided into four categories:

- Public state is all information that is publicly available to all players and such that its modifications by any player can be revealed.
- Private state is the state of a game player that cannot be revealed to other players, since this would violate the rules of the game.
- Conditional state is state that is hidden from all players, but may be revealed and modified if a condition is satisfied. The condition must be public (known to all players) and cannot depend on the private state of a player.

Concealed state is like conditional state, only the condition of the state's access depends on the private state of a player.

Player joins a game. From the bootstrap server or from a set of peers, if threshold PKI is used, the player must receive an *ID* and a public key certificate $C=\{ID, K_{pub}, s_{join}\}$ (where s_{join} is the signature of the bootstrap server (or the peers), and K_{pub} is the public key that forms a pair with the secret key k_{priv},) that allows strong and efficient authentication (see next section. Note that the keys will not be used for data encryption). The player selects a game group and reports to its coordinator (who can be found using DHT routing). The coordinator receives the player's certificate. The player's initial private state (or the state with which he joins the game after a period of inactivity) is verified by the coordinator. The player receives a **verification certificate** (*VC*) that includes a date of validity and is signed by the coordinator.

Player verifies his private state. In a client-server game, the game server maintains all private state of a user, which is inefficient. In the P2P solution, each player can maintain his own private state [3], causing trust management problems. We have tried to balance between the two extremes. It is true that a trusted entity (the coordinator) must oversee modifications of the private state. However, it may do so only infrequently. Periodically or after special events, a player must report to the coordinator for verification of his private state. The coordinator receives the initial (recently published) private state values and a sequence of modifications that he may verify and apply on the known private state. For each modification, the player must present a proof. If the verification fails, the player does not receive a confirmation of success. If it succeeds, the player is issued a VC that has an extended date of validity and is signed by the coordinator. Verification by a coordinator is done by "replaying" the game of the user from the time of the last verification to the present. The proofs submitted by the player must include the states of all objects and players that he has interacted with during the period.

Note that a verification may be performed for just a part of the private state, and the issued VC may specify which elements of the private state have been verified. Verification may also, for efficiency reasons, be performed for just a random part of the period, if the player submits all intermediate state changes.

After verification, the player maintains and modifies his own state. As shall be explained below, the player collects testimonies from other players that are sufficient to prove the correctness of his private state modifications.

Player interacts with a public object. Peer-to-peer overlays (like DHTs) provide an effective infrastructure for routing and storing of public knowledge within the game group. Any public object of the game is managed by some peer. The player issues modification requests to the manager *M* of the public object. The player also issues a commitment of his action *A* that can be checked by the manager. A commitment protocol works by publishing a value (the commitment) for each operation on the private state. The commitment could be, for example, a hash function of an object. The commitment is used when a player needs to prove that his private state has not been illegally modified. Let us denote the commitment by C*(ID, A)* (the commitment could be a hash function of some value, signed by the player).

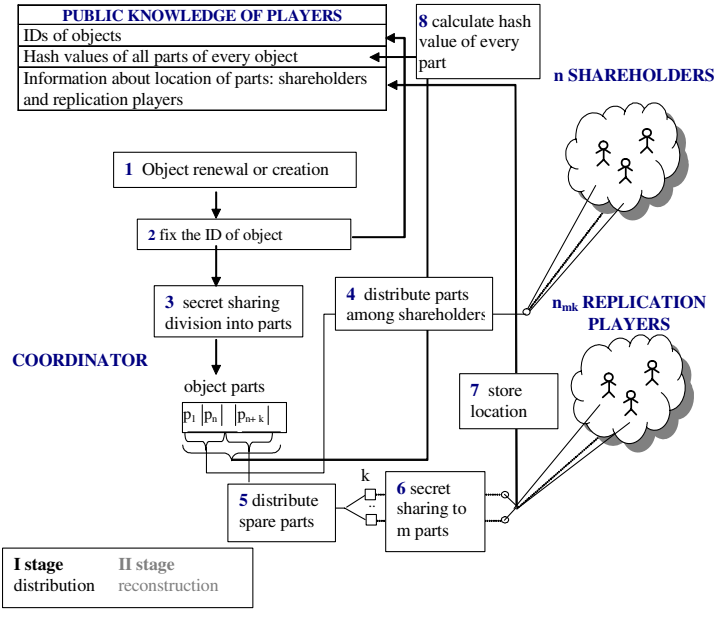


Fig. 3. Distribution of concealed and conditional state

Commitments should be issued whenever a player wishes to access any object, and for decisions that affect his private state. Commitments may also be used for random draws [1]. It will be useful to regard commitments as modifications of public state that is maintained for each player by a peer that is selected using DHT routing, as for any public object.

The request includes the action that the player wishes to execute, and the player's validation certificate, *VC*. Without a valid certificate, the player should not be allowed to interact with the object. If the certificate is valid, and the player has issued a correct commitment of the action, the manager updates his state and broadcasts an update

message. The manager also sends a signed testimony $T=\{t, A, S_i, S_{i+1}, P, s_M\}$ to the player. This message includes the time t and action A, state of the public object before (S_i) and after the modification (S_{i+1}) and some information P about the modifying player (f. ex., his location). The player should verify the signature s_M of the manager on the testimony. The manager of the object then sends an update of the object's state to the game group.

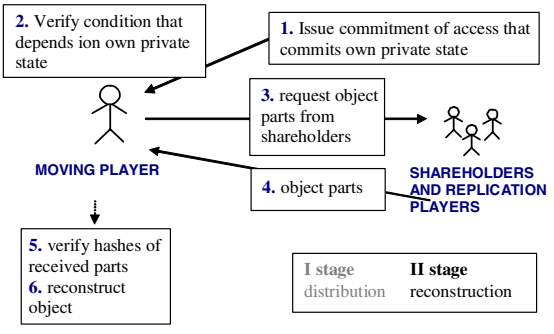


Fig. 4. Reconstruction of concealed and conditional state

If any player (including the modifying player, if T is incorrect) rejects the update (issues a veto), the coordinator sends T to the protesting player, who may withdraw his veto. If the veto is upheld, a Byzantine agreement round is started. (This kind of Byzantine agreement is known as the Crusader's protocol.)

Note that if a game player has just modified the state of a public object and has not yet sent an update, he may receive another update that is incorrect, but will not veto this update, but send another update with a higher sequence number.

To decide whether an update of the public state is correct, players should use the basic physical laws of the game. For example, the players could check whether the modifying player has been close enough to the object. Players should also know whether the action could be carried out by the modifying player (for example, if the player cuts down a tree, he must possess an axe). This decision may require knowledge of the modifying player's private state. In such a case, the modification should be accepted if the modifying player will undergo validation of his private state and present a validation certificate that has been issued after the modification took place.

Player executes actions that involve randomness. For example, the player may search for food or hunt. The player uses a fair random drawing protocol [1] (usually, to obtain a random number). This involves the participation of a minimal number (for instance, at least 3) of other players that execute a secret sharing together with the drawing player. The drawing player chooses a random share l_0 and issues a commitment of his share $C(ID, l_0)$ to the manager of his commitments (that are treated as public state). The drawing player receives and keeps signed shares $p_1,...,p_n$ from the other players, and uses them to obtain a random number. The result of the drawing can be obtained from information that is part of constant game state (drawing tables).

Player meets and interacts with another player. For example, let two players fight. The interaction must be overseen by an arbiter, who can be any player. The two players should first check their validation certificates and refuse the interaction if the certificate of the other player is not valid. The VCs of the players are also examined by the arbiter to avoid that one of the players denies the interaction by falsely claiming that his opponent's VC is invalid. Before the interaction takes place, both players may carry out actions $A_1,...,A_k$ that modify their private state (like choosing the weapon they will use). The players must issue commitments of these actions. The commitments must also be sent to the arbiter. The actions remain secret until they can be used to improve the game result of a player. Then, the actions and relevant private state are revealed, and commitments can be used to prove correctness. The arbiter will record the commitments and the revealed actions (as shown on Figure 2). After the interaction is completed, the arbiter will send both players a signed testimony about the interaction.

If the interaction involves randomness, the players draw a common random number using a fair drawing protocol (they both supply and reveal shares; shares may also be contributed by other players).

Finally, the players reveal their actions to each other and to the arbiter. The results of the interaction are also obtained from fixed game information and affect the private states of both players. The players must modify their private states fairly, otherwise they will fail verification in the future (this includes the case if a player dies. Player death is a special case. It is true that once a player is dead, he can continue to play until his VC expires. This can be corrected if the player who killed him informs the group about his death. Such a death message forces any player to undergo immediate verification if he wishes to prove that he is not dead). Note that at any time, both players are aware of the fair results of the interaction, so that a player who has won the fight may refuse further interactions with a player who decides to cheat.

Player executes an action that has a secret outcome. For example, the player opens a chest using a key. The chest's content is conditional state – other players should not be aware of what is inside the chest. The player will modify his private state after he finds out the chest's contents. To determine the outcome, the player will reconstruct conditional or concealed state.

Conditional and concealed state can be managed using secret sharing and commitment protocols, as described in [1]. The protocol developed in [1] concerned drawing from a finite set, but can be extended to handle any public condition. However, concealed state has not been considered in our previous research on fairness of P2P games. Therefore, we needed to extend and modify our results in order to provide trust management of concealed state.

The protocol for concealed and conditional state management has been shown on Figures 3 and 4. It is divided into two phases. The distribution phase needs to be executed by the coordinator whenever a concealed public object is renewed or created (step 1 on Figure 3). The coordinator divides the object into a fixed number of shares (step 3) that are sent to chosen players (shareholders in step 5), and to a number of replication players who are not members of the group (this is done in step 4 to decrease the likelihood of coalitions). The coordinator also calculates hash values of the shares that are public state (step 8). Apart from the initializing of the state, the coordinator does not participate in its management.

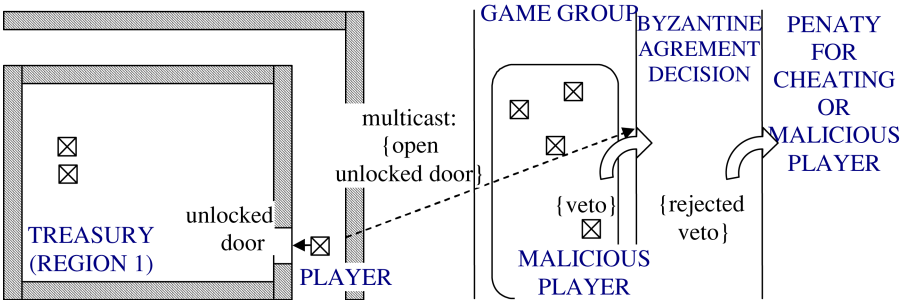


Fig. 5. Byzantine agreement / veto protection of public state updates

The shareholders may leave the game at any time, and the object parts from the replication players are used instead. If there are not enough parts to reconstruct an object, the object must be renewed by the coordinator. This replication approach is more resistant to coalitions than the approach proposed in [3] (relying on Pastry). Replicating the object parts by random players from the group increases the likelihood that one shareholder will receive more than one part. Then, this shareholder may form a coalition with a smaller number of other players to reconstruct the object. Also, the backup mechanism used in [3] may increase the number of object parts kept by a single player. The protocol described here can also be used for random draws from a finite set (see [1]).

The second phase of the protocol is the reconstruction phase. In step 1 on Figure 4, the player issues a commitment of his action C*(ID, A)* that is checked by the shareholders. If the condition is public but depends on the player's private state, the player decides himself in step 2 whether the condition is fulfilled (he will have to prove the condition's correctness during verification in the future). In this case, the issued commitment must also commit the player's private state before the object is accessed (the player must also keep a copy of the relevant private state). If the condition to access an object is secret, the condition itself should be treated as a conditional public object. After the player checks that the condition is fulfilled, he requests and receives the object shares (steps 3 and 4) and, after verifying that the shares are correct, reconstructs the concealed or conditional object (steps 5 and 6).

When the player has reconstructed the object, he must keep the shares for verification. At the same time, the testimony issued by the shareholders will include a value of the condition that will allow the coordinator to verify the answer of the player. The elements of private state that may constitute a zero-knowledge condition should be defined during game design.

Note that concealed and conditional public objects can have states that are modified by players. If this is the case, then each state modification must be followed by the distribution phase of the protocol for object management.

3.4 Authentication Requirements

A P2P game could use many different forms of authentication. At present, most P2P applications use weak authentication based on nick names and IP addresses (or IDs

that are derived from such information). However, it has been shown that such systems are vulnerable to the Sybil attack [13].

Most of the mechanisms discussed in this paper would not work if the system would be compromised using the Sybil attack. An attacker that can control an arbitrary number of clones under different IDs could use these clones to cheat in a P2P game. The only way to prevent the Sybil attack is to use a strong form of authentication, such as based on public-key cryptography. Public key cryptography will be used in our trust management architecture for authentication and digital signatures of short messages, but not for encryption.

In a P2P game, authentication must be used efficiently. In other words, it should not be necessary to repeatedly authenticate peers. The use of authentication could depend on the game type. For instance, in a closed game, authentication could occur only before the start of the game. Once all players are authenticated, they could agree on a common secret (such as a group key) that will be used to identify game players, using a method such as the Secure Group Layer (SGL) [15]. Alternatively, these users could exchange public keys.

Let us briefly discuss here how such an authentication could be implemented. The well-known solution is the PKI infrastructure. This solution also has the advantage of direct availability and possibility of implementation. However, developers of P2P games may be concerned about the single point of failure, lack of anonymity, or insufficient security of PKI [14]. The use of global PKI is unnecessary, since the bootstrap game server may issue certificates for players that are later used for authentication. There may exist solutions better suited to the needs of P2P games, such as the Web-of-Trust model, or solutions based on trust graphs. Also, we recognize that the use of local names as proposed in Simple Public Key Infrastructure (SPKI) [16] is more suited to P2P applications and should be a more scalable approach for P2P games. Another solution that is well suited to the P2P model is the use of threshold cryptography for distributed PKI [19].

4 Security Analysis

In this section, the attacks illustrated in previous paragraphs will be used to demonstrate how the proposed protocols protect the P2P MMO game.

4.1 Private State: Self-modification

Self-modification of private state can concern the parameters of a player, the player's secret decisions that affect other players, or results of random draws. The first type of modification is prevented by the need to undergo periodic verification of a private player's parameters. The verification is done by the coordinator on the basis of an audit trail of private state modification that must be managed by any player. Each modification requires proof signed by third parties (managers of other game objects, arbiters of player interactions). Any modification that is unaccounted for will be rejected by the coordinator. Players may verify that their partners are fair by checking a signature of the coordinator on the partner's private state. If a player tries to cheat during an interaction with another player by improving his parameters, he may succeed, but will not pass the subsequent verification and will be rejected by other players.

Modification of player's move decisions or results of random draws is prevented by the use of commitment protocols. The verification is made by an arbiter, who can be a randomly selected player. The verification is therefore subject to coalition attacks; on the other hand, making the coordinator responsible for this verification would unnecessarily increase his workload.

Note that in order for verification to succeed, the coordinator must possess the public key certificates of all players who have issued proof about the player's game. (If necessary, these certificates can be obtained from the bootstrap server). However, the players who have issued testimony need not be online during verification.

A player may try to cheat the verification mechanism by "forgetting" the interactions with objects that have adversely affected the player's state. This approach can be defeated in the following way. A player that wishes to access any object may be forced to issue a commitment in a similar manner as when a player makes a private decision. The commitment is checked by the manager of the object and must include the time and type of object. Since the commitment is made prior to receiving the object, the player cannot know that the object will harm him. The coordinator may check the commitments during the verification stage to determine whether the player has submitted information about all state changes.

4.2 Public State: Malicious / Illegal Modifications

We have suggested the use of Byzantine algorithms further supported by a veto mechanism (Crusader's protocol) to protect public state against illegal/malicious modifications. Any update request on the public state shall be multicast to the whole game group. The Byzantine verification within the group shall only take place when at least one of the players vetoes the update request of some other player. The cheating player as well as the player using the veto in unsubstantiated cases may be both penalized by the group by exclusion from the game (see figure 5). Such mechanism will act mostly as a preventive and deterring measure, introducing the performance penalty only on an occasional-basis.

The protection offered by Byzantine agreement algorithms has been discussed in [9]. It has been shown that the algorithms tolerate up to a third of cheating players (2N+1 honest players can tolerate N cheating players). Therefore, any illegal update on the public state will be excluded as long as the coalition of the players supporting the illegal activity does not exceed third of the game group. We believe such protection is far more secure than coordinator-based approach presented in [3] and tolerable in terms of performance. Performance could be further included if hierarchical Byzantine protocols are used [18].

4.3 Attacks on P2P Overlays

In our security architecture, players do not reveal sensitive information. A player does not disclose his own private state, but only commitments of this state. Concealed or conditional state is not revealed until a player receives all shares. If the P2P overlay is operating correctly and authentication is used to prevent Sybil attacks, the P2P MMO game should be resistant to eavesdropping by nodes that route messages without resorting to strong encryption. A secure channel is needed during the verification of a players private state by the coordinator.

4.4 Concealed State: Attack on Replication Mechanism

Concealed state, as well as any public state in the game, must be replicated among the peers to be protected against loss. The solution offered in [3] uses the natural properties of the Pastry network to provide replication. However, we have questioned the use of this approach for concealed state, where the replicas cannot be stored by a random peer. The existence of concealed state has not been considered in [3], and therefore the authors of [3] did not consider the fact that replicas may reveal the concealed information to unauthorized players.

Table 1. Complexity of concealed state management

	COMMUNICATION	*COMPUTATIONAL*	*MEMORY*
DISTRIBUTION STAGE	O(N * (s + m*k))	1. secret sharing: 1.1.matrix inversion mod p (N * (k + 1) times) 1.2.matrix multiplication (N * (k + 1) times) 2.hashing (N * (s + m*k) times)	1.set of objects of concealed or conditional public state: O(N)
	SIMPLE	**COMPLEX**	**SIMPLE**
RECONSTRUCTION STAGE	O(s) / O(1)	1.secret sharing : 1.1.matrix inversion mod p 1.2.matrix multiplication 2.hashing (s times)	1.public knowledge (hashes, rep. shares location): O(N*s + m*k) 2.private knowledge (secret shares, own objects): O(N + l) 3.access history: O(s * l)
	SIMPLE	**SIMPLE**	**COMPLEX**

In our approach for replication of concealed state, replication players are selected from outside the game group. This eliminates the benefits offered by Pastry network. On the other hand, this approach also eliminates the security risks. Please note that in our approach a certain number of players must participate to uncover specific concealed information. Therefore, a coalition with the replication player is not beneficial for a player within the game group.

4.5 Revocation of Verification Certificates

Verification is performed by trusted group coordinators that can be superpeers operated by the game provider. However, it is possible that the a malicious player could somehow set up a malicious coordinator and therefore defeat the verification mechanism. To avoid this, the superpeers should be equipped with public key certificates that are signed by other superpeers (using a Web-of-Trust model, or by a

signle central authority). When a Verification Certificate is examined by a player, the player should check the superpeers signature. To do this, the player should obtain or posess the superpeers public key certificate, and validate this certificate.

5 Performance Analysis

We have tried to manage trust in a P2P MMO game without incurring a performance penalty that would question the use of the P2P model. However, some performance costs are associated with the proposed mechanisms. Our initial assumption about partitioning of game players into groups (sets of players who are in the same region) is required for good performance.

Byzantine protocols have a quadratic communication cost, when a player disagrees with the proposed decision. Therefore, their use in large game groups may be prohibitive. This problem may be solved by restricting the Byzantine agreement to a group of superpeers that maintain the public state (an approach already chosen by a few P2P applications, such as OceanStore). Another possibility is the use of hierarchical Byzantine protocols that allow the reduction of cost but require hierarchy maintenance.

Since private state is still managed by a player, it incurs no additional cost over the method of Knutsson. The additional cost is related to the verification of a player's private state by a coordinator. The coordinator must "replay" the game of a player, using provided information, and verifying the proofs (signatures) of other players, as well as the modifications of the verified private state. This process may be costly, but note that a coordinator need not "replay" all of the game, but only a part (chosen at random). This may keep the cost low, while still deterring players from self-modification of private state.

The cost of maintenance of concealed or conditional state is highest in the initialization phase. This stage should be carried out only when an object is renewed. (when the object's state changes). The expense of this protocol may be controlled by reducing the constant number of object parts, at the cost of decreasing security. The number of object parts cannot be less than two. Table 1 shows a performance analysis of the most complex protocols in our trust management architecture: the protocols for management of concealed state. In the table, N is the number of objects (of concealed or conditional state); s – the number of shareholders; m*k – the number of replication players; and l – the number of objects stored by a peer. Note that computational cost of the distribution stage is complex – but this stage should be carried out only when an object is renewed. During most game operations, the cost of concealed state management is reasonable. The reconstruction phase has a constant const (fetching the parts of an object).

All the proposed protocols have allowed us to realize one goal: limit the role of the central trusted component of the system (the coordinator). The coordinator does not have to maintain any state for the players. He participates in the game occasionally, during distribution of concealed/conditional state and during verification of private state. The maintenance of public state remains distributed, although it requires a higher communication overhead.

6 Related Work

6.1 Massive Multiplayer Online Games

Role-playing games constitute a category of multi-player games, where the player assumes the role of a character in a virtual world. The game is most often based on some exciting stories or fables, where the players compete playing the role of such characters as: humans, elves, magicians or priests. The idea of the typical game is to accomplish various missions or quests with the controlled character, which involves certain typical types of operations. The character travels within the virtual world and interacts with other characters or objects. The knowledge of the virtual world possessed by the player may either increase with time as the player travels or remain constrained to just the closest surroundings. Early in the game, most of the virtual world remains unknown and the player has no knowledge where objects are located. By collecting objects and interacting with other characters (fighting, making friends, etc.) a player may, for instance, improve certain skills, accumulate experience, earn money or collect weapons or magic spells. All these properties can be used to improve the position over competing players. In most of RPG games the player accumulates ranking points that represent the result of his past competition with other characters. The objective is to survive within the virtual world and gain better ranking than other game players.

Massive Multiplayer Online games allow many players to interact using a client-server model. The virtual world, as well as the player's characters, is managed by a central server. In many games, there can be many servers that maintain separate virtual worlds. However, a single server's scalability is limited to tens of thousands of users (even for commercial versions of the game). MMO game servers support a considerably higher number of users than multiplayer versions of other, more interactive games (such as first-person shooters). This improved scalability is achieved by limiting the scope and type of interactivity of the game [3], and by dividing the large number of players into separate game groups (sessions) with maintainable number of players each. This is reasonable for types of games where the virtual world can be divided in multiple parallel sub-levels between which players move for instance on promotion-based manner (Warcraft III, Quake). Despite this limitation, MMO games are very popular and offer an attractive gaming experience. The most prominent examples of MMO RPG games include: EverQuest, Ultima Online, There.com, Star Wars Galaxies, The Sims Online or Warcraft III.

6.2 P2P MMO Games

Several multi-player games (MiMaze, Age of Empires) have already been implemented using the P2P model. However, the scalability of such approaches is in question, as the game state is broadcasted between all players of the game. AMaze [5] is an example of an improved P2P game design, where the game state is multicast only to nearby players. Still in both cases, only the issue of public state maintenance has been addressed. The questions how to deal with the private and public concealed states have not been answered (see section 4).

The authors of [4] have proposed a method of private state maintenance that is similar to ours. They propose the use of commitments and of a trusted "observer", who verifies the game online or at the end of the game. However, the authors of [4] have not considered the problem of concealed or conditional state. Therefore, their trust management architecture is incomplete. Also, the solution proposed in [4] did not address games implemented in the P2P model. In [25], the subject of fair ordering of public state updates has been considered that makes it impossible for players to cheat, while taking into account different network communication delays. This method can be used in P2P games that use peers to manage public objects, such as in the design of [3].

The paper on P2P support for MMO games [3] offers an interesting perspective on implementing MMO games using the P2P model. The presented approach addresses mostly performance and availability issues, while leaving many security and trust issues open. In this paper, we discuss protocols that can be applied to considerably improve the design of Knutsson in terms of security and trust management. In [24], a P2P approach to MMO games has also been proposed, again without taking into account issues of security and trust management.

6.3 Trust Management Using Reputation

One of the most common forms of trust management is the use of agent reputations. Among many applications of this approach, the most prominent are on-line auctions (Allegro, E-Bay). However, P2P file sharing networks such as Kazaa, Mojo Nation, Freenet also use reputation. Reputation systems have been widely researched in the context of multi-agent programming,, social networks, and evolutionary games [20,21,22,23].

Reputation-based mechanisms could be used in P2P games. A player would receive a reputation based on a history of previous games, and this reputation could be used to exclude cheating players from a game. The reason for the use of reputation mechanisms in many networked applications is the lack of enforcement mechanisms that could be used to provide trust management, as noted in [20].

Any reputation system has certain systematic drawbacks that are a reason why it may be worth avoiding to rely on reputation systems in P2P games. Among these, the most important is the problem of first-time cheating. Any peer may build up a high reputation and then cheat in order to win in a game (for example, behave fairly in unimportant encounters, and cheat during a critical encounter).

6.4 Cryptographic Research

Cryptography has considered the problems of fair agreements, games, and multi-party computations. There has been research on the problem of "mental poker", or fair implementation of a distributed poker game [26]. The protocol allows for fair drawing of cards in a poker game, however, it assumes that the game players do not leave the game and is therefore unsuitable for P2P applications. Protocols on fair agreement using a third party are utilized in our research. Multi-party computation allows to calculate an algebraic function of inputs supplied by several parties without revealing the inputs and without centralized control. The research so far in this area requires that the function should be expressed as a Boolean circuit. Applications of multi-party computation in our research are a problem of future work.

7 Conclusion

Many applications that could benefit from the P2P computing model cannot use it because of concerns of security and fairness. In this paper, we have attempted to show how a very sensitive application (a P2P Massive Multiplayer Online game) may be protected from unfair user behavior. However, let us note that while our research focused on P2P MMO games as a challenging application, the developed trust management protocols could be applied in other P2P applications. An example could be P2P auctions that have many properties similar to the discussed conditional or concealed game state in MMO games. Future work may consider applying trust enforcement mechanisms in P2P e-commerce applications.

In our proposed trust management architecture, we have been forced to abandon the pure peer-to-peer approach for a hybrid approach (or an approach with superpeers). However, we have attempted to minimize the role of the centralized trusted components. The result is a system that, in our opinion, preserves much of the performance benefits of the P2P approach, as exemplified by the P2P platform for MMO games proposed in [3]. At the same time, it is much more secure than the basic P2P platform.

We see trust as yet another security functionality (such as privacy or authentication) that can be provided by known cryptographic primitives. The approach that we have tried to use for trust management in peer-to-peer games is “trust enforcement”. It considerably different from previous work on trust management in P2P computing, that has usually relied on reputation. However, reputation systems are vulnerable to first time cheating, and are difficult to use in P2P computing because peers have to compute reputation on the basis of incomplete information (unless the reputation is maintained by superpeers). Instead, we have attempted to use cryptographic primitives to assure a detection of unfair behavior and to enable trust.

The mechanisms that form the proposed trust management architecture work on a periodic or irregular basis (like periodic verification of private players by the coordinator or Byzantine agreement after a veto). Also, the possibility of cheating is not excluded, but rather the trust enforcement mechanisms aim to detect cheating and punish the cheating player by excluding him from the game. In some cases, cheating may still not be detected (if the verification, as proposed, is done on a random basis); however, we believe that the existence of trust enforcement mechanisms may be sufficient to deter players from cheating and to enable trust, like (usually) in the real-world case of law enforcement.

References

1. A.Wierzbicki, T.Kucharski: “P2P Scrabble. Can P2P games commence?”, Fourth International IEEE Conference on Peer-to-Peer Computing , Zurich, August 2004, pp. 100-107
2. A. Wierzbicki, T. Kucharski, Fair and Scalable P2P Games of Turns, Eleventh International Conference on Parallel and Distributed Systems (ICPADS'05), Fukuoka, Japan, pp. 250-256, 2005
3. B. Knutsson, Honghui Lu, Wei Xu, B. Hopkins, Peer-to-Peer Support for Massively Multiplayer Games, IEEE INFOCOM 2004

4. N. E. Baughman, B. Levine, Cheat-proof playout for centralized and distributed online games, INFOCOM 2001, pp 104-113
5. E.J. Berglund and D.R. Cheriton. Amaze: A multiplayer computer game. IEEE Software, 2(1), 1985.
6. Butterfly.net, Inc., The butterfly grid: A distributed platform for online games, 2003, www.butterfly.net/platform/
7. N. Sheldon, E. Girard, S. Borg, M. Claypool and E. Agu, The effect of latency on user performance in Warcraft III, 2nd Workshop on Network and System Support for Games (NetGames 2003), pp 3-14
8. J. Menezes, P. C. van Oorschot, S. A. Vanstone, Handbook of applied cryptography, CRC Press, ISBN: 0-8493-8523-7, October 1996
9. L. Lamport, R. Shostak, M. Pease, Byzantine Generals Problem, ACM Trans. on Programming Laguages and Systems, 1982, 4, 3, pp 382-401
10. Erik Warren Selberg, How to Stop A Cheater: Secret Sharing with Dishonest Participation, Carnegie Mellon University, 1993
11. Martin Tompa and Heather Woll, How to share a secret with cheaters, Research Report RC 11840, IBM Research Division, 1986
12. Wierzbicki, R. Strzelecki, D. Świerczewski, M. Znojek (2002), Rhubarb: a Tool for Developing Scalable and Secure Peer-to-Peer Applications, Second IEEE Int. Conf. Peer-to-Peer Computing, P2P2002
13. J. Douceur, The Sybil Attack, In Proc. of the IPTPS02 Workshop, Cambridge, MA (USA), March 2002
14. M. Burmester, Y. Desmedt, Is hierarchical Public-Key Certification the next target for hackesr?, ACM Communications, August 2004, vol. 47, no. 8
15. D.Agrawal et al., An Integrated Solution for Secure Group Communication in Wide-Area Netwokrs, Proc. 6th IEEE Symposium on Comp. and Comm., July, 2001, pp 22-28
16. IETF, SPKI Working group, http://www.ietf.org
17. Zona Inc. Terazona: Zona application framework white paper, 2002, www.zona.net/whitepaper/Zonawhitepaper.pdf
18. H. Yoshino et al., Byzantine Agreement Protocol Using Hierarchical Groups, Proc. International Conference on Parallel and Distributed Systems, 2005
19. Hoang Nam Nguyen1, Hiroaki Morino, A Key Management Scheme for Mobile Ad Hoc Networks Based on Threshold Cryptography or Providing Fast Authentication and Low Signaling Load, T. Enokido et al. (Eds.): EUC Workshops 2005, LNCS 3823, pp. 905 – 915, 2005
20. L. Mui, Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks, Ph.D. Dissertation, Massachusetts Institute of Technology, 2003
21. K.Aberer, Z.Despotovic (2001), Managing Trust in a Peer-To-Peer Information System, Proc. tenth int. conf. Information and knowledge management, 310-317
22. B.Yu, M.Singh (2002), An Evidential Model of Distributed Reputation Management, Proc. first int. joint conf. Autonomous agents and multiagent sys., part 1, 294-301
23. P. Gmytrasiewicz, E. Durfee, Toward a theory of honesty and trust among communicating autonomous agents Group Decision and Negotiation 1993. 2:237-258
24. T. Iimura, H. Hazeyama and Y. Kadobayashi, Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multi-player Online Games, Proc. ACM SIGCOMM, 2004
25. B. Chen and M. Maheswaran, Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multi-player Online Games, Proc. ACM SIGCOMM, 2004
26. W. Zhao, V. Varandharajan and Y. Mu, A Secure Mental Poker Protocol Over The Internet, Proc. Australasian Information Security Workshop (AISW2003), 2003

A P2P-Based System to Perform Coordinated Inspections in Nuclear Power Plants

C. Alcaide, M. Díaz, L. Llopis, A. Márquez, and E. Soler

Dpto. Lenguajes y Ciencias de la Computación, Universidad de Málaga
29071 Málaga, Spain
{calcaide, mdr, luisll, amarquez, esc}@lcc.uma.es

Abstract. TEDDY applications form a distributed package designed for data acquisition and evaluation of inspections carried out in nuclear power plants. These inspections typically involve a number of acquisition equipments exploring the state of different parts of the plant in order to detect any kind of failure. Currently, these applications follow a typical client-server model, where every client –up to one hundred– accesses a central server containing all the information about the inspection itself; furthermore, it centralizes all the coordination and synchronization mechanisms of the work. The client applications act as producer/consumer from server data, so that a failure in the server should halt all the activity in the inspection. In this paper, we propose a peer to peer model where all the nodes share the information and discuss the pros and cons of the new system.

Keywords: Peer-to-peer, resource sharing, distributed applications, failure tolerance, replication, collaboration technologies.

1 Introduction

A nuclear power plant (NPP) is a thermal power station in which the heat source is one or more nuclear reactors generating nuclear power. Nowadays, they provide about 17 percent of the world's electricity.

The inspection of a nuclear power plant typically involves a number of acquisition equipments exploring the state of different parts of the plant in order to find any kind of defect. There are two types of inspections, the so called Pre Service Inspections (PSI) and the periodical inspections that are generally performed during the fuel recharge (also called In Service Inspection or ISI). Although the scope of an inspection includes all the areas of a nuclear power plant, (reactor vessels, pipes and turbines), this paper is concerned with automated eddy current inspections of steam generators and heat exchanger tubes [12].

The steam generator is one of the most important components of a nuclear power plant; its function is to transfer the heat from the reactor cooling system to the secondary side of the tubes which contain feed water. As the feed water passes through the tube, it picks up heat and is eventually converted to steam. Each NPP generally consists of three steam generators each one containing up to 15,000 tubes which are reviewed periodically to avoid escapes.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1181 – 1190, 2006.
© Springer-Verlag Berlin Heidelberg 2006

A steam generator inspection consist of the detection, characterization and dimensioning of loss of tube wall thickness, erosion, cracking, etc, with a view to increasing the safety and availability of the component. This check is performed using multi-frequency, digital, robot-operated, remote control equipment. In order to reduce the plant stoppage time, the inspection is carried out in turns by acquisition operators/analyst teams during the 24 hours of the day until it is completed, performing the following tasks:

- **Acquisition:** These activities are carried out in difficult to access areas with high levels of radiation and contamination, which makes it necessary to use remotely-controlled robot-operated equipment. In a PSI, data from all the tubes must be acquired and analyzed although in an ISI only part of them is inspected. The information obtained in this phase is called *raw data* and it is classified in the so called *calibrations* (set of acquired tubes).
- **Analysis:** The raw data must be studied by qualified staff using specific applications in order to detect any kind of anomaly. Due to the critical importance of the inspection, each acquired tube must be analyzed by several analysts. In a typical inspection there can be up to four analysts per tube, each one playing a different role. These roles are called primary, secondary, etc. It is important to point out that the analysis phase begins once the first tube is acquired so that analysis and acquisition are performed simultaneously in a pipeline manner.
- **Resolution:** The analyzed data for each tube in the different roles must be compared in order to detect discrepancies between the analyses. A resolution analyst must hold the highest qualification and he/she is responsible for the final report for each tube.

To carry out these activities the operator/analyst teams use applications based on a client/server architecture where all the inspection information is stored in a central server. This architecture presents some problems because an eventual failure in the server could stop all the analyst activities.

In order to solve this problem, we propose to solve this architecture to a peer-to-peer (P2P) approach [8] where the information is distributed in all network nodes.

P2P systems that dynamically organize, interact, and share resources are increasingly being deployed in large-scale environments. These present the evolution of the client-server model that was primarily used in small-scale distributed environments to accommodate a vast number of users. The most distinct characteristic of the P2P model is that there is a symmetric communication between the peers; each peer has both a client and a server role. The advantages of P2P systems are multidimensional:

- They improve scalability by enabling direct and real-time sharing of resources and data.
- They enable knowledge sharing by aggregating information and resources from nodes located on geographically distributed and potentially heterogeneous platforms.
- They provide high availability by eliminating the need for a single centralized manager.

P2P systems emphasize heterogeneity in computing capabilities, communication bandwidths, and connection times among peers. Thus, P2P systems have the following important properties:

- They can be deployed over heterogeneous operating systems, networks and platforms. Peer-to-peer applications can run on multiple devices ranging from powerful workstations to laptops and wireless PDAs.
- Peers are autonomous and highly dynamic, and have various join and disconnect cycles. Furthermore, each peer has only a limited view of the state and topology of the network.
- Resources and applications are highly replicated and available at multiple peers in the system.

Nowadays, there are some very popular systems which were designed to share CPU (Seti@home [13], XtremWeb [14], Entropia [4]) or to publish files (Napster, Gnutella, Kazaa). At the same time, some systems were designed to share disk space (OceanStore [5], Intermemory [15], PAST [10], Farsita [1]). The main goal of such systems is to provide a transparent distributed storage service.

The structure of the paper is as follows. In the following section, an overall description of the current centralized approach of TEDDY applications is presented. Section 3 describes the P2P approach and the benefits contributed by this model. Section 4 explains the system architecture that we propose. Finally, some conclusions are presented in section 5.

2 Client/Server Approach

In a client-server model, we have up to one hundred clients (operators and analysts) sharing their information using a central database server where the inspection plan data has been previously published.

The inspection plan is a list of a steam generator tubes that must be acquired by the acquisition operators and must be analyzed in different roles by the analysts.

Each client gets the inspection plan from the central database and, if necessary, receives data previously stored by another client (for example, if the user is going to supervise an analysis). This model could be a natural way to develop this type of system: data acquisition and evaluation applications generate a lot of information and store it in a structured way in order to perform different types of queries, to manage reports easily and to keep this important information to study the state of the nuclear plant and to facility future inspections.

Since a failure in the central server would stop all the analyst activities, a local replica of the database is created. This replica contains the data related to the set of tubes to be analyzed. This way, an analyst can continue working in his local database if the central database is down. When the server is up again, the analyst uploads his local copy to the central server. Since each analyst inspects a tube in a different role, there is no conflict with this algorithm. The resolution analyst must access the central database in order to compare the analyses performed by different analysts for the same tube.

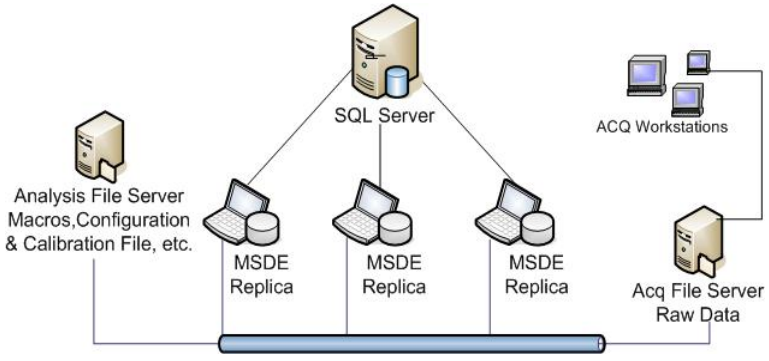


Fig. 1. Centralized (Client-Server) approach

Figure 1 shows that two file servers and one database system are used. On the one hand, the Acquisition File Server has all the raw data with the information on the tubes and on the other hand, the Analysis File Server contains files used by the analysts to determine whether there are defects in the tubes or not. Furthermore, the central server (SQL Server) has the inspection data obtained by each analyst locally.

However, this approach has the following disadvantages:

- The resolution phase depends on the availability of the central server. Furthermore, if an analyst finishes his current work, he cannot obtain more calibrations until the server is up and running again.
- A failure in a central server that stops the analyst's activity is unacceptable: Inspection time is critical because the NPP is stopped while the inspection is taking place.
- Higher costs are incurred due to the price of the server and the SQL Server license.
- Scalability is less, because the response times of the server increase when there are more analysts working.

3 P2P Approach

At present, TEDDY follows a model as commented above, and our research group is now improving this set of applications, updating its mechanisms to make these inspections more efficient. As mentioned previously, this model introduces some problems related to system weakness in the case of a failure in the server or in the network.

A first approach to solving this problem was to make a local copy of the work data in the analyst's machine and then update the central database when the work is finished in order to reduce server dependency and give more autonomy to the analyst's work (Fig. 2 left). This first attempt is now running correctly, but does not solve the problem completely; it is only is a mechanism to allow the analysts to continue working during short-time failures on the server. Therefore, we started to see P2P systems as a good model to be deployed in future inspections (Figure 2 right).

Therefore, we propose a P2P-based system, where collaboration and content sharing are taken into account.

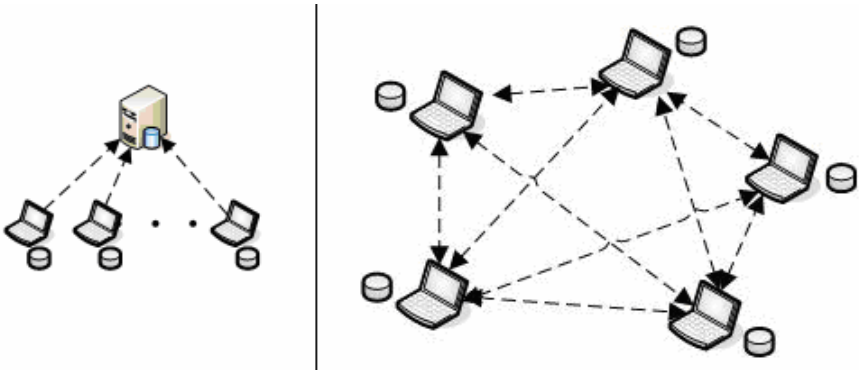


Fig. 2. A system to replicate the server data in the analyst's computers (left). Dynamically organized peer network, where every peer interacts with the others and shares their resources (right).

3.1 Benefits

When using P2P technology not only it is solved the problem of a server failure but other very interesting properties appear:

- **Cost reduction.** The client-server model, placing a strong database system on the server side and using a redundant server where information is saved in the case of a server failure may be expensive. Normally, licenses for big database systems are expensive, and several licenses are needed. Furthermore, although hardware performance and costs have improved, centralized repositories are expensive to set up and difficult to maintain. In general, they require human intelligence to build and to keep the information they contain relevant and current. In addition, the deployment of the client/server architecture with a powerful server may be slow and, once the inspection is finished, it must be removed until the next inspection.
- **Efficiency.** The topology of centralized systems inevitably yields inefficiencies, bottlenecks, and wasted resources.
- **Increased autonomy.** In many cases, users of a distributed system are unwilling to rely on any centralized service provider.
- **Scalability.** An immediate benefit of decentralization is improved scalability.
- **Dynamism.** P2P systems assume that the computing environment is highly dynamic. That is, resources, such as compute nodes, will be continuously entering and leaving the system. This is particularly important in our application since analysts and operators work in turns and enter and leave the system several times during the inspection. When an application is intended to support a highly dynamic environment, the P2P approach is a natural choice.
- **Replication.** Replication puts copies of object/files closer to the requesting peers, thus minimizing the connection distance between peers requesting and providing the objects. In our case, the data replication process is described in the next section. These pieces of data are replicated in most of the machines, solving the bottleneck problem that exists in centralized systems.

4 System Architecture

Figure 3 shows the proposed architecture over an existing P2P framework such as Windows Peer-to-Peer Networking [3]. Each client machine in the system becomes a peer in the network, and every peer acts both as a server offering its information to others, and as a client receiving information from them.

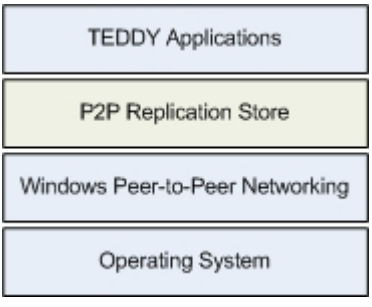


Fig. 3. The P2P Replication Store is the proposed intermediate layer between the final applications and the Windows Peer-to-Peer Networking API

4.1 Data Partitioning

Inspection data must be broken down into smaller pieces of information in order to transmit it among the peers in the network. TEDDY users do not need all the inspection information in the network to perform a specific acquisition, analysis or

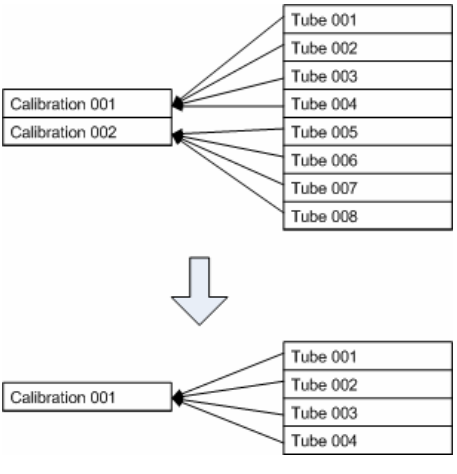


Fig. 4. Horizontal data partition. Any analysis operation is performed on a single calibration; therefore a natural horizontal division is separate tubes from different calibrations in different pieces.

resolution; they need only one piece of information to do their work. For example, if the user wants to complete an acquisition, he/she needs only the inspection base data (component information and model geometry) and the work plan to follow in order to acquire the tubes previously planned. Therefore, he/she does not need the other work plans in the system or any analysis information.

It is important to say that the data model of these applications (imposed for security reasons) is developed in such a way that every update made by one user does not change the previous existing information in the system. This update is translated into new data which can be merged with the existing information.

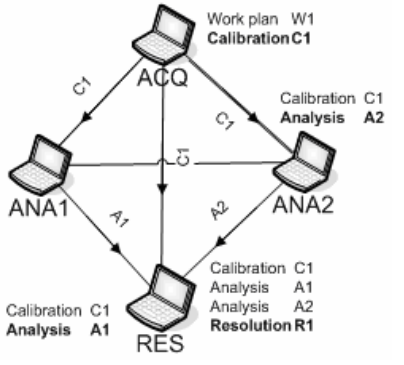


Fig. 5. The pieces of data are propagated through the peers as they require

In order to provide an agile way of sharing the information in the network, the inspection information is divided into autonomous pieces of data (Figure 4), which can be easily shared between the peers which require them (Figure 5).

1. The acquisition step generates a new piece of information witch the tubes acquired (raw data).
2. This data is propagated to other peers who use it to generate new data with the analysis information.
3. The analysts perform different analysis using de acquisition data, and publish its results.
4. The resolution gets all the previous data published, merges it, and generates new results.
5. At the end of the resolution step, the results are published as a new data item.

4.2 Registers and Publishing

At this point, we know that the inspection data can be divided into small pieces of information and in order to perform a specific job, the application only needs several pieces of data. When an analyst is going to do a job, the system ascertains whether the necessary data is available in the local machine. If some information is not local, then it must be copied from another peer which stores it.

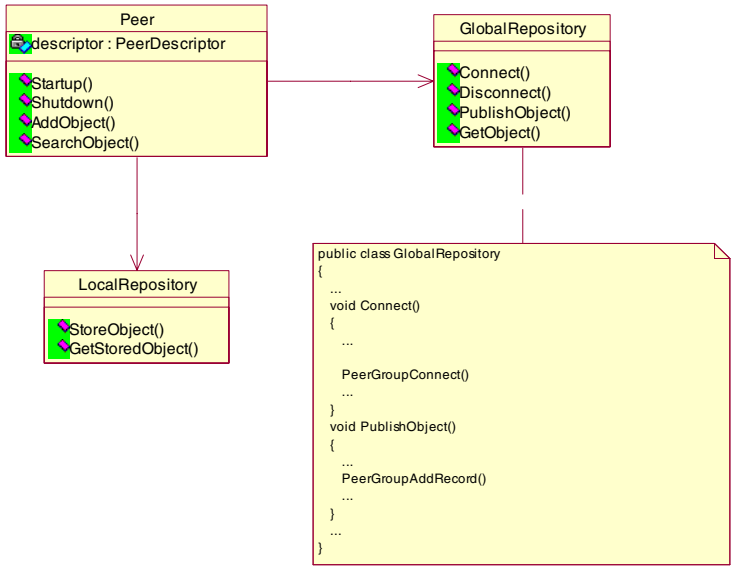


Fig. 6. The middleware layer is developed by calling to Windows Peer-to-Peer Networking API functions and handling their events

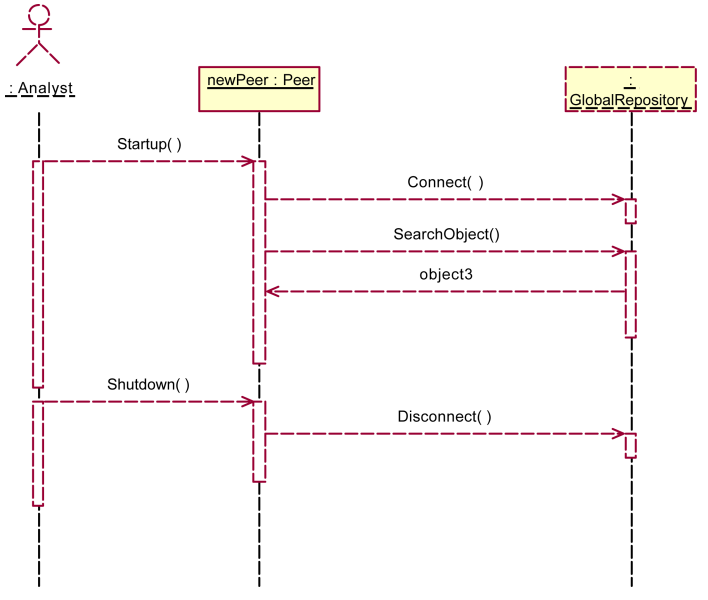


Fig. 7. Data publishing. Each peer publishes its local data in the network to let other peers know the resources it is sharing.

The main question here is: what are their identities and what information is kept by each one locally? We delegate all peer-to-peer paradigm related questions (communications, name resolution, peer discovery, etc.) over the Windows Peer-to-Peer Networking Framework (Figure 6). This framework lets us see the network as a large shared repository in which a peer can publish the content of the information it stores, handle events to get new registers added by other peers, search for a specific register, etc.

A register is a special structure that peers publish in the P2P network in order to let other peers know that they store a particular piece of information which may be important to others in an analysis or resolution phase (Figure 7). In this way, when clients start to work, they publish all their information in the network.

It is important to print out that a published record does not contain the actual information, only the peer identity and the information identity, i.e., the basic information to know what type of information the peer holds, for example the primary analysis of the calibration 001, and the peer identity in order to establish direct connections between peers as data streams.

5 Conclusions

In this paper, we have described a system developed to carry out such a critical work as the inspection of a nuclear power plant. The typical client/server architecture presents some problems related to robustness, cost, and the need for a computer engineer to be present in the plant during the inspection to solve a failure of the system should such a circumstance arise. A highly replicated system may become unaffordable since the inspection only lasts several days and the software and hardware system must be uninstalled once the inspection is finished. A peer-to-peer approach to overcome these problems is proposed. Currently, we are implementing the middleware layer with the most important methods: Connections, record publication and data transfer between peers in such a way that the already existing applications can be smoothly updated in order to work with the middleware instead of the centralized database system.

In our future work, we will have to solve some security problems and to implement algorithms that improve the operation of the system (for example, related to load balancing).

Acknowledgments. We would like to acknowledge the help from the people of Tecnatom S.A. who contributed to the creation of this paper.

References

1. Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer., R.: Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.
2. Barkai, D.: Technologies for Sharing and Collaborating on the Net. First International Conference on Peer-to-Peer Computing (2001) 13-28.

3. Davies, J., Manion, T., Rao, R., Miller, J., Zhang, X.: Introduction to Windows Peer-to-Peer Networking (2003).
4. Entropia Web site: http://www.entropia.com
5. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao., B., Oceanstore: An architecture for global-scale persistent storage. In Proceedings ofACM ASPLOS. ACM, November 2000.
6. Kortuem, G., Schneider, J., Preuitt, D., Thompson, T., Fickas, S., Segall, Z.: When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad hoc Networks. First International Conference on Peer-to-Peer Computing (2001) 75-91.
7. Kalogeraki, V., Chen, F.: IEEE Volume 18, Issue 1 (Jan-Feb 2004) 22-29.
8. Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard., B., Rollins, S., Xu, Z.: Peer-to-Peer Computing. HP (2002).
9. Ji, L., Deters, R.: Coordination & Enterprise Wide P2P Computing. IEEE International Conference on Services Computing (2005).
10. Druschel, P., Rowstron, A.: PAST: A large-scale, persistent peer-to-peer storage utility. In Procedings of HOTOS, pages 75-80, 2001.
11. Serrano, J., Ravier, D., Fraga, J.; Orjales, V., Molano, A.: GEODAS: An Industrial Experience with Component Frameworks for Data Acquisition and Analysis Systems (2001) 72-92.
12. Tecnatom Web site: http://www.tecnatom.es
13. SETI@home Web site: http://setiathome.ssl.berkelev.edu/
14. XtremWeb Web site: http://www.lri.fr/~fedak/XtremWeb/
15. Chen, Y., Edler, J., Goldberg, A., Gottlieb, A., Sobti, S., Yianilos, P.: A prototype implementation of archival intermemory. In Proceedings of the Fourth ACM International Conference on Digital Libraries, 1999.

Grid File Transfer During Deployment, Execution, and Retrieval

Françoise Baude, Denis Caromel, Mario Leyton, and Romain Quilici

INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France
First.Last@sophia.inria.fr

Abstract. We propose a file transfer approach for the Grid. We have identified that file transfer in the Grid can take place at three different stages: deployment, user application execution, and retrieval (post-execution). Each stage has different environmental requirements, and therefore we apply different techniques. Our contribution comes from: (*i*) integrating heterogeneous Grid resource acquisition protocols and file transfer protocols including deployment and retrieval, and (*ii*) providing an asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation.

We validate and benchmark the proposed file transfer model using ProActive, a Grid programming middleware. ProActive provides, among others, a Grid infrastructure abstraction using deployment descriptors, and an active object model using transparent futures.

1 Introduction

Scientific and engineering applications that require, handle, and generate large amount of data represent an increasing use of Grid computing. To handle this large amount of information, file transfer operations have a significant importance. For example, some of the areas that require handling large amount of data in the Grid are: bioinformatics, high-energy physics, astronomy, etc.

Although file transfer utilities are well established, when dealing with the Grid, environmental conditions require reviewing our previous understanding of file transfer to fit new constraints and provide new features at three different stages of Grid usage: deployment, execution, and post-execution. At deployment time, we focus on integrating heterogeneous file transfer and resource acquisition protocols to allow *on-the-fly* deployment. During the application run time, we offer a parallel and asynchronous file transfer mechanism based on active objects, wait-by-necessity, and automatic continuation. Once the user application has finished executing, we offer a file retrieval mechanism.

This document is organized as follows. In section 2 we provide some background on the Grid programming middleware ProActive. In sections 3 and 4 we describe our file transfer proposal for the Grid, and show how this is implemented in the context of ProActive. We benchmark the implementation of the model in section 5. Related work is reviewed in section 6, and finally we conclude in section 7.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1191–1202, 2006.
© Springer-Verlag Berlin Heidelberg 2006

2 Background on ProActive

Figure 1, shows the *active object* (AO) programming model used in ProActive[16]. AO are remotely accessible via method invocations, automatically stored in a queue of pending requests. Each AO has its own thread of control and is granted the ability to decide in which order incoming method calls are served (FIFO by default). Method calls on AO are asynchronous with automatic synchronization (including a rendezvous). This is achieved using automatic *future objects* as a result of remote methods calls, and synchronization is handled by a mechanism known as *wait-by-necessity* [4].

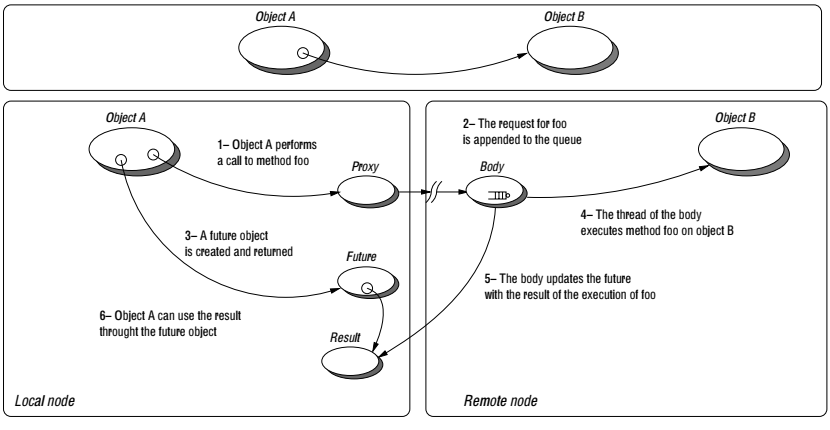


Fig. 1. Execution of a remote method call

ProActive also provides a *Descriptor Deployment Model* [3], which allows the deployment of applications on sites using heterogeneous protocols, without changing the application source code. All information related with the deployment of the application is described in the XML Deployment Descriptor. Thus, eliminating references inside the code to: machine names, resource acquisition protocols (local, rsh, ssh, lsf, globus-gram, unicore, pbs, lsf, nordugrid-arc, etc..) and communication protocols (rmi, jini, http, etc...).

The Descriptor Deployment Model is shown in Figure 2. The infrastructure section contains the information necessary for booking remote resources. Once booked, ProActive Nodes can be created (or acquired) on the resources. To link the Nodes with the application code, a Virtual Node (VN) abstractions is provided, which corresponds to the actual references in the application code. Virtual Nodes have a unique identifier which is hardcoded inside the application and the descriptor.

A deployer can change the mapping of the application → Virtual Node to deploy on a different Grid, without modifying a single line of code in the application.

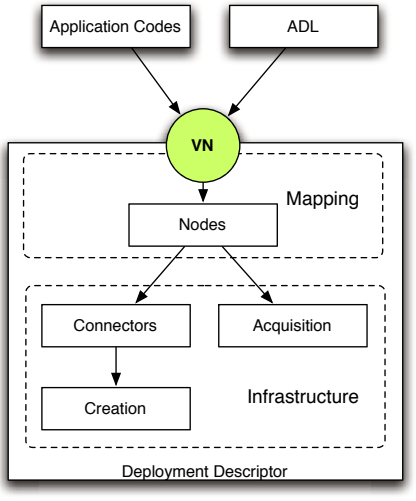


Fig. 2. Descriptor Deployment Model

3 Grid Deployment and File Transfer

3.1 On-the-Fly Deployment

We consider that deployment on the Grid represents the fulfillment of the following tasks: (*i*) Grid infrastructure setup (protocol configuration, installation of Grid middleware libraries, etc...), (*ii*) resource acquisition (job submission), (*iii*) application specific setup (installing application code, input files, etc...), and (*iv*) application deployment (setting up the logic of the application).

Usually, the deployment requires files transfer during the above cited tasks to succeed, for such files as: Grid middleware libraries (*i*), application code (*iii*), and application input files (*iv*). We say a Grid deployment can be achieved **on-the-fly** if the required files can be transferred when deploying, without having to install them *in advance*. It is our belief, that on-the-fly deployment greatly reduces the Grid infrastructure configuration, maintenance and usage effort.

In the rest of this section, we describe how heterogeneous protocols for file transfer and resource acquisition can be integrated to achieve on-the-fly deployment for the Grid. To explain the approach, we first introduce the notation and review some general concepts concerning resource acquisition and file transfer.

3.2 Concepts

Let r be a resource acquisition protocol, t a file transfer protocol, n a Grid node, p a Grid infrastructure parameter, and f a file definition. We say a node n_k is acquirable from n_0 iff $\exists\{r_0(p_0), \ldots, r_{k-1}(p_{k-1})\}$ and $\exists\{n_0, \ldots, n_{k-1}\}$ as shown in Figure 3(a). The nodes are acquired sequentially one after the other, i.e. n_k is acquired before n_{k+1} using a resource acquisition protocol r_k.

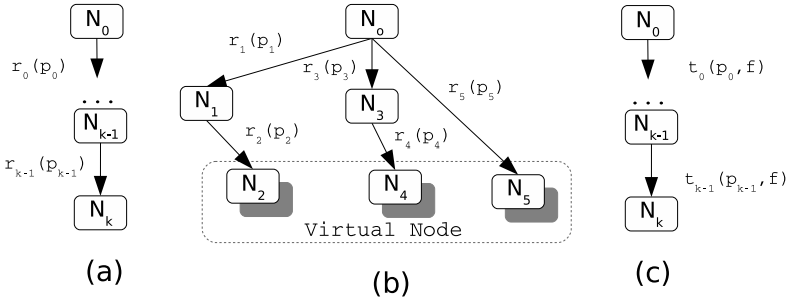


Fig. 3. Resource Acquisition and File Transfer

A Grid infrastructure resource acquisition can more precisely be seen as a tree, since more than one node can be acquired in parallel. As shown in Figure 3(b), the leaf nodes represent the acquired resources[1], and we will call them `virtualNode`, using the ProActive terminology.

Given a file transfer protocol t we say a file f can be transferred from n_0 to n_k iff $\exists\{t_0(p_0, f), \ldots, t_{k-1}(p_{k-1}, f)\}$ and $\exists\{n_0, \ldots, n_{k-1}\}$ (Figure 3(c)).

A file transfer protocol can be of two types: internal if the file transfer protocol is executed by the resource acquisition protocol, i.e. $r(p, f)$ executes the file transfer and performs the resource acquisition (`unicore`, `nordugrid`); or external if they are not part of a resource acquisition protocol (`scp`, `rcp`). Therefore, internal file transfer protocols can not be used separately from the corresponding resource acquisition protocol.

3.3 Integration Proposal

Supposing that n_{k+1} is acquirable from n_k using r_k, and given an ordered list of file transfer protocols $\overrightarrow{t_k}$ that can or cannot be successful at transferring f from n_k to n_{k+1}. Then, if there $\exists t_k^i \in \overrightarrow{t_k}$ which corresponds to the lower indexed transfer protocol capable of transferring f, we propose the sequencing of file transfer and resource acquisition protocols in the following way:

1. If t_k^i is external, then we will execute

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^i(p,f),r_k(p)} n_{k+1}$$

   That is to say, that the file transfer protocols will be executed sequentially until one of them succeeds, and then the resource acquisition protocol will be executed.
2. If t_k^i is an internal file transfer protocol of r_k, then we will execute:

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^{i-1}(p,f),r_k(p,f)} n_{k+1}$$

[1] Depending on the deployment mechanism, sometimes the internal nodes also represent acquired resources.

The assumption is that the internal t_k^i of a given r_k will always succeed. This is reasonable, because if the internal t_k^i fails, this implies that r_k will also fail, and thus there is no point on testing further file transfer protocols.

The problem with the sequencing approach, is that no file transfer protocol $t_k^i \in t_k$ may be successful at transferring f. To solve this, we propose the usage of a *failsafe* file transfer protocol, which is reliable at performing the file transfer, but only after the resource acquisition has taken place. Therefore, if t_k^i is a failsafe protocol, then we will execute:

$$n_k \xrightarrow{t_k^0(p,f),\ldots,t_k^{i-1}(p,f),r_k(p),t_k^i(p,f)} n_{k+1}$$

Note that in the failsafe approach, the actual file transfer is performed after the resource acquisition.

There are two main reasons for trying to avoid using a failsafe protocol. The first one, is that failsafe performs the file transfer at a higher level of abstraction, not taking advantage of lower level infrastructure information, as shown in the benchmarks of section 5.2. The second reason is that *on-the-fly* deployment becomes limited: the libraries required to use the failsafe protocol cannot be transferred using the failsafe protocol, and must be transferred in advance.

3.4 File Transfer in ProActive Deployment Descriptors

Figure 4 shows how the approach is integrated into ProActive XML Deployment Descriptors. We take advantage of the descriptors structure to apply separation of concerns. The actual files requiring file transfer are specified in a different section (`FileTransferDefinitions`) than the Grid infrastructure parameters (`FileTransferDeploy`). The infrastructure parameters holds information such as: the sequence of protocols that will be tried to copy the file (`copyProtocol`)[2], hostnames, usernames, etc. Finally, the `FileTransferRetrieve` tag specifies which files should be retrieved from the nodes in the retrieval (post-execution) phase (reviewed in further depth in section 4.2).

4 File Transfer During Execution and Retrieval

Applications can generate data, and transferring this data during the application execution is usually achieved using a specific communication protocol for transferring the file's data. Nevertheless, Grid resources are characterized by distributed ownership and therefore diverse management policies, as our own experiments [14,15] confirm it. As a result, setting up the Grid to allow message passing is a painfull task. Additionally configuring and maintaining a specific file transfer protocol between any pair of nodes seems to us as an undesirable burden[3].

[2] The *failsafe* protocol shown in the example is described in further detail in section 4.1.

[3] Deployment file transfer does not impose this burden, because the file transfer does not take place between each possible pair of nodes.

```
<FileTransferDefinitions>
   <FileTransfer id="requiredfiles">
      <file src="application.class" dest="application.class"/>
      <file src="ProActive.jar" dest="ProActive.jar"/>
      <file src="input.dat" dest="input.dat"/>
  </FileTransfer>
  <FileTransfer id="results"><file src="output.dat"/></FileTransfer>
<FileTransferDefinitions>
...
<virtualNode name="exampleVNode" FileTransferDeploy="requiredfiles"/>
...
<processDefinition id="xyz">
  <sshProcess>
    <!-- The refid attribute can be set to "implicit", which will use the value defined in
         the VirtualNode. -->
    <FileTransferDeploy refid="implicit">
      <copyProtocol>processDefault, rcp, scp, failsafe</copyProtocol>
      <sourceInfo prefix="/home/user"/>
      <destinationInfo prefix="/tmp" hostname="foo.org" username="smith" />
    </FileTransferDeploy>
    <!-- The refid can also directly reference the FileTransfer id. -->
    <FileTransferRetrieve refid="results">
      <sourceInfo prefix="/tmp"/>
      <destinationInfo prefix="/home/user"/>
    </FileTransferRetrieve>
  </sshProcess>
</processDefinition>
```

Fig. 4. Example of File Transfer in Deployment Descriptor

Therefore, we propose that the file transfer protocol should be built on top of other protocols, specifically the message passing protocols. Standard message passing is not well suited for transferring large amounts of information, mainly because of memory limitations and lack of performance optimizations for large amounts of data. In this section we show how an active object based message passing model can be used as the ground for a portable efficient scalable file transfer service for large files, where large means bigger than available runtime memory. Additionally by using active objects as transport layer for file transfer, we can benefit from the automatic continuation to improve the file transfer between peers, as we will show in the benchmarks of section 5.

4.1 Asynchronous File Transfer with Futures

We have implemented file transfer as service methods available in the ProActive library as shown in Figure 5. Given a ProActive **Node** *node*, a **File**(s) called *source*, and a **File**(s) called *destination*, the *source* can be pushed (*sent*) or pulled (*get*) from *node* using the API. The figure also shows a `retrieveFiles` method, which is discussed in section 4.2.

The *failsafe* algorithm mentioned in section 3.3 is implemented using the `pushFile` API, which is itself built using the *push* algorithm depicted in Figure 6 and detailled as follows:

1. Two File Transfer Service (FTS) active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The *push* function is invoked by the caller on the local FTS: $LocalFTS.push(\ldots)$.

```
//Send file(s) to Node node
static public File pushFile(Node node, File source, File destination);
static public File[] pushFile(Node node, File[] source, File[] destination);

//Get file(s) from Node node
static public File pullFile(Node node, File source, File destination);
static public File[] pullFile(Node node, File[] source, File[] destination);

//Retrieve files specified for the virtualNode
public File[] virtualNode.retrieveFiles();
```

Fig. 5. File Transfer API

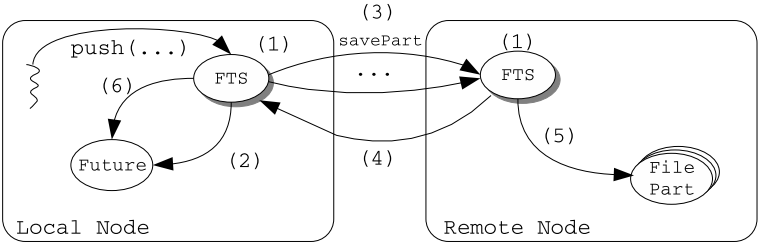


Fig. 6. Push Algorithm

2. The local FTS immediately returns a File future to the caller. The calling thread can thus continue with its execution, and is subject to a wait-by-necessity on the future to determine if the file transfer has been completed.
3. The file is read in parts by the local FTS, and up to $(o-1)$ simultaneous overlapping parts are sent from the local node to the remote node by invoking $RemoteFTS.savePartAsync(p_i)$ from local FTS [2].
4. Then, a $RemoteFTS.savePartSync(p_{i+o})$ invocation is sent to synchronize the parameter burst, as not to drown the remote node. This will make the sender wait until all the parts $p_i, \ldots, p_i + o$ have been served (ie the *savePartSync* method is executed).
5. The *savePartSync*(...) and *savePartAsync*(...) invocations are served in FIFO order by the remote FTS. These methods will take the part p_i and save it on the disk.
6. When all parts have been sent or a failure is detected, local FTS will update the future created in step 2.

The `pullFile` method is implemented using the *pull algorithm* shown in Figure 7, and is detailled as follows:

1. Two FTS active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The *pull* function is invoked on the local FTS: $LocalFTS.pull(\ldots)$.
2. The local FTS immediately returns a File future, which corresponds to the requested file. The calling thread can thus continue with its execution and is subject to a wait-by-necessity on the future.

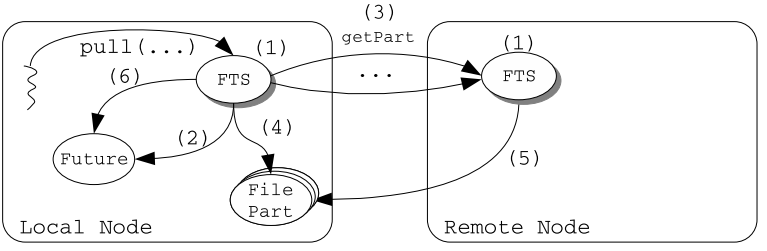


Fig. 7. Pull Algorithm

3. The $getPart(i)$ method is invoked up to o (internally defined) times, by invoking $RemoteFTS.getPart(i)$ from the local FTS [2].
4. The local FTS will immediately create a future *file part* for every invoked $getPart(i)$.
5. The $getPart(...)$ invocations are served in FIFO order by the remote FTS. The function $getPart$ consists on reading the file part on the remote node, and as such, automatically updating the local futures created in step 4.
6. When all parts have been transferred, then the local FTS will update the future created in step 2.

4.2 File Transfer After Application Execution

Collecting the results of a Grid computation distributed in files on different nodes is an indispensable task. Since determining the termination of a distributed application is hard and sometimes impossible, we believe that the best way is to have non-automatic file retrieval, meaning that it is the user's responsability to trigger the file transfer at the end of the application execution (i.e once the application data has been produced).

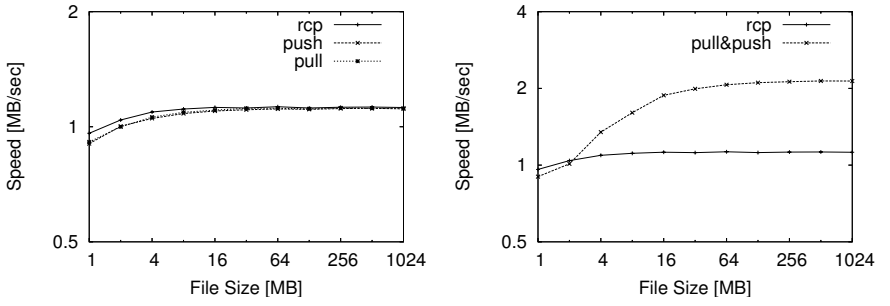
The file transfer retrieval is implemented as part of the API shown in Figure 5. For each node in the `virtualNode`, a `pullFile` is invoked, and an array of futures (`(File[]`) is returned. The retrieved files are the ones specified in the deployment descriptor, as shown in Figure 4.

5 Benchmarks

5.1 File Transfer Push and Pull

Using a 100Mbit LAN network with a $0.25[ms]$ ping, and our laboratory desktop computers: Intel Pentium 4 (3.60GHz) machines, we experimentally determined that overlapping 8 parts of size $256[KB]$ provides a good performance and guarantees that at the most $2[MB]$ will be enqueued in the remote node. The communication protocol between active object was configured to `RMI`.

Since peers usually have independant download and upload channels, the network was configured at $10[\frac{Mbits}{sec}]$ duplex. Figure 8(a) shows the performance results of `pull`, `push`, and *remote copy protocol* (`rcp`) for different file sizes. The performace achieved by `pull` and `push` approaches our ideal reference: `rcp`.



(a) Push, pull, and rcp speed (b) Pushing while pulling a file

Fig. 8. Performance comparisons

More interestingly, Figure 8(b) shows the performance for getting a file from a remote site, and then sending this file to a new site. This corresponds to a recurrent scenario in data sharing peer to peer networks[11], where a file can be obtained from a peer instead of the original source.

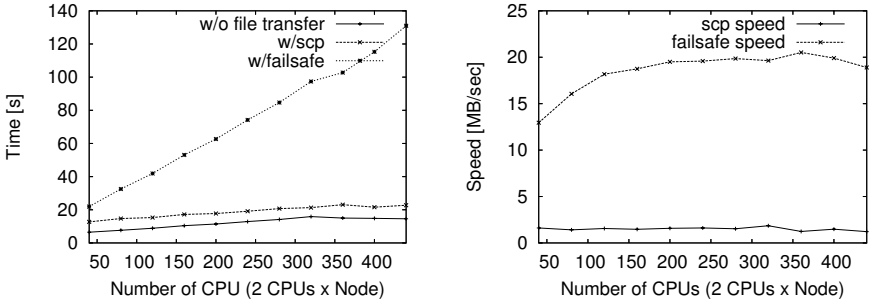
As we can see in Figure 8(b), `rcp` is outperformed when using `pull` and `push` algorithms. While `rcp` must wait for the complete file to arrive before sending it to a peer, the `pull` algorithm can pass the future file parts (Figure 7) to the `pull` algorithm even before the actual data is received. When the future of the file parts are updated, automatic continuation [5,6] will take care of updating the parts to the concerned peers. The user can achieve this with the API shown in Figure 5, by passing the result of an invocation as parameter to another.

5.2 Deployment with File Transfer on a Grid

Our deployment experiments took place on the large scale national french wide infrastructure for Grid research: *Grid5000* [8], gathering 9 sites geographically distributed over France.

Figure 9(a) shows the time for three different deployment configurations combined with a transfer of a 10[MB] file: regular deployment without involving file transfer, deployment combined with (`scp`), and deployment combined with the failsafe file transfer protocol (which uses the `push` algorithm). The figure shows that combining deployment with `scp` adds a constant overhead, while failsafe adds a linear overhead. This happens, because the nodes in Grid5000 are divided into sites, and each site is configured to use *network file sharing*. If the deployment descriptor is configured with `scp`, the file transfer only has to be performed a time proportional to the number of sites used (2 for the experiment). On the other hand, since the failsafe mechanism transfers files from node to node using the file transfer API (of section 4.1), then the overhead is proportional to the number of acquired nodes.

It is important to note, that when using failsafe, the files are deployed in parallel to the nodes. This happens because several invocations of push, on a set



(a) Time (b) Speed

Fig. 9. Deployment with 10[MB] File Transfer on Grid5000

of nodes, are eventually served in parallel by those nodes. On the other hand, `scp` transfers the files sequentially to each site in turn. The result is that failsafe reaches a better speed than `scp`, as shown in Figure 9(b), where `scp` averages $1.5[\frac{MB}{sec}]$ while failsafe averages $18[\frac{MB}{sec}]$.

6 Related Work

The importance of file transfer and resource acquisition has been studied, among others, by Giersch et al.[7], and Ranganathan et al.[12], who showed that data transfer can affect application scheduling performance. Solutions for integrating resource acquisition and file transfer have been developed by several Grid middlewares like Unicore[17], and Nordugrid[10]. Our approach differs mainly because it allows on-the-fly deployment while combining heterogeneous resource acquisition and file transfer protocols.

The proposed deployment approach can be seen as a wrapper for third party file transfer tools. Other approaches for using third party tools exist. The main goal behind them is to provide a uniform API. This has been done in Java CoG[18], and GAT[13]. Nevertheless, our motivations differ since we seek on-the-fly deployment, rather than file transfer during application execution.

For transferring files between Grid nodes, GridFTP[1] is a popular tool, which extends the traditional FTP[9]. The approach we propose at the programming level mainly varies from GridFTP because we do not require an underlying file transfer protocol to perform file transfer. On the contrary, we only rely on portable always executable asynchronism with future remote method calls. Therefore, we can benefit from automatic continuation to improve peer to peer file transfer performance, as shown in Figure 8(b).

Concerning the retrieval of files, Unicore[17] and NorduGrid[10] have addressed this issue. Once the job has finished, files generated during the computation can be downloaded from the job workspace using the respective middleware client. Our approach differs because we provide a user triggered API file retrieval mechanism, which allows the user further flexibility. The API can be used by

the application at any point during execution once output results are relevant to be transferred, and not only at the very end of the run.

7 Conclusions and Future Work

We have addressed file transfer for the Grid by focusing on three different stages of Grid usage: deployment, execution and retrieval. Our experiments show that it is possible to integrate heterogeneous file transfer with resource acquisition protocols to allow on-the-fly deployment, which can deploy the Grid application and install the Grid middleware at the same time. Experimentally, we have benchmarked the proposed solution, and shown that it is scalable.

For the application execution, we proposed an asynchronous overlapping file transfer mechanism using `push` and `pull` algorithms, built on top of an active object communication model with futures and wait-by-necessity. Experimentally we showed that both can achieve a performance similar to `rcp`. Additionally, we showed how automatic continuation can be used to transfer files between peers in an efficient way.

Finally, we proposed a user triggered file retrieval mechanism for the Grid. This mechanism uses the algorithms developed here in combination with infrastructure information located inside the deployment descriptors.

In the future we would like to explore distributed file systems, built on top of the proposed file transfer API. We also plan to investigate the interaction of file transfer with structured distributed programming models known as skeletons.

References

1. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.
2. F. Baude, D. Caromel, N. Furmento, and D. Sagnol. Overlapping communication with computation in distributed object systems. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 744–754, Amsterdam, The Netherlands, 1999. Springer-Verlag.
3. F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
4. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
5. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.
6. S. Ehmety, I. Attali, and D. Caromel. About the automatic continuations in the eiffel model. In *International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'98*, Las Vegas, USA., 1998. CSREA.
7. A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files on heterogeneous master-slave platforms. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 364–371, A Coruña, Spain, February 2004. IEEE Computer Society Press.

8. Grid5000. http://www.grid5000.fr.
9. J. Reinolds J. Postel. Rfc959 file transfer protocol.
10. NorduGrid. http://www.nordugrid.org.
11. A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
12. K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 352. IEEE Computer Society, 2002.
13. E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.
14. INRIA OASIS Team and ETSI. 2nd grid plugtests report. http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf.
15. INRIA OASIS Team and ETSI. Second grid plugtests demo interoperability. *Grid Today*, 2005. http://www.gridtoday.com/grid/520958.html.
16. ProActive INRIA Sophia Antipolis OASIS Team. http://proactive.objectweb.org.
17. Unicore. http://www.unicore.org.
18. G. von Laszewski, B. Alunkal, J. Gawor, R. Madhuri, P. Plaszczak, and X. Sun. A File Transfer Component for Grids. In H.R. Arabnia and Youngson Mun, editors, *Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.

A Parallel Data Storage Interface to GridFTP

Alberto Sánchez, María S. Pérez, Pierre Gueant, Jesús Montes, and Pilar Herrero

Facultad de Informática
Universidad Politécnica de Madrid
Madrid, Spain

Abstract. Most of the grid projects are characterized by accessing huge volumes of data. For supporting this feature, different data services have arisen in the "grid" world. One of the most successful initiatives in that field is GridFTP, a high-performance transfer protocol, based on FTP but optimized for wide area networks. Although GridFTP provides reasonably good performance, GridFTP servers keep constituting a bottleneck for data-intensive applications.

One of the most important modules of a GridFTP server is the Data Storage Interface (DSI), which specifies how to read and write to the storage system, allowing the server to transform the data. With the aim of improving the performance of the GridFTP server, we have designed a new DSI, based on MAPFS, a parallel file system. This paper describes this new DSI and its evaluation, showing the advantages of dealing data through this optimized GridFTP server.

Keywords: Data grid, GridFTP, Data Storage Interface (DSI), parallel file system, MAPFS.

1 Introduction

In grid projects there is usually a need of transferring large files among different virtual organizations. This is specially significant in data-intensive applications, where accessing and dealing with data is the most critical process.

The most known protocol for transfer files in wide area networks is GridFTP [1], which is an extension of the popular FTP protocol to provide high-performance transferences in a grid environment.

Although there are different approaches for increasing the performance of the transference between client and servers, (e.g., parallelism and striping), the access to an only server constitutes a bottleneck in the whole system, since the I/O bandwidth could be considerably lower than the network bandwidth. Nevertheless, the advantage of GridFTP is the possibility of modifying its DSI (Data Storage Interface) in order to transform the data retrieval process. Approaches from the parallel I/O field can be succesfully applied to this scenario. This is the main motivation of our work. We have built a new DSI, named MAPFS-DSI, which making use of MAPFS [14], a parallel file system, can improve largely the performance of GridFTP.

The rest of this paper is as follows. Section 2 describes the problems related to data management in grids. Section 3 shows our proposal, MAPFS-DSI, which enhances the file transference in grid environments. In Section 4, the evaluation of MAPFS-DSI is analyzed. Section 5 shows related work. Finally, Section 6 explains the main conclusions and outlines the ongoing and future work.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1203–1212, 2006.
© Springer-Verlag Berlin Heidelberg 2006

2 Data Management Services

One of the most significant steps in the foundations of grids was the definition of OGSA (Open Grid Services Architecture) [7]. This architecture has represented a 180-degree turn in the conception of current grids. From the moment of its definition, there has been a strong convergence between grid and web services. In fact, the grid field has been improved through the use of Web services concepts and technologies. This action has brought a larger number of advantages, since the sinergies of both fields. However, some limitations have also arisen in this context. Maybe the most important drawback of this combination is the poor performance exhibited by web services. In fact, the use of XML and SOAP as transference protocol is not appropriate for performance-critical applications [6].

There are different proposals for dealing with this decrease of performance (see [24], [21], [12]). Nevertheless, none of them are suitable in scenarios demanding high throughput.

In this context, GridFTP has emerged as an optimized protocol for large files transference. Unlike other grid services, GridFTP service is not based on SOAP transferences, but the GridFTP server must be started on a given port. In fact, GridFTP has a control channel and one or more data channels. A GridFTP server services client requests on a predefined port (2811), which corresponds to the control channel. Furthermore, the server is multi-threaded. The client chooses a temporary port to connect to the server's port 2811.

In addition, GridFTP provides two important characteristics [2]:

1. Striping: GridFTP extends the basic protocol for supporting the data transference among multiple servers.
2. Parallelism: GridFTP offers the possibility of using multiple TCP streams in parallel from a source to a sink.

Both features can be used combined and they offer good performance. Nevertheless, the server storage system keeps being a bottleneck.

Different alternatives have been proposed as alternatives (see Section 5). However, as far as we know, any of them are focused on applying two levels of parallelism and agent characteristics to enhance the performance of the GridFTP server.

3 MAPFS-DSI

For building MAPFS-DSI, we need to modify the GridFTP server. The GridFTP server is composed of three modules:

1. *The GridFTP Protocol module*, which is responsible for reading and writing to the network, implementing the GridFTP protocol. In order to be interoperable, this module should not be modified.
2. *The Data Transform module*
3. *The Data Storage Interface*.

The last two modules funcionalities are merged in the Globus Toolkit 4.0.x [8]. From this point, we will name DSI to both modules.

The main features of a DSI are:

- It is responsible for reading and writing to the local storage system. The transformation of data is optional.
- It consists of several function signatures, which must be filled with suitable semantics for providing a specific functionality.
- It can be loaded at runtime.
- The GridFTP server request to the DSI module whatever operation it needs from the underlying storage system. Once the DSI performs this operation, it notifies the GridFTP server the completion of such one.

We have used this flexibility of the GridFTP server for transforming the I/O operations. MAPFS I/O routines are used instead. Next sections describe MAPFS and the DSI we have designed and implemented for the enhanced use of GridFTP.

3.1 MAPFS

MAPFS [15] is a parallel and multiagent file system for clusters, whose main goal is to enhance the I/O operations performed in a cluster of workstations. MAPFS is situated at the client side.

With the goal of performing its tasks, MAPFS is composed of two subsystems with different responsabilities:

1. MAPFS_FS, which implements the parallel file system functionality.
2. MAPFS_MAS, responsible for the information retrieval and other additional tasks.

MAPFS_MAS is an independent subsystem, which provides support to the major subsystem (MAPFS_FS) in three different areas:

- Information retrieval: The main task of MAPFS_MAS is to facilitate the location of the information to MAPFS_FS. Data is stored in I/O nodes, that is, data servers.
- Caching and prefetching services: MAPFS takes advantage of the temporal and spatial locality of data stored in servers. Cache agents in MAPFS_MAS manage this feature.
- Use of hints: The use of hints related to different aspects of data distribution and access patterns allows MAPFS to increase the performance of these operations.

MAPFS is a three-tier architecture, whose layers are:

- *MAPFS clients:* They implement the MAPFS_FS functionality, providing parallelism, the file system interface and the interface to the server-side for accessing data.
- *Storage servers:* In order to fulfill the MAPFS requirements, servers must not be modified. These servers only store data and metadata. Nevertheless, we have defined a formalism called *storage groups* [17], for providing dynamic capacities to storage servers, without modifying them.
- *Multiagent subsystem:* MAPFS_MAS is composed of agents responsible for performing additional tasks, which can be executed on different nodes, including data servers. This fact is not contradictory to our requirements, since servers are not

modified. The tasks of a multiagent system are mainly: (i) give support to the file system for data retrieving; (ii) cache management; (iii) storage group management; and (iv) hints creation and management.

Figure 1 represents the three-tier architecture, showing the three layers and the relation between them. At the top of the hierarchy, the file system interface is shown. MAPFS clients are connected to other two modules: (i) the multiagent subsystem, through MAPFS_MAS interface and (ii) data servers, through the corresponding access interface.

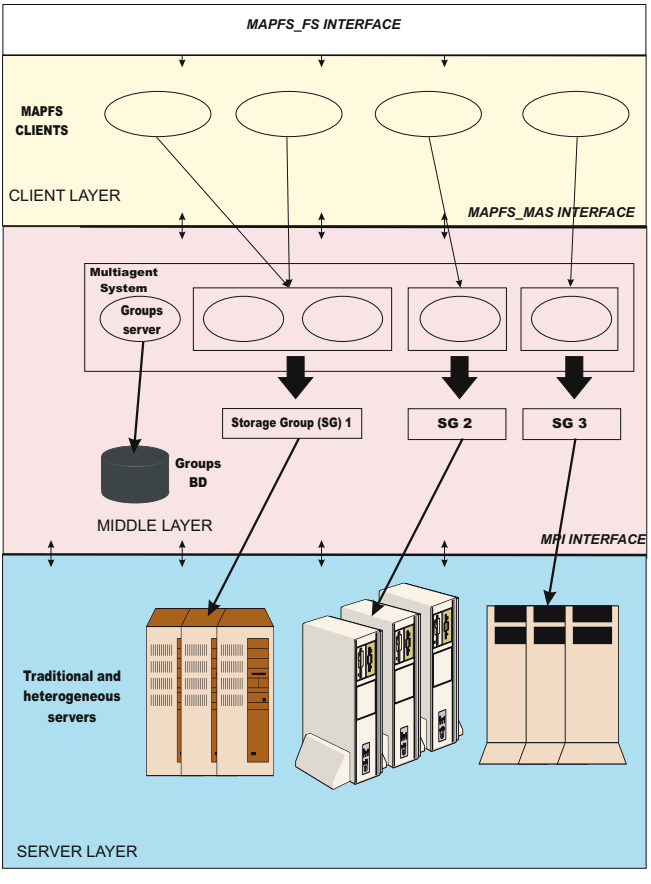


Fig. 1. MAPFS three-tier architecture

For implementing the multiagent subsystem, MPI (Message Passing Interface) [9], [13] is used. This technology provides the following features:

1. MPI is an standard message-passing interface, which allows agents to communicate among them by means of messages.
2. Message-passing paradigm is useful for synchronizing processes.

3. MPI is broadly used in clusters of workstations.
4. It provides both a suitable framework for parallel applications and dynamic management of processes.
5. Finally, MPI provides operations for modifying the communication topologies.

3.2 MAPFS-DSI Architecture

The combination of MAPFS and GridFTP aims to alleviate the bottleneck that constitutes the GridFTP server. The main idea is that the I/O system attached to the GridFTP server can be optimized by means of parallel I/O techniques.

MAPFS-DSI enables GridFTP clients to read and write data to a storage system based on MAPFS. As we have mentioned in the previous subsection, the appropriate architecture for MAPFS is a cluster of worsktations. Thus, the GridFTP server should be the master node from a cluster of workstation, where MAPFS is installed.

The interface of any DSI is composed of the following functions to be implemented:

- send (get)
- receive (put)
- simple commands such as mkdir

We have implemented the previous operations invoking MAPFS functions (A list of these functions can be found in [16]). In order to provide access to MAPFS, the GridFTP server must contain two separated modules, which are designed and implemented to work together:

- The DSI itself. It manages the GridFTP data, acting as an interface between the server and the I/O system. The aim of this module is to provide a modular abstraction layer to the storage system. This module can be loaded and switched at runtime. When the server requires action from the MAPFS system it passes a request to this module. Nevertheless, the I/O operation is not directly performed by the DSI. For this purpose a second module has been implemented.
- The MAPFS driver, the second module previously cited, carries the responsibility of performing the I/O operations. It receives the requests from the DSI and performs a parallel access to the cluster nodes.

This modular structure provides system flexibility, allowing to switch between different interfaces and storage systems without stopping the GridFTP server.

MAPFS-DSI is embedded within the general scenario in which GridFTP is used. As we can see in the Figure 2, there are two independent parts of the architecture that can improve the performance of a data transference operation. Firstly, the specific features of GridFTP (parallelism and striping), which can be used in any GridFTP server. Secondly, the parallelism and striping provided by MAPFS. This implies that the use of MAPFS within the GridFTP server offers two levels of parallelism and striping, avoiding that the server storage system becomes a bottleneck in the whole data transference process.

MAPFS-DAI offers great flexibility, since several combinations of both levels of parallelism and/or striping can be used in different configurations.

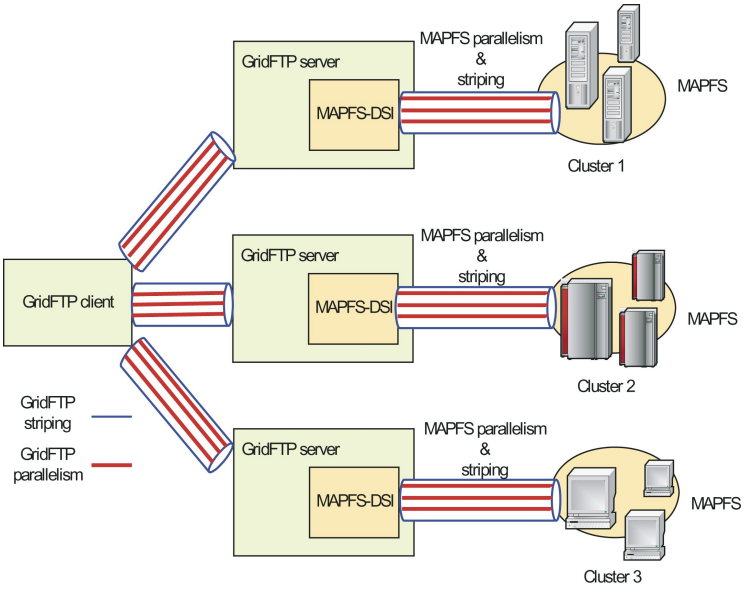


Fig. 2. MAPFS-DSI within a data transference scenario

4 Performance Analysis

This section shows the performance of our Parallel Data Storage Interface. Through this analysis, our aim is stating the performance improvements obtained due to the parallel data access in clusters of workstations.

In orders to run the experiments two storage elements were required. The first one is a cluster composed of eight Intel Xeon 2.40GHz nodes with 1 GB of RAM memory, connected by a Gigabit network. Each node has a hard disk that provides 35 MB/sec approximately in a write access. The second one is a Network Attached Storage (Intel Pentium IV 2.80GHz with 1 GB of RAM memory) composed of four hard disks configured in RAID 0. The performance of the NAS write bandwidth is roughly 220 MB/sec. Both are connected by means of a Gigabit network.

In order to evaluate the performance we have compared three different data access systems. The first one is the usual Data Storage Interface implemented by GridFTP, called *file* DSI, which only accesses to the master cluster node. The second one is our Parallel Data Storage Interface, MAPFS-DSI, to access in a parallel way to the whole cluster. And the last one is the NAS.

For the comparison of the performance of read operations, a 1GB filesize is read from each data file system, and then written to a remote machine connected through a Gigabit ethernet network. The file is written on the memory of the remote machine to avoid writing bottlenecks. Since GridFTP is designed to manage massive amount of data, block size is an important parameter to consider. The bandwidth provided is measured considering the default 1MB block size, then 8MB, 40MB, 80MB, 160MB block sizes. For instance using 8MB blocks, MAPFS-DSI stores 1 MB in a parallel way in each of its nodes.

Figure 3 shows the I/O bandwidth obtained reading the file by using these three techniques. MAPFS-DSI is the most efficient using block-sizes larger than 5MB. With smaller blocks, disk access times tend to reach transfer times. Considering that we are testing our proposal in a real environment and the network is not dedicated, it is acting as a bottleneck. Using MAPFS to read inside a single cluster it is possible to obtain better performance [14]. In short, it can be observed the performance is higher using our Parallel Data Storage Interface, which is limited by the network, both NAS and MAPFS-DSI. Nevertheless, the performance obtained by the *file* Data Storage Interface of GridFTP (file DSI) is limited by the I/O bandwidth of the hard disk of the cluster master node. And this is more critical.

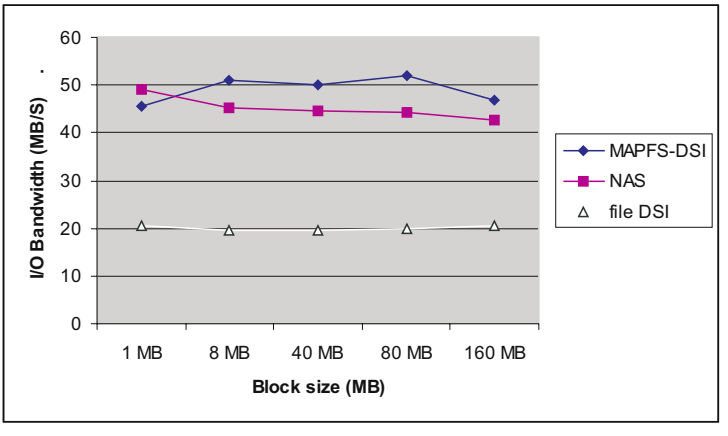


Fig. 3. Comparison of the bandwidth obtained using MAPFS-DSI, file DSI and NAS to read a 1 GB filesize to memory

Figure 4 shows the I/O bandwidth obtained writing the file using by MAPFS-DSI and file DSI. Efficiency is optimal for both systems using 8MB blocks. As reading operation are done in the NAS, the maximum theorical bandwidth is 45 MB/s for 8MB blocksize. In fact MAPFS-DSI obtains 75% of this theorical value, due to overhead introduced by the network transfer and the synchronization among the MAPFS nodes. In that case the efficiency is improved by 24% using the Parallel Data Storage Interface.

Thus, this would allow us to extract some interesting conclusions that assert our previous proposals. As grid environments are composed of different and heterogeneous resources, and clusters stand out among them because of its good relation power vs. cost, it is possible to improve the grid data transfers accessing in a parallel way to the clusters resources.

In addition, it is possible to communicate with GridFTP sites in a tranparent way using the common protocol. It is advisable to take into account the fact that different kinds of systems (such as MAPFS, High Performance Storage System (HPSS), Storage Resource Broker (SRB), and so on), implement different DSI and they could be used together. In these sense, it is possible to improve the performance by means of paralelism among the different sites using striped transfers. The main limitation is the

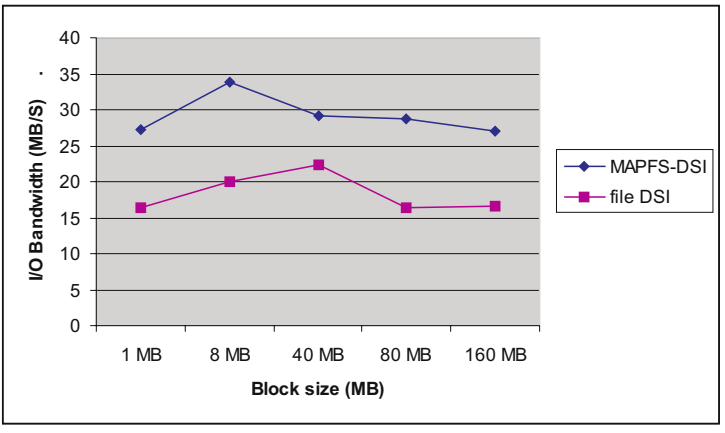


Fig. 4. Comparison of the bandwidth obtained using MAPFS-DSI and file DSI to write a 1 GB filesize

network bandwidth instead of the I/O phase in the different storage elements. But the predictions in Computer Science state that the network performance will increase in a higher scale than the I/O system [23]. Thus, the I/O system will continue being the bottleneck and any improvement aimed to enhance the I/O phase, like MAPFS-DSI, will be welcome.

5 Related Work

Due to the specific necessities for storing and accessing large datasets, the Grid community has concentrated some of its efforts on designing and modelling a large number of storage systems. However, the specific characteristics of each of them are quite different among all of them. Some of these storage systems, such as the Distributed Parallel Storage System (DPSS) [22] or the High Performance Storage System (HPSS) [25], focus on high-performance access to data and utilize parallel data transfer streams and/or striping across multiple servers to improve performance.

The Storage Resource Manager (SRM) interface provides a standard uniform management interface to heterogeneous storage systems, providing a common interface to data grids, abstracting the peculiarities of each particular Mass Storage System. SRMs are middleware components whose function is to provide dynamic space allocation and file management on shared storage components on the Grid [19,18]. SRM interface could be used to access the different storage system such as CASTOR ("CERN Advanced STORage manager") [4], a scalable and distributed hierarchical storage management system developed at CERN in 1999. Other mass storage systems which provide a SRM interface are HPSS, Enstore, JASMine, dCache and SE.

OceanStore [11] provides a global data storage solution with large scalability. However, this infrastructure does not provide demand storage allocations, important for supporting high efficiency in dynamic environments. On the other hand, the Darwin resource management mechanism [5], focuses on the dynamic resource allocation.

One of the most known repository manager is the Storage Resource Broker (SRB) [3]. SRB provides: (i) a centralized resource management middleware for the data grid; (ii) a client interface to such repositories; and (iii) metadata for locating data in the storage system.

Both SRB and HPSS have built customized DSIs for GridFTP [20], [10]. Unlike these DSIs, MAPFS-DSI is intended to enhance the final I/O operations in the data storage by means of parallel I/O and agent techniques, as we have previously mentioned.

6 Conclusions and Future Work

Different improvements of GridFTP, such as striping and parallelism, have become this protocol in one of the most optimized data transference schemes.

However, the storage system attached to the GridFTP server could be the bottleneck of the data transference scenario. With the aim of alleviating this problem, we have designed, implemented and evaluated MAPFS-DSI, which improves the Data Storage Interface (DSI) module of the GridFTP server, by means of parallel I/O and agent techniques.

This paper shows that the combined use of GridFTP and MAPFS-DSI can improve largely the data transference performance.

As future work, we are planning to compare the performance of MAPFS-DSI vs. SRB-DSI and HPSS-DSI.

Acknowledgements

This work is partially supported by the Ontogrid Project (FP6-511513) and two UPM pre-doctoral grants.

References

1. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. Web Page, September 2002.
2. William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The globus striped gridftp framework and server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 54, Washington, DC, USA, 2005. IEEE Computer Society.
3. Chaitanya K. Baru, Reagan W. Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 5. IBM, 1998.
4. The Castor Project, *http://www.castor.org*.
5. Prashant R. Chandra, Allan Fisher, Corey Kosak, T. Ng, Peter Steenkiste, Eiichi Takahashi, and Hui Zhang. Darwin: Customizable resource management for value-added network services. In *ICNP*, pages 177–188, 1998.
6. Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

7. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Published online at *http://www.globus.org/research/papers/ogsa.pdf*, January 2002.
8. The Globus Project. *http://www.globus.org*.
9. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.
10. HPSS - high performance storage system, *http://www.hpss-collaboration.org*.
11. John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, pages 190–201, 2000.
12. Hartmut Liefke and Dan Suciu. Xmill: An efficient compressor for xml data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 153–164. ACM, 2000.
13. Mpi Forum. *http://www.mpi-forum.org/docs/docs.html*.
14. María S. Pérez, Jesús Carretero, José M. Peña Félix García, and Víctor Robles. MAPFS: A flexible multiagent parallel file system for clusters. *Future Generation Computer Systems*, 22(5):620–632, 2006.
15. María S. Pérez, Jesús Carretero, Félix García, José M. Peña Sánchez, and Victor Robles. A flexible multiagent parallel file system for clusters. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, *International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2003.
16. María S. Pérez, Félix García, and Jesús Carretero. A new multiagent based architecture for high performance I/O in clusters. In *ICPP Workshops*, pages 201–206. IEEE Computer Society, 2001.
17. María S. Pérez, Alberto Sánchez, José M. Peña, and Víctor Robles. A new formalism for dynamic reconfiguration of data servers in a cluster. *J. Parallel Distrib. Comput.*, 65(10):1134–1145, 2005.
18. Arie Shoshany et al. SRM Interface Specification v.2.1, *http://sdm.lbl.gov/srm-wg/doc/srm.spec.v2.1.final.pdf*.
19. Arie Shoshany et al. SRM Joint Design v.1.0, *http://sdm.lbl.gov/srm-wg/doc/srm.v1.0.pdf*.
20. Gt 4.0 gridftp: Storage resource broker (SRB),*http://www.globus.org/toolkit/docs/4.0/data/gridftp/gridftp_srb.html*.
21. Neel Sundaresan and Reshad Moussa. Algorithms and programming models for efficient representation of xml for internet applications. *Computer Networks*, 39(5):681–697, 2002.
22. Brian Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake. A network-aware distributed storage cache for data intensive environments. In *HPDC*, 1999.
23. Cleo Vilett. Moore's law vs. storage improvements vs. optical improvements. *Scientific American*, January 2001.
24. W3C. Soap message transmission optimization mechanism, *http://www.w3.org/tr/soap12-mtom*, January 2005.
25. Richard W. Watson. High performance storage system scalability: Architecture, implementation and experience. In *MSST*, pages 145–159. IEEE Computer Society, 2005.

Parallelization of a Discrete Radiosity Method Using Scene Division

Rita Zrour, Fabien Feschet, and Rémy Malgouyres

LAIC, University of Auvergne, IUT - Campus des Cézeaux
63172 Aubière Cedex, France
{zrour, feschet, remy.malgouyres}@laic.u-clermont1.fr

Abstract. We present in this article a parallelization of a discrete radiosity method, based on scene division and achieved on a cluster of workstations. This method is based on discretization of surfaces into voxels and not into patches like most of the radiosity methods do. Voxels are stocked into visibility lists representing the space partition into discrete lines and allowing mutually visible neighbour voxels to exchange their radiosities. The parallelization distributes the scene among the processors by dividing it into parts. Exchanges of radiosity values are accomplished between neighbourhood voxels belonging to a same list but located on different processors. This parallelization improved time and distributed memory offering thus the capability to deal with large scenes.

1 Introduction

Radiosity is a very useful technique used in many domains. It is used to simulate several physical processes involving heat transfer, light propagation and weather forecast. Its main field of application is 3D rendering in computer graphics. Like ray tracing, radiosity is used to generate photo-realistic images for games and movies with high quality images to make them more realistic. One of the advantages of radiosity is that it is nearly viewpoint independent ; it computes the energy distribution for a specific point of view and necessitates minimum computation when the point of view is changed. Radiosity is a very demanding process in terms of time and memory which leads researchers to parallelize radiosity algorithms trying to minimize both time and memory. We have seen in the past many parallelizations of different radiosity algorithms. These algorithms can be classified according to their parallelization strategies as well as their memory system.

Various strategies for parallelization were adopted for the radiosity algorithms. In [4] the environment is split into sub-environments and a visibility mask is used for the transfer of light. In [14] the scene is divided into cells by axial occluders then every cell is augmented to form its visible sets according to cell's visibility. A master slave principle is used in [5] to split the radiosity solving tasks between a master and slaves. A new partitioning method applied to non axial building is proposed in [15] where the partitioning and the visibility computations are done in a pre-processing step.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1213–1222, 2006.
© Springer-Verlag Berlin Heidelberg 2006

From the point of view of the memory system, we can distinguish two systems: shared memory systems [7] and distributed memory systems [3,11]. Distributed memory systems can be divided into two branches: the cluster of workstations and the distributed shared memory. The cluster of workstations is composed of many machines where the exchanges between the machines is done via the message passing strategy [3,9,12,14]. The distributed shared memory systems use shared variables for communication [4,8,10,11].

We propose in this paper a parallel implementation of the discrete radiosity method [1]. This method is different from most radiosity methods because it is based on discretization of surfaces into voxels and not into patches. In terms of complexity the method is quasi-linear in time and space and has a lower complexity than other methods for large scenes containing a lot of details. A first parallelization of this method has been achieved in [2] where a distribution of the tasks as well as a data distribution based on voxels transmission were proposed. The parallelization presented in this article is based on scene division. Voxels are stable on the processors and only the radiosity is communicated. It has been tested on a distributed memory system, a cluster of workstations composed of bi-processors hyperthreaded Xeon Pentium processors.

The paper is organized as follows. Section 2 explains the sequential algorithm and its complexity. Section 3 presents the distribution of data through scene division. Finally Section 4 states some conclusions and perspectives.

2 Sequential Algorithm

2.1 Discretization of the Radiosity Equation

Radiosity is defined as the total power of light leaving a point. The continuous radiosity equation has been discretized in [6]. The voxel-based radiosity equation is the following:

$$B(x) = E(x) + \rho_d(x) \sum_{\overrightarrow{\sigma} \in D} B(V(x, \overrightarrow{\sigma})) \frac{\cos\theta(x, V(x, \overrightarrow{\sigma}))}{\pi} \hat{A}(\overrightarrow{\sigma}) \qquad (1)$$

This equation shows that the total power of light $B(x)$ leaving a voxel x depends on two terms, the first is the proper emittance of this voxel as a light source (the $E(x)$ term), and the second is some re-emission of the light it receives from its environment (the sum). The term $B(\cdot)$ that is present in both sides of the equality, reflects interdependence between a point and its environment ; it does not consider any outgoing direction, so it supposes that each point emits light uniformly in every direction (diffuse hypothesis). The factor $\rho_d(x)$ indicates that a point re-emits only a fraction of the light that it receives. D is a set of discrete directions in space. $V(x, \overrightarrow{\sigma})$ is a visibility function, returning the first point y seen from x in the direction of $\overrightarrow{\sigma}$. The term $\hat{A}(\overrightarrow{\sigma})$ is the fraction of a solid angle associated to the direction $\overrightarrow{\sigma}$ it quantifies how much of an object is seen from a point, we call it a *direction factor*. The $\cos\theta(x, V(x, \overrightarrow{\sigma}))$ expresses that incident

light is more effective when it comes perpendicularly to the surface. Finally the π factor is a normalization term deriving from radiance considerations.

This equation is usually solved using Gauss-Seidel relaxation requiring thus several iterations to obtain an estimated solution achieving a reasonable error threshold [13].

2.2 Computing Visibility Using Discrete Lines

An important information in the radiosity equation is the visibility factor, reflecting the visibility between the voxels allowing mutually visible voxels to exchange their radiosities along a given direction. Our radiosity algorithm is based on notions of discrete geometry, the partition of space into 3D discrete lines. Given a voxel $(x, y, z) \in \mathbb{Z}^3$, this voxel belongs to the discrete line $L_{i,j}$ where (i, j) are two integers calculated from both the coordinates of the voxel and the directing vector of the current direction. The set of all $L_{i,j}$'s represents the space partition into 3D discrete lines (see [1] for details). The placement of voxels into their list $L_{i,j}$ does not insure correct visibility between the voxels ; to ensure the correct visibility a possible solution consists in doing a precomputation step that sorts the voxels according to different lexicographic orders ; usually eight lexicographic orders are needed but four are sufficient because of symmetries. Once this precomputation step is done, the correct sorting is chosen according to the direction and then the voxels are placed into their list $L_{i,j}$.

2.3 Algorithm and Complexity

The sequential algorithm of the radiosity is described shortly in Fig. 1. This algorithm requires two parameters, the number of iterations I and the number of directions D. If N is the number of voxels of the scene, the time complexity is: $4 \times O(NlogN) + I \times D \times (O(N) + O(N))$. The term $4 \times O(NlogN)$ represents the four lexicographic sorting of the voxels. It is negligible, because the four sorting are pre-computed once. The term $I \times D \times (O(N) + O(N))$ reflects the dispatch of the voxels in their lists and the propagation of the light in the lists. Note that I is a small constant. D is a constant that is independent of the number of voxels ; it depends on the intensity of the light sources. It is increased to avoid aliasing effects when the sources are small but having high intensities. As for the space complexity, we just need to store the voxels in memory. More details about the complexity can be found in [1]. One of the goals of the present work is to obtain a parallelization distributing the voxels since the complexity is mostly influenced by the number of voxels of the scene.

3 Data Distribution Through Scene Division

3.1 General Principle

The data distribution distributes the voxels between the processors by dividing the scene into parts placing virtual walls as separators [4]. The division is

```
Prepare the four lexicographic orders;
Compute a set of discrete directions;
for each direction do
    Compute and store the associated direction factor;
for iterations=1 to MaxIterations do
    for each direction do
        Select the appropriate lexicographic order
        for each voxel do
            Put it in the list (3D discrete line) it belongs to;
        for each list do
            Propagate radiosity between contiguous voxels;
        Reset the lists;
```

Fig. 1. Sequential algorithm

accomplished in 1D according to the largest dimension (length, width or height). The placement of the wall depends on the density of voxels and the number of processors in a way to give to all of the processors about the same number of voxels. Once the voxels distributed, every processor starts the radiosity computation. It dispatches its voxels in their lists and propagates light in these lists. After these two computations, the exchange step should start. It consists in exchanging radiosities between neighbouring voxels belonging to a same list but present on different processors. Fig. 6 shows an example of the scene distribution and voxels exchange. Note that the exchange do not involve just neighbouring processors but can be also between distant processors due to the presence of a list filled on one processor and empty on another. Once the exchanges between the processors are accomplished, the radiosity of implicated voxels is updated. To achieve this principle two approaches were possible . The first approach does the exchange between neighbour processors based on a 1D cartesian topology and non-blocking communications (Section. 3.2) and the second does collective communications between the processors (Section. 3.3).

3.2 Exchange Between Neighbor Processors

This approach builds a virtual 1D cartesian topology between the processors. Non-blocking communications are then accomplished just between direct neighbour processors. Non-blocking communications allow the processors to do their work without having to wait for others to accomplish. It is good because some or more processors may have more or less data to send or receive than others. However in this approach the processors do not exchange any information about each other. Every processor sends the radiosity information regarding the first and last voxel of each of its non empty list to the preceding and next processor respectively (or vice versa depending on the direction). It is just at the reception when decoding the data to update the voxels that the processor notices that a received data from a specific list cannot be updated because the local list is empty ; at this moment it knows that the information received should be

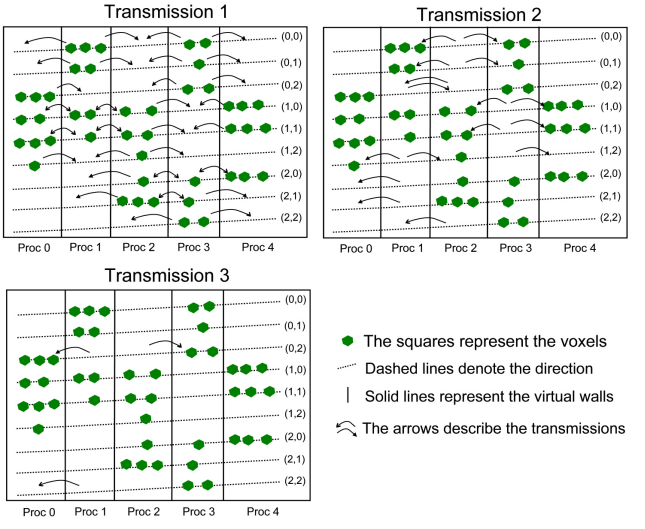


Fig. 2. The different transmissions for a given scene using cartesian topology and non-blocking communications

transmitted again to the neighbour. This work is repeated until no empty list is reached or the end or begin processor are reached. Fig. 2 shows an example of this approach. In the first transmission "Transmission 1" every processor sends radiosity information for each of its non empty lists to its neighbour processor. At the reception if the list is non empty the voxels are updated otherwise the information is retransmitted again to the next neighbour generating a second transmission "Transmission 2" and the procedure continues until no empty lists receive data or the first or last processors are reached. The main disadvantage of this approach is that sometimes the data is retransmitted and not used for update. For example in Fig. 2 the list $(2, 2)$ is filled on processor 3 and empty on the others, despite the processors cannot know and the data is transmitted to all of the others. This approach may be performing if most of the lists present on different processors are not empty. However, most of the time, a non empty list is present on distinct processors, necessitating though many transmissions before the update of voxels.

Results. The results of this approach are tested on two different scenes the Cabin scene and the Church scene composed of 3 million and 3.5 million of voxels respectively. The tests are done with 6 iterations and using 5000 directions for the Cabin scene and 8000 directions for the Church scene. It should be noted that all the scenes used for testing the data distribution approaches were generated using a modelling program designed by our own group using mainly cubic B-splines and cubes. Fig. 3 shows the time variation as the number of processors increases. Fig. 4 shows the speed-up ; a speed-up of 4 is obtained using 10 processors. We call "first exchange" the time for the first transmission with the voxels update,

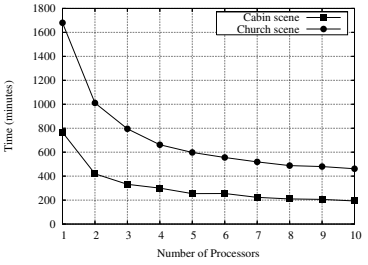


Fig. 3. Time versus the number of processors using the cartesian topology and non-blocking communications

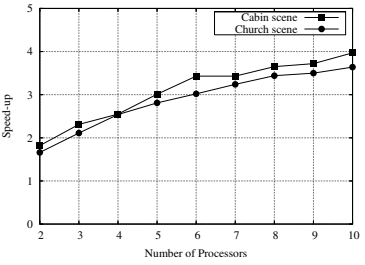


Fig. 4. Speed-up for two different scenes using the cartesian topology and non-blocking communications

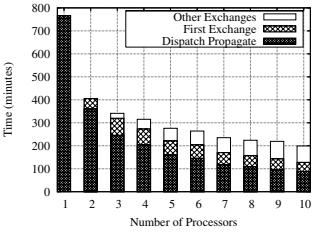


Fig. 5. Time distribution for the different operations for the Cabin scene using the cartesian topology and non-blocking communications

and "other exchange" the time for the rest of the transmission and updates. Fig. 5 shows the time distribution of the different operations for the Cabin scene. It can be seen that the dispatch-propagate time is divided by the number of processors. As for the exchanges, "other exchange" is zero when having two processors since one transmission is needed. When the number of processors increases the exchanges varies slightly in time but remain almost constant.

3.3 Collective Exchange

The exchange between neighbours (Section. 3.2) necessitates many transmissions before the update of the voxels is done. This leads to think of sending the radiosity information to the right processor directly without having to pass by intermediate processors. A direct communication can be established if each processor knows in advance before the exchange, for every non empty list the processor to which it should communicate the radiosity information ; if such processor is known then the radiosity information designated to a given processor is stored and then sent to the right processor. The information received is then used to update the voxels radiosities. The steps of this approach and its results are detailed in the next subsections.

Communication Matrices. As was stated in section 2.2, each list is characterized by two integers (i, j) representing the space partition into discrete lines.

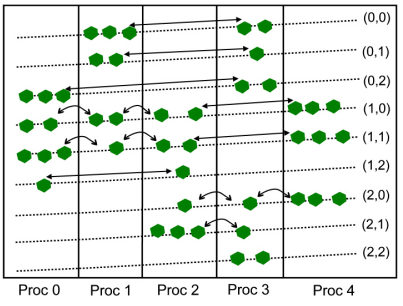


Fig. 6. Scene distribution and radiosity exchange between the first and last voxels of the lists

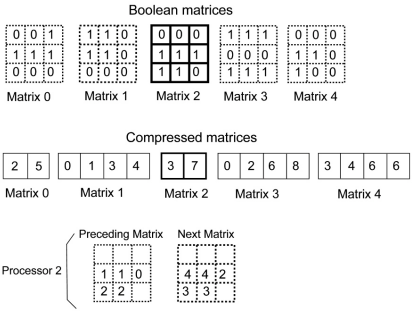


Fig. 7. Compression, gathering and computation of the communication matrices for the scene of Fig. 6

The direct communication between the processors implies that every processor has information regarding empty and non empty lists present on all the other processors. Collective communications are therefore required to gather lists information of the processors. Representing the empty lists by false and the non empty by true, allows every processor to construct its own boolean matrix. However for a given scene the (i, j) interval is a large interval that depends on the direction making the gathering of all the boolean matrices between the processors expensive in time (gathering operation) and in memory (stocking the gathered matrices) especially when the number of processors increases. Constructing a matrix containing just the non empty lists location is also expensive in time and memory since the number of non empty lists is large. Compressing the matrices is a solution. The scenes have usually many empty lists ; for the non empty lists there is a large possibility that they locate next to each. This facilitates the compression. Instead of putting a boolean number for every lists describing if it is empty or not, or noting the location of the non empty lists, it is possible to compress the matrices by noting for a series of non empty lists, the location of the first list and that of the last list. Fig. 7 shows an example of the boolean matrix together with the compressed matrices for the scene shown in Fig. 6. The gathering is done after the compression of the matrices. Fig. 8 reflects the effect of the compression for the Cabin scene distributed on 9 processors. It is clear that the number of elements of the matrix decreases largely after compression. After the gathering every processor has the information concerning the lists of all the other processors. The two communication matrices can thus be found. These two matrices contain for every non empty list the rank of the next and preceding processors to which the last and first voxel of a list should communicate their radiosity information respectively and vice versa depending on the direction. Note that for a given (i, j) list, the next or preceding processor will be the first next or preceding processor in which a nonempty (i, j) list is found. Fig. 7 shows the communication matrices (next and preceding matrices) obtained on "proc 2" after the gathering of the matrices ; these communication matrices will allow processor 2 to establishes the right communication at a given direction.

Filling Data. Once the two communication matrices found, each processor traverses all of its non empty lists, finds from the next and preceding matrices the processor to which it should be sent ; if it is to be sent outside the machine, it extracts the radiosity information of the first and last voxel of the list, and fills these information in the designated data structure. Note that data structures for the sent voxels are labelled with respect to the destination they should go so that to facilitate the filling and sending steps.

Sending and Receiving Data. Once the filling of the data structures is done, the exchange of the radiosity information between the machines starts. These exchanges are collective from every processor to all of the others sending and receiving just necessary information.

Radiosity Update. After the exchange operation, the received data must be decoded in order to update the radiosity of voxels. In the received information the list number (i, j) and the information source (i.e. the rank of the processor) from which the data is sent will be the key to know whether the radiosity of the first or last voxel of the list (i, j) has to be updated.

Results. The results of this parallelization are tested for three different scenes, the Cabin scene (3 million voxels), Church scene (3.5 million voxels) and the Room scene (6 million voxels). Tests are done using 6 iterations and 5000, 8200

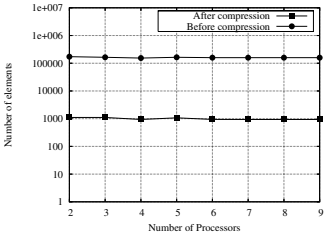


Fig. 8. Varations in the number of elements after compression for the Cabin scene using log-scale for the ordinate

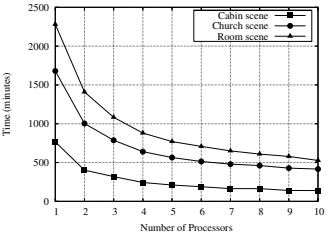


Fig. 9. Time versus number of processors using collective communications

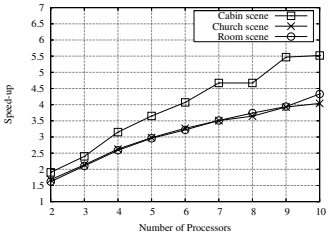


Fig. 10. Speed-up for three different scenes using collective communications

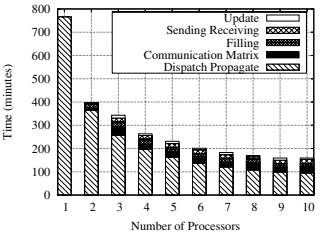


Fig. 11. Time distribution for the operations for the Cabin scene using collective communications

and 6300 directions. Fig. 9 shows the time versus the number of processors for the three scenes. Fig. 10 shows the speed-up of this parallelization. A speed-up of 5.5 is obtained using 10 processors. The time distribution for the different operations of this parallelization is shown on Fig. 11. It can be noticed that the time of the dispatch and propagate, is divided by the number of processors. As for the time of the four operations added by the parallelization, (Communication Matrices, Filling, Sending Receiving and the Radiosity Update) it does not vary too much and is almost stable as the number of processors increases.

4 Conclusion and Perspectives

In this paper we have examined two approaches for the data distribution using scene division. They have improved the sequential time while distributing the data. The second approach showed a better speed-up since the communications are done directly to the right processor without any secondary transmissions which is important when dealing with large scenes. As for the perspectives and further works, we will try to cumulate the exchanges for the collective communications for many directions and do them for a set of directions instead of doing them at every direction ; this is expected to decrease the time by minimizing the synchronization between the processors. A comparison between the data distribution presented in this article and the one detailed in [2] will also be the subject of another work. We also intend to apply the distribution of the computation together with the data distribution proposed in this paper to distribute the directions. The number of directions may sometimes be a big constant if the light sources are intense and so distributing them may bring further time improvement.

Acknowledgments. The authors wish to acknowledge the support of Conseil Regional d'Auvergne within the framework of the Auvergrid project.

References

1. Chatelier, P., Malgouyres, R.: A Low Complexity Discrete Radiosity Method. Discrete Geometry for Computer Imagery. **3429** (2005) 392–403
2. Zrour, R., Chatelier, P., Feschet, F., Malgouyres, R.: Parallelization of a discrete radiosity method. Euro-Par (2006) (to appear)
3. Arnaldi, B., Priol, T., Renambot, L., Pueyo, X.: Visibility Masks for solving Complex Radiosity Computations on Mutliprocessors. Parallel Computing **23(7)** (1988) 887–897
4. Renambot, L., Arnaldi, B., Priol, T., Pueyo, X.: Towards Efficient Parallel Radiosity for DSM-based Parallel Computers Using Virtual Interfaces. Proceedings of the IEEE symposium on Parallel rendering (1993) 76–86
5. Funkhouser, TA.: Coarse-grained parallelism for hierarchical radiosity using group iterative methods. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (1996) 343–352
6. Malgouyres, R.: A Discrete Radiosity Method. Discrete Geometry for Computer Imagery **2301** (2002) 428–438

7. Podehl, A., Rauber, T., Runger, G.: A Shared-Memory Implementation of the Hierarchical Radiosity Method. Theoretical Computer Science **196** (1998) 215–240
8. Bouatouch, K., Mebard, D., Priol, T.: Parallel Radiosity Using a Shared Virtual Memory. Proceedings of Advanced Techniques in Animation, Rendering and Visualization (1993) 71–83
9. Sturzlinger, W., Wild, C.: Parallel Visibility Calculations for Radiosity. Proceedings of the Third Joint International Conference on Vector and Parallel Processing: Parallel Processing (1994) 405–413
10. BarLev, A., Itzkovitz, A., Raviv, A., Schuster, A.: Parallel Vertex-To-Vertex Radiosity on a Distributed Shared Memory System. Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel. **1457** (1998) 238–250
11. Sillion, F., Hasenfratz, J.M.: Efficient Parallel Refinement for Hierarchial Radiosity on a DSM Computer. Third Eurographics Workshop on Parallel Graphics and Visualisation (2000) 61–74
12. Guitton, P., Roman, J., Subrenat, G.: Implementation Results and Analysis of a Parallel Progressive Radiosity. Proceedings of the IEEE symposium on Parallel rendering (1995) 31–38
13. Cohen, M.F., Greenberg, D.F.: The hemi-cube: a radiosity solution for complex environments. SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques **19(3)** (1985) 31–40
14. Feng, C., Yang, S.: A parallel hierarchical radiosity algorithm for complex scenes. Proceedings of the IEEE symposium on Parallel rendering (1997) 71–78
15. Meneveaux, D., Bouatouch, K., Maisel , E., Delmont, R.: A New Partioning Method for Architectural Environments. The Journal of Visualization and Computer Animation **9(4)** (1998) 195–213

A Mixed MPI-Thread Approach for Parallel Page Ranking Computation

Bundit Manaskasemsak[1], Putchong Uthayopas[2], and Arnon Rungsawang[1]

[1] Massive Information & Knowledge Engineering
Department of Computer Engineering, Faculty of Engineering
Kasetsart University, Bangkok 10900, Thailand
{un, arnon}@mikelab.net
[2] Thai National Grid Center, Software Industry Promotion Agency
Ministry of Information and Communication Technology, Thailand
putchong_ut@thaigrid.or.th

Abstract. The continuing growth of the Internet challenges search engine providers to deliver up-to-date and relevant search results. A critical component is the availability of a rapid, scalable technique for PageRank computation of a large web graph. In this paper, we propose an efficient parallelized version of the PageRank algorithm based on a mixed MPI and multi-threading model. The parallel adaptive PageRank algorithm is implemented and tested on two clusters of SMP hosts. In the algorithm, communications between processes on different hosts are managed by a message passing (MPI) model, while those between threads are handled via a inter-thread mechanism. We construct a synthesized web graph of approximately 62.6 million nodes and 1.37 billion hyperlinks to test the algorithm on two SMP clusters. Preliminary results show that significant speedups are possible; however, large inter-node synchronization operations and issues of shared memory access inhibit efficient CPU utilization. We believe that the proposed approach shows promise for large-scale PageRank applications and improvements in the algorithm can achieve more efficient CPU utilization.

Keywords: PageRank computation, parallel computing, link analysis.

1 Introduction

The information and knowledge resources provided via the Internet continue to grow at a rapid rate, making effective search technology an essential tool for users of this information. However, the continual growth in volume of web pages presents a great challenge to search engines that must classify the relevance of web resources to user searches. Much current research is directed toward finding more efficient methods to obtain effective search results. One important research area is the development of algorithms to estimate the authoritative score of each web page by analyzing the web's hyperlinked structure. Leading algorithms are HITS [6] and PageRank [10], first proposed in 1998 and subsequently enhanced. The scores computed by these algorithms are utilized by the search engine's ranking algorithm: pages with the more

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1223 – 1233, 2006.
© Springer-Verlag Berlin Heidelberg 2006

significant or higher scores are more highly ranked in search results ordering. Currently, web link analysis remains one of the most important components of the search engine.

The PageRank algorithm is less complex than HITS, making it more practical for large-scale application; PageRank is utilized by several well-known search engines such as Google. PageRank requires analyzing the entire hyperlinked structure of the web graph once and then iteratively calculates the page scores. Unfortunately, the vast number of pages that must be ranked increasingly make PageRank very computationally expensive. For large-scale computation, most researchers usually propose to first partition a huge web graph into several parts and then to compute them separately. Some studies, such as [3, 1, 5], propose to sequentially compute each partition and then combine the sub-results into the global PageRank scores. Other studies utilize parallel computing on a PC cluster [8, 2] or even distributed P2P architecture [14, 15] to improve performance.

In this paper, we investigate the use of cluster technology together with an efficiently parallelized version of the PageRank computation to improve performance. The main idea of our algorithm is to efficiently employ the computing power of the cluster to compute subsets of PageRank scores in parallel, and then combine them to obtain the total scores of web pages. We also investigate the use of lightweight processes, or threads [11], in a Symmetric Multiprocessing (SMP) environment to reduce communication overhead and take advantage of shared memory. Cluster communication is implemented using the standard Message Passing Interface (MPI) protocol [9]. Since our implementation employs both the multi-threading and cluster computing models, we call this approach to parallelization a "mixed model".

The rest of this paper is organized as follows. Section 2 briefly reviews the PageRank algorithm and introduces acceleration techniques for PageRank computation. Section 3 discusses the need of parallelization and gives the detail of such algorithms as well as system design. Section 4 describes our experiments and discusses the results. Finally, section 5 presents conclusions and planned future work.

2 Basic PageRank Concept

2.1 The Intuition

The concept behind PageRank [10] is to estimate the importance of web pages by hyperlinked structure analysis. A link from page u to page v indicates that the author of u recommends and thus confers some importance on page v. Furthermore, a page mostly referred by other "important" pages is also important.

To mathematically formulate this intuitive concept, let n be the total number of pages in the web graph and let R be the PageRank vector: $R(u)$ is the PageRank score of page u. Also let $O(u)$ be the number of pages that u points out, called "out-degree". All pages u linked to v are grouped into a set of backward pages of v, denoted B_v. The rank score of all pages v can be computed using the iterative formula:

$$\forall v : R^{(k+1)}(v) = \alpha \sum_{u \in B_v} \frac{R^{(k)}(u)}{O(u)} + (1-\alpha)E(v) \quad (1)$$

The two terms in this equation represent two factors contributing to a page's rank. The first is the traditional rank propagation calculated from the hyperlinked structure, weighted by α (usually set to 0.85). The second term represents a random surfer process over a uniform distribution (i.e., $\forall v : E(v) = 1/n$). When the surfer visits any page he can subsequently jump to a randomly selected page, with probability $1/n$. This term also guarantees convergence of $R^{(k)}$ and avoids the "rank sink" problem [10]. The convergence of $R^{(k)}$ can be proved by application of Markov's Theorem [12].

$R^{(0)}$ is the initial distribution; in general, the process is started with a uniform distribution, $\forall u : R^{(0)}(u) = 1/n$. Iterative computation of $R^{(k)}$ is performed using Equation (1) until the rank scores converge. The usual convergence criterion is that the relative change in scores of all pages between iterations k and k+1 be below a prescribed tolerance. That is, $\forall v$

$$\frac{\left| R^{(k+1)}(v) - R^{(k)}(v) \right|}{\left| R^{(k)}(v) \right|} \leq \delta \quad (2)$$

The relative tolerance for convergence, δ, is a pre-assigned value; in our experiments, δ is set to 0.0001.

2.2 Adaptive PageRank Technique

Application of the PageRank computation in Equation (1) reveals that the convergence rate of elements of $R(v)$ is highly non-uniform. Kamvar et al. [4] have found that the rank computation for many pages with small rank scores converge quickly to their final values, while the ranks of pages with high scores take a much longer time for the values to converge.

To eliminate the redundant computation of converged page ranks, Kamvar et al. proposed an "adaptive PageRank algorithm" [4] that omits recomputation of PageRank scores that have already satisfied the convergence criterion. For these pages, they simply use the previous score in successive iterations. This reduces the running time of the original algorithm. Using their modification to Equation (1) yields:

$$\forall v : R^{(k+1)}(v) = \begin{cases} R^{(k)}(v) & \text{if } R(v) \text{ converged} \\ \alpha \sum_{u \in B_v} \frac{R^{(k)}(u)}{O(u)} + (1-\alpha)E(v) & \text{otherwise} \end{cases} \quad (3)$$

A PageRank score $R^{(k)}(v)$ is marked as converged when it satisfies Equation (2). The algorithm terminates when all PageRank scores have been marked as converged. We utilize this adaptive technique in our parallel implementation, described next.

3 Parallel PageRank Computation

To compute the PageRank scores of the totality of pages comprising the hyperlinked structure of the Internet by application of Equations (1) or (2) would require massive computing power as well as enormous amounts of memory. In practice, this computation is nearly infeasible. To obtain a more tractable algorithm, we exploit parallelism and partitioning of the problem, and utilize shared resources of a computational cluster.

In this section, we introduce a parallelized version of the PageRank algorithm. First we develop the web graph representation. Then, we provide detail of the parallel algorithm.

3.1 Web Graph Representation

From the crawled web collection, we only consider the hyperlinks between pages. So we first map URLs into ordinal numbers, and represent the web's hyperlinked structure using three binary files. The first one, called a *link structure file* (*L*), represents the relationship between pages via their hyperlinks. Each record in this file consists of *dest_id* field (the target page) and a list of *src_id* fields (the set of authoritative pages.) The other two files, called the *out-degree file* (*O*) and *in-degree file* (*I*), contain the numerical out-degree and in-degree, respectively, corresponding to each *dest_id* in file *L*. An example of these files is textually shown in Fig. 1.

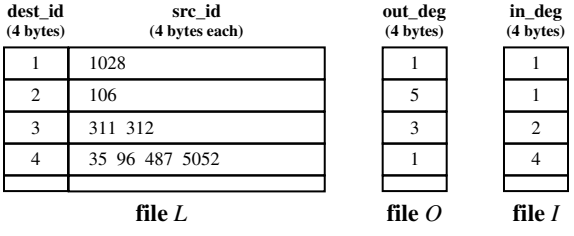


Fig. 1. Three binary files representing the link structure of a web graph

As shown in Fig. 1, all values are expressed as 4-byte integers. In this example, *dest_id* 1 is the target of a hyperlink from page (*src_id*) 1028, and it also has a hyperlink to one other page. The *dest_id* 2 is the target of a hyperlink from page (*src_id*) 106 and has hyperlinks to five other pages.

In the following subsections we describe an approach to parallelizing the PageRank computation utilizing a partitioning of these data files.

3.2 PPR-M Algorithm

The PPR-M Algorithm [13] applies the adaptive PageRank technique (expressed in Equation (2)) to a cluster computing environment. Implementation of the PPR-M algorithm uses the MPI model for message-based communication between nodes in the cluster.

The algorithm first partitions the three binary files representing a web graph (pre-processing phase) for assignment to compute nodes. The files L and I are partitioned for assignment to compute nodes, while the file O is not partitioned. Let p be the number of compute nodes used in the computation. Then we partition L and I into p equal parts by *dest_id*. Each node is assigned an identifying number i ($0 \le i < p$) and allotted a partition of the *dest_id* with data L_i and I_i. Each node will receive a copy of the entire O file.

Pseudo-code of the PPR-M algorithm is shown in Fig. 2. Before beginning the computation, each node loads the files I_i and O into main memory. The algorithm also loads as much of the file L_i into memory as possible (line 1), while the remaining values are loaded from hard disk as required. The algorithm iteratively performs the adaptive PageRank computation (lines 7-12) until all ranks converge. After completing each iteration, every node exchanges its computed rank scores, called a synchronization process (line 14). Further details of this process are given in [13].

PPR-M (i, L_i, I_i, O, p, n)

1: load file L_i, I_i, and O into main memory
2: V_i : source PageRank vector, is initialized to $\left[1/n\right]_{n\times 1}$
3: V_i' : target PageRank vector, is initialized to $\left[0\right]_{\frac{n}{p}\times 1}$
4: *score* : a temporary score
5: While all pages do not converge
6: For each record $l \in L_i$
7: $score = 0$
8: If $l.dest_id$ converges then
9: $V_i'[l.dest_id] = V_i[l.dest_id]$
10: Else
11: Compute all $l.src_id$ as $\frac{V_i[l.src_id]}{O[l.src_id]}$ and add to *score*
12: $V_i'[l.dest_id] = (\alpha \times score) + \frac{(1-\alpha)}{n}$
13: Store V_i' in V_i
14: Synchronize V_i' with other processess and also store in V_i
15: Report V_i' as local PageRank vector

Fig. 2. The parallel PageRank algorithm using only MPI

3.3 PPR-MT Algorithm

The use of parallel processing in the PPR-M algorithm reduces the elapsed computational time required to compute PageRank scores, but adds significant time for network communications during the synchronization process after every iteration. In the PPR-M algorithm, these communications are managed entirely by the MPI library, which also adds overhead. To reduce the communication overhead, we

investigate the use of lightweight processes in combination with MPI-based inter-process communication. In this subsection, we present the PPR-MT algorithm (*threaded* PPR-M) that improves on PPR-M by using POSIX threads [11] for both computation and inter-process communication.

PPR-MT begins the pre-processing phase by partitioning a web graph and allocating partitions to computing nodes as done in PPR-M. After that, each node will load a portion of web graph into memory, the same as in PPR-M. The difference is that a node may create a number of threads for cooperative computing. The synchronization process between threads is done via shared memory within the node, while synchronization between nodes is still done by MPI. Pseudo-code of the PPR-MT algorithm in shown in Fig. 3.

PPR-MT (i, L_i, I_i, O, p, n)

1: load file L_i, I_i, and O into main memory
2: V_i : source PageRank vector, is initialized to $\left[1/n\right]_{n\times 1}$
3: V_i' : target PageRank vector, is initialized to $\left[0\right]_{\frac{n}{p}\times 1}$
4: *score* : a temporary score
5: **Create threads**
6: While all pages do not converge
7: For each record $l \in L_i$ **will be assigned to a thread**
8: $score = 0$
9: If $l.dest_id$ converges then
10: $V_i'[l.dest_id] = V_i[l.dest_id]$
11: Else
12: Compute all $l.src_id$ as $\frac{V_i[l.src_id]}{O[l.src_id]}$ and add to *score*
13: $V_i'[l.dest_id] = (\alpha \times score) + \frac{(1-\alpha)}{n}$
14: Store V_i' in V_i
15: Synchronize V_i' with other processes and also store in V_i
16: **Join threads**
17: Report V_i' as local PageRank vector

Fig. 3. The parallel PageRank algorithm using mixed MPI and threads

Before computation, the algorithm loads I_i, O, and all or part of L_i into memory to reduce disk I/O and improve CPU utilization (line 1). In lines 5 and 16, threads are created and terminated, respectively. During iterative computation, an unloaded thread will be assigned a record of L_i for rank score computation (line 7). Inter-thread communication occurs via shared memory used for accessing/storing the PageRank scores (line 10 and 13); while inter-process synchronization between nodes occurs after each iteration (line 15) via MPI.

The architecture of the algorithm is depicted in Fig. 4. It consists of three components: allotment, computation, and merge. The Allotment component performs pre-processing, including transforming the web graph into the link structure, in-degree, and out-degree (*L, I, O*) files, partitioning them, and allocating partitions to computing nodes. The Computation component invokes each node (i.e., processor-*i*) to compute rank scores for its partition. At any node, multiple threads collaborate in the computation. Finally the Merge component merges the computed scores from all computing nodes to obtain the global scores as output.

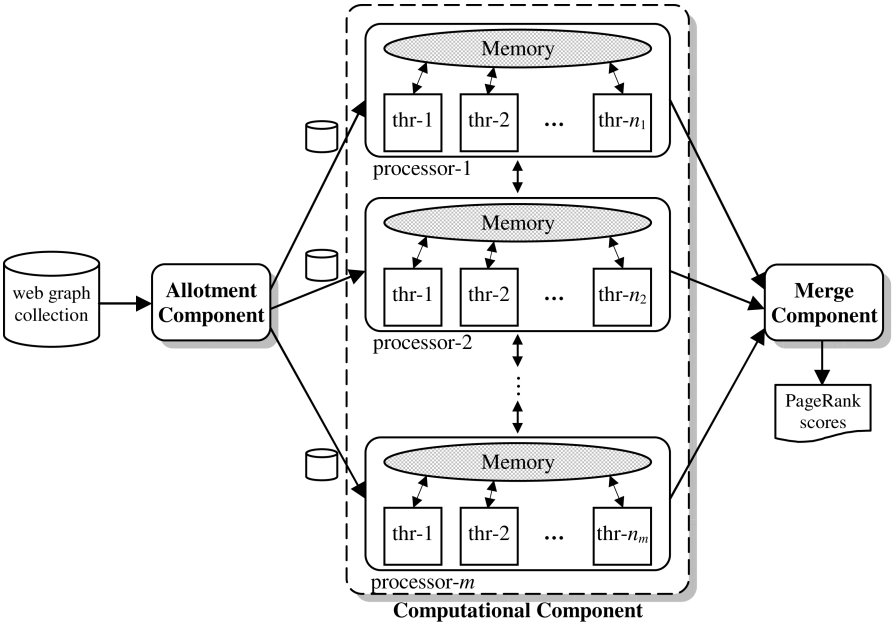


Fig. 4. An architecture of PPR-MT algorithm

4 Experiments and Results

In this section we present the experimental configuration, results obtained, and a discussion of the experimental results.

4.1 Experimental Setup

Environment and Configuration: The algorithm was tested on two clusters of x86-based SMP computers. The first cluster (WSC cluster) consists of dual 3.2GHz Intel quad-core processors, 4GB of main memory, and 120GB SATA disk in each node. The second cluster (MAEKA cluster [7]) consists of hosts with dual AMD Opteron240 processors, 3GB main memory, and 72GB SCSI disk. All hosts in both clusters run the Linux operating system and are inter-connected via switched Gigabit Ethernet. In our implementation, we use the MPICH library, version 1.2.7, and

standard POSIX thread library. Experiments were run under unloaded conditions, without competition from other computing tasks in the cluster.

Data Setup: The test data is a subset of the web graph compiled from the Stanford WebBase Project [16], hereafter termed SF-Graph, consisting of approximately 28 million pages and 227 million hyperlinks. We also synthesize another larger and denser web graph based on the first one to investigate scalability, termed Syn-Graph. Syn-Graph contains approximately 62.6 million pages and 1.37 billion hyperlinks.

4.2 Evaluation Results and Discussion

The main objective of our experiments is to study the performance of the PageRank algorithms as a function of cluster size, threads per compute node, and size of the web graph. We measured the total time needed for the PageRank computation to converge using a relative tolerance of 0.0001, varying the number of compute nodes and threads per node. Due to limitations in the amount of time available on the clusters used, we limited the number of threads per machine in PPR-MT experiments to 2, 4, 8, and 16 threads. We repeated each experiment at least 3 times and averaged the run times.

Table 1. The average run time of PPR-M (1 thread per machine) and PPR-MT algorithms

			Average run time (seconds)				
		# of mach	1thr/mach	2thrs/mach	4thrs/mach	8thrs/mach	16thrs/mach
WSC cluster	SF-Graph	1	266.40252	126.32589	85.35699	83.34633	**83.26815**
		2	214.84687	113.87603	80.89533	79.93591	**77.96629**
		4	152.33151	104.42615	84.71894	68.88086	**67.00578**
	Syn-Graph	1	1750.74109	1186.18676	986.33184	**924.34139**	1120.25558
		2	435.94806	275.22307	182.84255	119.10821	**82.63472**
		4	346.88886	188.56568	108.77058	67.92434	**40.10643**
MAEKA cluster	SF-Graph	1	420.38131	219.53706	173.33427	**120.43113**	130.17395
		2	285.86809	174.88750	136.59907	110.10362	**100.15860**
		4	209.13478	137.59722	124.17826	95.84211	**81.45755**
		8	148.58302	109.82293	93.88049	**74.50150**	76.56715
	Syn-Graph	1	2342.90540	**2009.79438**	2210.43035	3229.86019	12551.50322
		2	1836.33356	1314.93663	**992.73288**	1043.49289	1572.52810
		4	677.85948	378.02183	262.63831	**227.48226**	276.16426
		8	622.81540	330.72717	218.10580	**178.85681**	190.38320

Table 1 shows the average total run times required for both SF-Graph and Syn-Graph data. The upper half of the table shows results using 1-4 machines in the WSC cluster for each web graph. The bottom half show results for experiments using 1-8 nodes from the MAEKA cluster.

The column "1 thr/mach" (1 thread per machine) gives average times using the PPR-M algorithm. The other columns give average times using multiple threads and the PPR-MT algorithm. Using the pseudo-code shown in Fig. 3, the total time needed in each experiment consists mainly of the time for PageRank vector initialization and thread creation (lines 2-5), the adaptive PageRank computation (lines 7-13), and vector synchronization between processors and/or machines (line 15). In each row of Table 1, the best time is shown in bold text. For example, the best run time for Syn-Graph data using 1 machine on the WSC cluster is 924.34 seconds, decreasing to 40.11 seconds for runs utilizing 4 machines.

Table 2 summarizes the best run times from Table 1, and gives a speedup factor for cluster-based run times relative to the single machine case. The results in Table 2 show that the performance, in terms of speedup as the number of nodes increases, is very poor for the smaller SF-Graph data. This indicates that the communication time required for costly PageRank vector synchronization is relatively high, reducing efficiency of processor utilization.

For the larger and denser Syn-Graph, the results are more encouraging: we obtain a speedup of 23.05 and 8.83 using 4 machines in the WSC and MAEKA clusters, respectively. This suggests that the processors are better utilized in the PageRank computation part of overall process. This suggests that it may be more cost effective to invest in more computing resources for cluster farms when we need to compute the PageRank of a very large web graph, such as those on commercial search engines, and using the proposed mixed model algorithm to incorporate multi-threading.

For both clusters, the speedup when increasing from 1 to 4 nodes is greater than the rate of increase in number of nodes. Normally, when increasing from 1 to 4 nodes the best one can hope for is a 4-fold speedup, yet the experimental times show an 8.83 and 23.05 fold acceleration. The unexpectedly high increase in performance is due to improved memory utilization. As the number of nodes increases, the size of each node's data partition decreases. This can increase the percentage of data held in physical memory and reduce or eliminate paging. Utilizing more threads (and more CPU core) per node does not reduce the size of a node's data partition and, conversely, can actually increase contention for memory access. No super-linear speedup was observed as a function of increasing threads per node.

Table 2. The best performance speedup as a function of machines utilized

		WSC cluster		MAEKA cluster	
	# of mach	Best time	Speedup	Best time	Speedup
SF-Graph	1	83.26815	1.00000	120.43113	1.00000
	2	77.96629	1.06800	100.15860	1.20240
	4	67.00578	1.24270	81.45755	1.47845
	8	-	-	74.50150	1.16149
Syn-Graph	1	924.34139	1.00000	2009.79438	1.00000
	2	82.63472	11.18587	992.73288	2.02450
	4	40.10643	**23.04721**	227.48226	**8.83495**
	8	-	-	178.85681	11.23689

5 Conclusion

We investigate the use of cluster technology for parallelization of page rank calculations as a solution to the challenging problem of rapid growth in the Internet page data that must be scored by a page rank algorithm. In this paper, we present the PPR-M and PPR-MT algorithms that efficiently run on SMP based clusters. The first algorithm uses simple message passing for inter-process communication while the second algorithm combines the standard thread library for inter-thread communication with MPI for cluster communications. Both algorithms exploit the power of SMP based clusters to compute the rank scores of a large-scale web graph in parallel. Our experiments show encouraging results, speeding the computational process up to 23 times using 4 machines, compared to base run times on a single machine.

In future work, we plan to explore some web graph partitioning algorithms for better load balancing of the computation between the compute nodes. We will also investigate way to reduce the communication overhead of PageRank synchronization, as well as study the convergence rate for accelerating the algorithm.

Acknowledgments. We would like to thank to all anonymous reviewers for their comments and suggestions. We also thank to Dr. James Edward Brucker for his reading of the final version of this paper. The research is funded by the Kasetsart University Research and Development Institute (KURDI).

References

1. Chen, Y., Gan, Q., Suel, T.: I/O-efficient techniques for computing PageRank. Proceedings of the 11th International Conference on Information and Knowledge Management (2002)
2. Gleich, D., Zhukov, L., Berkhin, P.: Fast parallel PageRank: a linear system approach. Technical Report, Yahoo! Research Labs (2004)
3. Haveliwala, T.H.: Efficient computation of PageRank. Technical Report, Stanford University (1999)
4. Kamvar, S.D., Haveliwala, T.H., Golub, G.H.: Adaptive methods for the computation of PageRank. Technical Report, Stanford University (2003)
5. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Exploiting the block structure of the web for computing PageRank. Technical Report, Stanford University (2003)
6. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. Journal of the ACM (JACM), Vol. 46 (1999) 604-632
7. MAEKA: Massive Adaptable Environment for Kasetsart Application. Available source: http://maeka.ku.ac.th (2003)
8. Manaskasemsak, B., Rungsawang, A.: Parallel PageRank computation on a gigabit PC cluster. Proceedings of the International Conference on Advanced Information Networking and Applications (2004)
9. MPI library: Massive Passing Interface. Available source: http://www-unix.mcs.anl.gov/mpi (2006)
10. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. Technical Report, Stanford University (1999)

11. POSIX Threads Programming. Available source: http://www.llnl.gov/computing/tutorials/pthreads (2006)
12. Ross, S.M.: Introduction to probability models. 8th Edition, Academic Press (2003)
13. Rungsawang, A., Manaskasemsak, B.: Parallel adaptive technique for computing PageRank. Proceedings of the 14th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (2006)
14. Sankaralingam, K., Sethummadhavan, S., Browne, J.C.: Distributed PageRank for P2P systems. Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (2003)
15. Shi, S., Yu, J., Yang, G., Wang, D.: Distributed page ranking in structured P2P networks. Proceedings of the International Conference in Parallel Processing (2003) 179-186
16. The Stanford WebBase Project. Available source: http://www-diglib.stanford.edu/~testbed/ doc2/WebBase (2004)

A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments

George V. Iordache, Marcela S. Boboila, Florin Pop,
Corina Stratan, and Valentin Cristea

Computer Science Department, University Politehnica of Bucharest, Romania
{george, simona}@rogrid.pub.ro, {florinpop, corina, valentin}@cs.pub.ro

Abstract. The paper describes a solution to the key problem of ensuring high performance behavior of the Grid, namely the scheduling of activities. It presents a distributed, fault-tolerant, scalable and efficient solution for optimizing task assignment. The scheduler uses a combination of genetic algorithms and lookup services for obtaining a scalable and highly reliable optimization tool. The experiments have been carried out on the MonALISA monitoring environment and its extensions. The results demonstrate very good behavior in comparison with other scheduling approaches.

Keywords: grid computing, scheduling, load-balancing, genetic algorithms.

1 Introduction

Grid computing developed in recent years in response to challenges raised by complex problems solving and resource sharing in collaborative, dynamic environments. In grid computing, load-balancing plays an essential role, in cases where one is concerned with optimized use of available resources. A well-balanced task distribution contributes to reducing execution time for jobs and to using resources, such as processors, efficiently, in the system. On the other hand, the problem of scheduling heterogeneous tasks onto heterogeneous resources is intractable, thus making room for good heuristic solutions. We denote heterogeneous tasks as tasks that have different execution times, memory and storage requirements.

Various strategies for scheduling have been developed, in order to achieve optimized task planning in distributed systems. Researchers have directed their studies towards static schedulers [20][25][27], in which the assignment of tasks to processors and the time at which tasks start execution are determined a priori [14]. Static strategies cannot however be applied in a scenario where tasks appear a-periodically, and the environment undergoes various state changes. In dynamic scheduling techniques, which have been widely explored in literature [10][14][17][19][29], tasks are allocated dynamically at their arrival.

Furthermore, to-date research on the subject has been focused on both centralized and decentralized scheduling approaches. In centralized scheduling

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1234–1251, 2006.
© Springer-Verlag Berlin Heidelberg 2006

algorithms [14][17][27][29], a single processor collects the load information in the system and determines the optimal allocation. Decentralized algorithms [4][5][22][26] come with reliable solutions for robust systems, at the expense of high communication costs. The centralized control is substituted in distributed approaches by an increased level of decision-making authority for the nodes involved in running the scheduling algorithm.

Genetic algorithms have been largely used for the task allocation problem [14][17][27][29]. The successful results obtained by means of GAs have proven their robustness and efficiency in the field. Research has been done recently, particularly in the area of hybrid algorithms, which use problem-specific knowledge to speed up the search or lead to a better solution [14]. In a novel approach, Wu et al. [27] focus on a thorough exploration of the search space by means of an incremental fitness function and a flexible representation of chromosomes. Moreover, the optimization of scheduling via GAs using the load balancing performance metric has been a key concern for genetic research [17][27][29].

This paper presents SAGA ("Scheduling Application using Genetic Algorithms"), a decentralized solution for task scheduling in heterogeneous environments. The paper is structured as follows: Section 2 is a general presentation of the SAGA features. Section 3 describes the structure and functionality of the proposed system. Section 4 introduces the main implementation issues. We describe and comment on the experimental results in the 5th section. Section 6 contains conclusions and directions for future research.

2 General Presentation of the SAGA Features

In our scheduling scheme, tasks may arrive simultaneously and resources may dynamically join or leave the system. Aspects of heterogeneity of tasks and processors are also considered in [17], which reports results of simulated experiments. In our work, we present a simulation study, and supplement it with experimental results obtained in existing monitoring and job execution platforms. For experiments, the MonALISA monitoring platform and its extensions were employed [12][16].

Another major accomplishment of this research is the migration towards a decentralized scheduler by means of lookup services. Using this, we overcome one of the main drawbacks of centralized schedulers, which is the lack of robustness in realistic scenarios. Decentralized scheduling approaches have focused on partitioning the task sets or the computation resources into subparts and on running the algorithm on each of them [4][5][22][26]. The results generally indicate high overloads and low balancing, which lead to scarce performance. We also directed our research towards speeding up the convergence of genetic algorithms by using multiple agents (see Section 3.2) and different populations to schedule sets of tasks. The experimental results show that the number of generations necessary for the algorithm to converge is significantly reduced. The use of multiple initial search points in the problem space favors a high probability to converge towards a global optimum. Combined with the lookup services, this approach offers a solution to high scalability and reliability demands.

3 SAGA Model

3.1 System Anatomy

A schematic view of the SAGA system is presented in Figure 1. Users submit *Scheduling requests*. A near-optimal schedule is computed by the *Scheduler* based on the *Scheduling requests* and the *Monitoring data* provided by the *Grid Monitoring Service* (MonALISA). The schedule is then sent as a *Request for task execution* to the *Execution Service*. The user receives feedback related to the solution determined by the scheduler, as well as to the status of the executed jobs in the form of the *Schedule and task information*. Furthermore, the system can easily integrate new hosts in the scheduling process, or overcome failure situations by means of the *Discovery Service*.

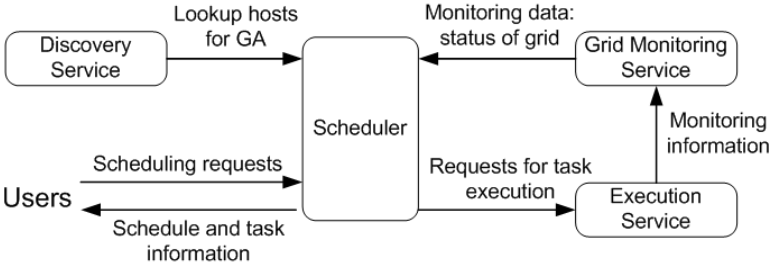


Fig. 1. Anatomy of the SAGA system

The *Scheduling request* contains a description (in XML) of the tasks to be scheduled. This way, a user may ask for the scheduling of more than one task at a time. Various parameters have been taken into account for task description:

- resource requirements (CPU Power, Free Memory, Free Swap)
- restrictions (deadlines), and
- priorities.

The assignment of a task on a given computing node is conditioned by meeting the resource requirements [6]. We have focused our study on classes of independent tasks, as described in [13], which avoids communication costs due to dependencies. We have built a model based on a real scenario in which groups of tasks are submitted by independent users, to be executed on a group of nodes.

The characteristics and functionalities of the services that interact with the scheduler - *Grid Monitoring Service*, *Execution Service* and *Discovery Service* - are further detailed.

Grid Monitoring Service. The *Grid Monitoring Service* has the specific purpose to obtain real-time information in a heterogeneous and dynamic environment such as a Grid. We are using the MonALISA [16] distributed service system in conjunction with ApMon [12]. ApMon is a library that can be used to send any

status information in the form of UDP datagrams to MonALISA services. MonALISA provides system information for computer nodes and clusters, network information for WAN and LAN, monitoring information about the performance of applications, jobs or services. It proved its reliability and scalability in several large scale distributed systems[16].

We have deployed the existing implementation of the MonALISA Web Service Client to connect to the monitoring service via proxy servers and obtain data for the genetic algorithm. Task monitoring is achieved by means of a daemon application based on ApMon. This daemon provides information regarding task status parameters on each node (amount of memory, disk and CPU time used by the tasks). The up-to-date information offered by the Grid Monitoring Service leads to realistic execution times for assigned tasks, as shown by experimental results.

Execution Service. Given its capability to dynamically load modules that interface existing job monitoring with batch queuing applications and tools (e.g. Condor [24], PBS [11], SGE [8], LSF [28]), the *Execution Service* can send execution requests to an already installed batch queuing system on the computing node to which a particular group of tasks was assigned. Sets of tasks are dynamically sent on computing nodes in the form of a specific command. The time ordering policy established in the genetic algorithm for tasks assigned on the same processor is preserved at execution time.

Discovery Service. Lookup processes are triggered by the *Discovery Service* and determine the possibility of achieving a decentralized schedule by increasing the number of hosts involved in the genetic scheduling. The apparition or dysfunction of agents in the system can easily be intercepted, resulting in a scalable and highly reliable optimization tool. If one agent ceases to function, the system as a whole is not prejudiced, but the probability of reaching a less optimal solution for the same number of generations increases.

3.2 Functionality Aspects

In our approach, the grid nodes are part of a group, according to their specific function, as described below:

Scheduling Group. The computers in this group run the scheduling algorithm. They receive requests for tasks to be scheduled, and return a near-optimal schedule according to the genetic algorithm.

Execution Group. The computers in this group execute the tasks that have been previously scheduled and assigned to them.

The use of real-time monitoring information about the *Execution Group* by means of MonALISA implies that computers in the Grid may well take part in both scheduling and execution, provided that they have enough resources to support the load produced by the scheduling algorithm. In the case of less stringent deadline requirements, the solution of using the computers for both scheduling and execution is valid, and employs a reduced number of resources.

The computers in the *Scheduling Group* are either *brokers* or *agents*. The brokers are designated to receive user requests, and the agents run the genetic algorithm in order to find a near-optimal solution.

One major objective of our research is to fit in real scenarios of grid scheduling. In this context, SAGA can well be mapped on two scenarios that may appear. In the first scenario, the Scheduling requests are sent from remote sites by means of a portal to the computers in the Scheduling Group. In this case, the brokers provide the input for our scheduling algorithm, but do not actually run it. In the second scenario, all the computers in the Scheduling Group run the scheduling algorithm and are open to Scheduling requests at the same time. The execution of the genetic algorithm is not limited to only a part of the computers in the Scheduling Group, and that translates into an increase in their functionality.

The first approach is preferable when users are somewhere remote and would rather not overload their computer with running the scheduling algorithm. In the second case, the resources available are not preferentially treated and therefore the number of resources used is reduced. Depending on the situation, the intended objectives or cost limits, one of the two scenarios may be adopted.

In both scenarios, the groups of tasks are stored in local queues on each of the agents. The genetic algorithm starts when the queue is not empty and either a predefined waiting period of time has passed or there are enough tasks to complete a chromosome.

4 Genetic Algorithm

4.1 Chromosome Encoding

In our chromosome encoding, each gene is a pair of decimal values (T_j, P_i), indicating that task T_j is assigned to processor P_i, where j is the index of the task in the batch of tasks and i is the processor id. This representation has been regarded in literature [14][27] as being efficient and compact, and having reduced computational costs. Each individual in a GA population consists of a vector of genes. Two tasks assigned on different processors can be processed in parallel if they do not require the same resource in exclusive mode. However, the tasks assigned to the same processor must be executed in the specified order [14].

The initial population is stochastically initialized by placing each task on a processor. In order for the search space to be thoroughly explored, the chromosomes are initialized using different random number generators. Various probabilistic distributions are used by each agent for population initialization (Poisson, Normal, Uniform, Laplace).

4.2 Genetic Operators

A thorough analysis of the genetic operators and their way of application is essential for an exhaustive search in the problem space. The danger of imposing too many restrictions may arise, on the account of arbitrarily human interference [3][27].

We have experimentally tested three types of crossover: single-point crossover, two-point crossover and uniform crossover. Single-point crossover functioned better in most of the cases and delivered the best scheduling solutions. The likelihood of crossover being applied is typically between 0.6 and 1.0 [3]. If crossover is not applied, offspring are produced simply by duplicating the parents. This gives each individual a chance of passing on its genes without the disruption of crossover.

All new chromosomes have a certain probability of being affected by mutation. The search space expands to the vicinity of the population by randomly altering certain genes. The result is the tendency to converge to a global rather than to a local optimum [3][9][21][23]. We have opted for an adaptive mutation operator, which gave better results than the usual static ones. Research has been carried out in the field of dynamic operators, but such studies usually focus on either increasing or decreasing the probability of mutation during a run [3]. In our experiments, we modeled a linear increase in mutation rate when the population stagnates, and a decrease towards a predefined threshold when population fitness varies and the search space has moved to a vicinity.

4.3 Fitness Function

The fitness function measures the quality of each individual in the population and therefore is essential for the evaluation of chromosomes. With scheduling objectives in mind, fitness functions have been developed in order to achieve algorithm convergence on a near-optimal solution.

The main objective of our research is to obtain a well-balanced assignment of tasks to processors. That implies low overall completion time by the processors, which means minimization of maxspan [4][29]. Maxspan is defined as:

$$t_M = \max_{1 \le i \le n} \{t_i\} \ , \tag{1}$$

where n is the number of processors and t_i is the total execution time for processor i, computed as the sum of processing times for all tasks assigned to this processor in the current or previous schedules:

$$t_i = t_i^p + t_i^c = \sum_{j=1}^{T_i} (t_{i,j}) \ . \tag{2}$$

We have considered T_i to be the total number of tasks assigned to processor i for execution and $t_{i,j}$ the running time of task j on processor i. t_i^p is the execution time for previously assigned tasks to processor i, and t_i^c is the processing time for currently assigned tasks.

A mapping of maxspan in the [0, 1] interval leads to the use of the factor $\frac{1}{t_M}$ in fitness computing [4][29]. We optimized this factor for a more efficient search of well-balanced schedules. In our approach, we determined that reducing the difference between the minimum and the maximum processing times is a factor worth considering, in terms of load-balancing optimization. Therefore, one factor introduced for fitness computation is:

$$f_1 = \frac{t_m}{t_M} = \frac{\min_{1 \le i \le n}\{t_i\}}{\max_{1 \le j \le n}\{t_j\}} \ , \quad 0 \le f_1 \le 1 \ . \tag{3}$$

The factor converges to 1 when t_m approaches t_M, and the schedule is perfectly balanced.

The second factor considered for fitness computation is the average utilization of processors:

$$f_2 = \frac{1}{n} \sum_{i=1}^{n} \frac{t_i}{t_M} \ , \quad 0 \le f_2 \le 1 \ . \tag{4}$$

Zomaya [29] pointed out the efficiency of this factor for reducing idle times by keeping processors busy. Division by maxspan is pursued in order to map the fitness values to the interval [0, 1]. In the ideal case, the total execution times on the processors are equal and equal to maxspan, which leads to a value of 1 for average processor utilization.

Another factor considered in our research is represented by the fulfillment of the imposed restrictions. In a realistic scenario, task scheduling must meet both deadline and resource limitations (in terms of memory, cpu power). In deadline computation for a task t, we must consider the execution times for each of the tasks assigned to run before task t on the same processor. These tasks occupy a previous slot on the respective processor in our encoding of a chromosome [14][15]. The fitness factor is subsequently defined as:

$$f_3 = \frac{T_s}{T} \ , \quad 0 \le f_3 \le 1 \ , \tag{5}$$

where T_s denotes the number of tasks which satisfy deadline and computation resource requirements, and T represents the total number of tasks in the current schedule.

This factor acts like a contract penalty on the fitness. Its value varies reaching 1 when all the requirements are satisfied and decreases proportionally with each requirement that is not met. The chance of being selected for future generations is reduced due to the penalty introduced by this factor. Still, the schedule is not dismissed, but may be used in subsequent reproduction stages that lead to valid chromosomes.

The fitness function applied in our research consists of the contribution of the factors presented:

$$F = f_1 \times f_2 \times f_3 = \left(\frac{t_m}{t_M}\right) \times \left(\frac{1}{n} \sum_{i=1}^{n} \frac{t_i}{t_M}\right) \times \left(\frac{T_s}{T}\right) \ , \quad 0 \le F \le 1 \ . \tag{6}$$

5 Experimental Results

The experimental cluster was configured with 11 nodes, named $P1, P2, \ldots, P11$, which represent heterogeneous computing resources with various processing capabilities and initial loads. The input tasks are typical cpu-intensive computing

programs. Default parameters for the genetic algorithm were established at 0.9 for crossover rate and 0.005 for mutation rate threshold. The values were experimentally determined, in order to widely and thoroughly explore the search space.

Furthermore, the reproduction operators applied were single point crossover and adaptive mutation, as described in Section 4.2. We used the roulette wheel selection method to choose individuals who will survive in the next generation [7][17][29].

5.1 Algorithm Convergence

We studied the improvement achieved by the algorithm over several generations. The metric used for performance measurements is load-balancing, a performance attribute of high demand in distributed environments.

In this work, the load-balancing of the system was investigated according to the following definition:

$$L = 1 - \frac{\Delta}{\overline{u}}, 0 \leq L \leq 1 \ , \tag{7}$$

where the symbols are defined below:

$\overline{u}$ is the average processor utilization of the system:

$$\overline{u} = \frac{1}{n} \sum_{i=1}^{n} u_i \ , \tag{8}$$

where u_i denotes the utilization rate of processor i and n denotes the number of processors in the system. The processor utilization rate is defined as:

$$u_i = \frac{t_i}{t_M} = \frac{t_i}{\max_{1 \leq i \leq n}\{t_i\}} \ . \tag{9}$$

In the formula above, t_i and t_M have the same values as in Section 4.3.

Furthermore, Δ in the load-balancing formula denotes the square deviation of u_i from the mean $\overline{u}$:

$$\Delta = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (u_i - \overline{u})^2} \ . \tag{10}$$

Load-balancing in the system converges to 1 when the tasks are equally distributed among the processors such that the processing times are approximately equal relative to one another and equal to maxspan. In this case, the dispersion of processor utilization rates from the average tends to zero:

$$u_i \rightarrow \overline{u} \Rightarrow \Delta \rightarrow 0 \Rightarrow L \rightarrow 1 \ . \tag{11}$$

For the purpose of this experiment, we use a GA scheduler with one agent. Figure 2 shows the dispersion of load-balancing in the system over 500 generations, as well as the average and maximum load-balancing.

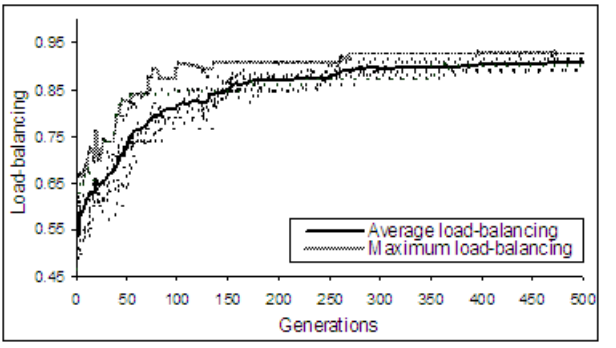


Fig. 2. Convergence of load-balancing in the system over 500 generations

The convergence study consisted of ten distinct experiments. The points in Figure 2 represent values obtained at different generations. With lower numbers of generations, the results are spread over a larger interval and usually achieve low values, since there is not enough time for the algorithm to cover a larger search space. For the set of tasks used, little improvement is obtained over 200 generations; therefore, this limit was considered a stopping condition of the GA with the following experiments performed on the same set of tasks.

5.2 Decentralization

Convergence Speed-up. Next, the influence of decentralization on the load-balancing performance metric was analyzed (Figure 3). Experiments were carried out with one, two, three and four agents, with the data averaged over ten runs for each case. The increase in the number of agents gives the best results in terms of convergence speed-up when the GA is run with fewer generations, less than 100 for the tasks set we used. This is to be expected, since the GA is a stochastic algorithm, in which every run is initialized randomly. Using more agents means that the genetic algorithm has more start points in the search of a near-optimal

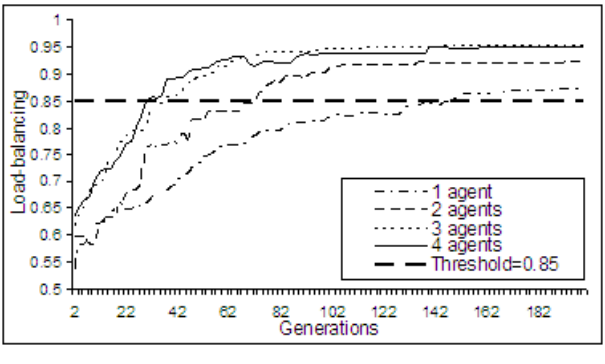


Fig. 3. Load-balancing with various number of agents

solution. As the number of agents increases, the improvement rate decreases. In our case, the benefit of four agents over three is almost negligible because the convergence is already established at low generations, below which the chances of finding optimal individuals are very small. At higher generations, the cost of employing multiple agents is not justified, because the algorithm has already converged.

Table 1. Speed-up increase rates for various decentralization levels at $\alpha = 0.85$ load-balancing threshold

Experiment no.	Speed-up rates relative to previous Exp. (%)	Speed-up rates relative to Exp. 1 (%)	Relative execution time decrease (%)
Exp. 1	-	-	-
Exp. 2	50.66	50.66	5.46
Exp. 3	48.64	74.66	14.10
Exp. 4	21.05	80	-0.55

Table 1 shows the speed-up increase rate for achieving a near-optimal solution at the $\alpha = 0.85$ load-balancing threshold for different numbers of agents. The *speed-up relative to the previous experiment* is computed as:

$$s_i^\alpha = \frac{g_{i-1}^\alpha - g_i^\alpha}{g_{i-1}^\alpha} \times 100\% \ , \tag{12}$$

with g_i^α representing the number of generations at which the threshold is reached in the experiment with i agents, and g_{i-1}^α is the number of generations at which the threshold is reached in the experiment with $i-1$ agents. With 2 agents, the increase rate is high (50.66%), meaning that the generations were reduced to less than half. In the case of three agents, a rate almost as high (48.64%) is obtained, while a low increase rate of 21.05% for the four-agent experiment shows that the improvement in speed-up is minimal.

The *speed-up relative to Exp. 1* is determined using the same formula described above, with $g_{i-1}^\alpha = g_{pc}^\alpha$, where g_{pc}^α is the number of generations at which the α threshold is reached in an experiment with one agent (pseudo-centralized).

Furthermore, the value of the *relative execution time decrease* is:

$$td_i^\alpha = \frac{t_{pc}^\alpha - t_i^\alpha}{t_{pc}^\alpha} \times 100\% \ , \tag{13}$$

with t_i^α representing the time at which the threshold is reached in the experiment with i agents, and t_{pc}^α is the time at which the threshold is reached in the experiment with one agent.

The method used results in a substantial decrease in execution time for the scheduling algorithm. For our experimental configuration, an optimum is achieved with three agents, when a decrease in execution time of over 14% relative to the one-agent experiment is observed. The communication time costs are higher when running the algorithm with four agents, and consequently the

performance is diminished. It is also worth noting that not only execution performance, but also the probability of obtaining a global optimum is improved by employing multiple agents with various start points in the solution space.

Scalability and Robustness. We also performed experiments to study the ability of the system to resist to failure situations. Figure 4 illustrates a scenario in which the system starts to function with two agents and one of them dysfunctions after the algorithm runs for 40 generations. For the comparison, we also represented the estimated evolution with one agent and two agents, averaged over 10 runs. Although in the initial phases the algorithm performed well with two

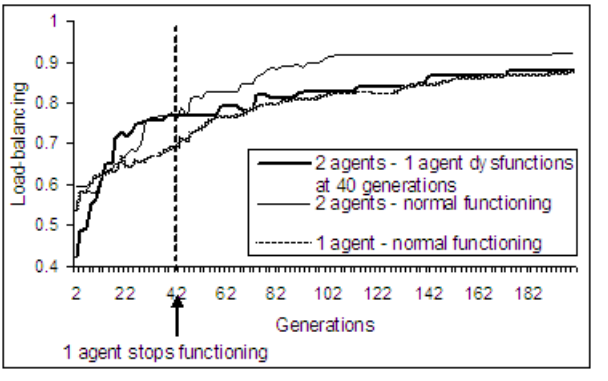


Fig. 4. Reliability: 1 agent out of 2 dysfunctions after 40 generations

agents, achieving load-balancing values similar to those obtained during normal functioning, the improvement is reduced after one agent ceases to function. The convergence rate decreases, which nevertheless leads to slightly better results than in the case of one agent functioning normally. The performance is visibly impaired, but the system resists failure and continues to perform scheduling.

5.3 Estimated Times Versus Real Execution Times

The accuracy of estimated times is essential for the quality of the schedule. We want the computed processing times to closely approximate the real execution times. This is especially important in hard real-time scheduling problems, in which missing deadlines is extremely problematic [14].

Figure 5 provides a comparison between the estimated time and the real processing time achieved during the experiment. A configuration of the algorithm with 200 generations was used for the computation of estimated times. Job running was carried out with PBS [1] and job monitoring information was achieved by means of MonALISA Service and its extensions (ApMon and MonALISA Client). The error of approximation is determined according to the method

[1] The experiments were pursued using TORQUE Resource Manager 2.0.

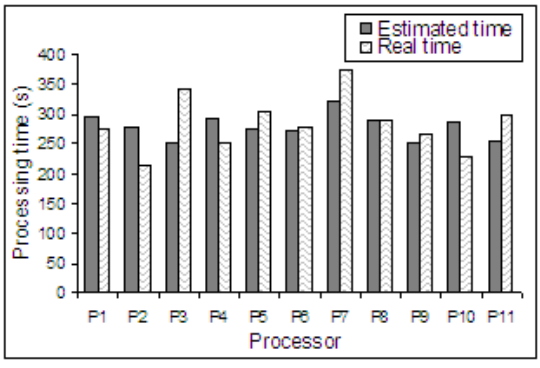


Fig. 5. Estimated and real processing times on each processor

described below. The relative deviation of the estimated time from the real time for processor i is noted δ_i and determined as:

$$\delta_i = \frac{t_i^r - t_i^e}{t_i^r} \ , \tag{14}$$

where t_i^r represents the real processing time obtained by running the jobs in the execution system, and t_i^e is the estimated processing time as calculated by the genetic algorithm.

The approximation error is further determined as:

$$\epsilon = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \delta_i^2} \ , \quad 0 \leq \epsilon \leq 1 \ . \tag{15}$$

We obtained an average error value over 10 runs of approximately $\epsilon = 0.16$. The scheduling policy employing real monitoring data from the grid environment is therefore a viable one, providing good estimations, with an accuracy of about 84%.

5.4 Comparison of Various Scheduling Methods

Four different methods were compared with respect to load-balancing performance. In the first stage, we use the Opportunistic Load Balancing (OLB) algorithm [1][2] in order to assign 100 tasks to the eleven computation resources. The following experiments test heuristic strategies based on genetic algorithms. The experiments were carried out with 100 generations. The purpose was to demonstrate that very good results can be obtained even at reduced numbers of generations, which implies low computation costs.

Opportunistic Load Balancing Algorithm. In the OLB policy, a task is picked from the queue and assigned to the processor where it is expected to start first. In order to determine the fittest processor for task allocation, we must first

estimate the time at which already assigned tasks will finish execution on each processor. The expected completion time for processor is computed as follows:

$$tc_i = \sum_{j=1}^{T_i} t_{i,j} \ , \quad 1 \leq i \leq n \ , \tag{16}$$

where T_i and $t_{i,j}$ have the same values as described in Section 4.3.

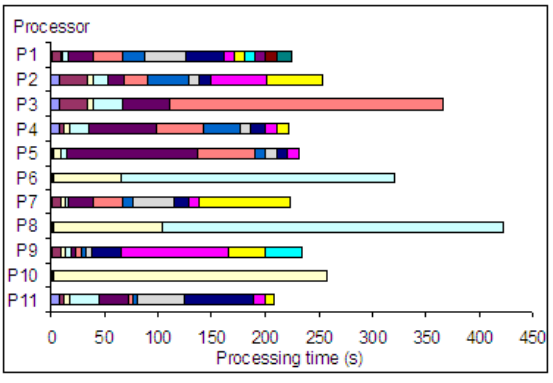


Fig. 6. Task assignment with the Opportunistic Load Balancing Algorithm

We can now compute the estimated start time for a newly arrived task k:

$$ts_k = \min_{1 \leq i \leq n} \{tc_i\} = \min_{1 \leq i \leq n} \left\{ \sum_{j=1}^{T_i} t_{i,j} \right\} . \tag{17}$$

The ordering of tasks in the queue influences the balancing level in the system. Improved solutions are obtained if we use a heuristic, which would reorder the task set on the expenses of algorithm complexity [4]. For the task set used, the strategy obtains an average processor utilization of about 0.64 on a [0:1] interval and a load-balancing of about 0.754 (Figure 6). The least loaded processor ($P1$) has over 200 s (about half the total execution time) additional idle time relative to the overloaded processor $P8$. It is clear that heterogeneity of tasks is a highly influential factor and leads to low values for load-balancing. The large tasks disposed on processors $P3$ and $P8$ heavily overloaded these resources.

Centralized Genetic Algorithm. This heuristic method searches for an optimal assignment of tasks in order to achieve reduction of total execution time and highly homogeneous loads on processors. In the second experiment, we tested the centralized genetic algorithm on the same set of input tasks (Figure 7). The load-balancing obtained is with 0.2% lower in comparison with the OLB policy. The result is indeed satisfactory, considering the reduced number of generations run by the algorithm. Moreover, an important maxspan reduction of 33 s was achieved, as well as an increase in the average processor utilization of 6%.

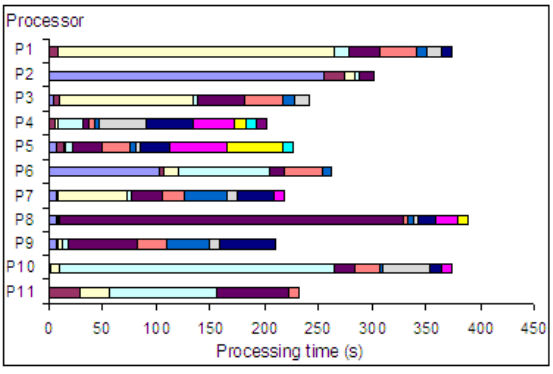


Fig. 7. Task assignment with the Centralized Genetic Algorithm

With the OLB strategy, the previously submitted task introduces a load into the system which must be taken into account when assigning the next task. Therefore, monitoring information is collected at the allocation of each task, in order to determine the processor with the earliest start time. Although the complexity of this method is reduced in comparison to genetic algorithms, the need to monitor data results in high execution times. On the other hand, the genetic algorithm allocates the whole group of tasks simultaneously, without introducing load variations in the system.

Decentralized Non-cooperative Genetic Algorithm. In experiment 3, the decentralized non-cooperative genetic algorithm was studied. Based on the experimental results described in Section 5.2, the decentralized algorithm was run with three agents. No cooperation mechanism is applied among agents; therefore, individuals are not interchanged at different stages during the algorithm. The genetic algorithm starts from different search points, and the fittest individual is chosen during the final step, from the results determined by all agents.

The distribution of tasks after 100 generations can be seen in Figure 8. Compared to the previous experiments, load-balancing, average processor utilization and maxspan performance metrics are all improved. The load-balancing has a 3% increase relative to the centralized algorithm, and an average processor utilization improvement of 4%. The maximum execution time has been reduced by 18.5 s. All metrics indicate that resources are better utilized in this configuration, although processors such as $P3$ or $P6$ are still idle 0.47% and 0.44% of the total execution time, respectively, to the detriment of overloaded processors like $P9$, $P4$ and $P1$.

Decentralized Cooperative Genetic Algorithm. The fourth experiment analyses the metrics previously discussed in a decentralized scheduling scenario, in which a cooperative genetic strategy has been employed. The cooperative characteristic implies optimal individuals interchange in order to speed up convergence. The input task set is the same as previously, as well as the level of decentralization (3), and the number of generations (100). Figure 9 illustrates the schedule obtained. An essential improvement of all metrics has been achieved:

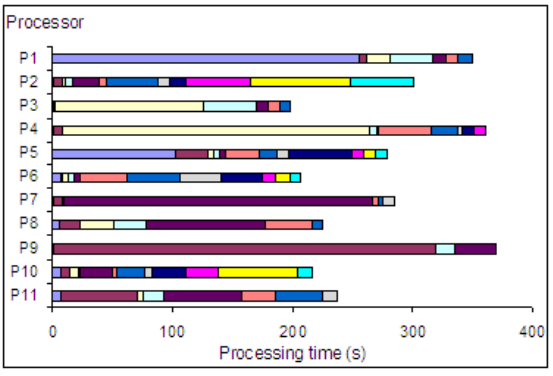


Fig. 8. Task assignment with the Decentralized Non-cooperative Genetic Algorithm

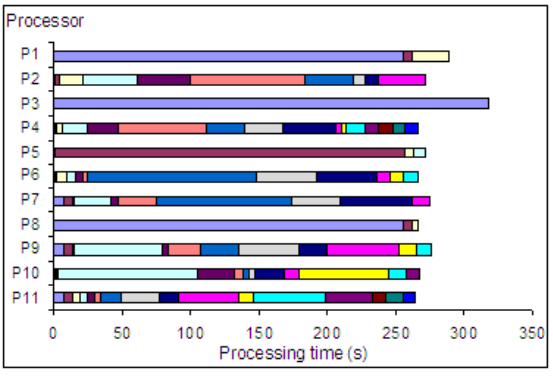


Fig. 9. Task assignment with the Decentralized Cooperative Genetic Algorithm

Table 2. Performance metrics for various scheduling strategies

	Load-balancing (%)	Average processor utilization (%)	Maxspan (s)
OLB	0.754	0.64	421
Centralized GA	0.752	0.70	388
Decentralized Non-Cooperative GA	0.78	0.74	369.5
Decentralized Cooperative GA	0.94	0.86	317

16% for the load-balancing and 12% for the average processor utilization. The maxspan has been reduced by 52.5 s. The processor $P3$ is executing only the largest task in the system; therefore, the result obtained is very close to the best possible distribution.

Table 2 is a synthesis of the comparative results discussed above, and highlights the solution quality obtained by each of the four strategies employed.

Previous work on distributed strategies usually proposes scheduling scenarios in which the execution resources or the set of tasks are divided among agents. In a

distributed multi-agent mechanism, as described in [4], each agent has up-to-date information only on its neighboring agents, which limits the scheduling effect. Therefore, load-balancing at the global level of the grid is reduced, as compared to a centralized strategy. As the experimental results show, our approach achieves global load-balancing, while also ensuring scalability and reliability.

6 Conclusions

In grid environments, various real-time applications require dynamic scheduling for optimized assignment of tasks. This paper describes a genetic scheduling approach, which features a decentralized strategy for the problem of task allocation. We carry out our experiments with complex scheduling scenarios and with heterogeneous input tasks and computation resources. We improve upon centralized genetic approaches with respect to scalability and robustness. Our experimental results show that the system continues to work well even when agents dysfunction. The use of lookup services also facilitates rapid integration of new agents that arise in the system. Moreover, the strategy of starting the search from multiple initial points in the problem space is favorable for obtaining global convergence and avoiding premature blocking in a local optimum. Also, significant convergence speed-up is achieved by means of the cooperative scheduling strategy, although there is a trade-off in terms of implied communication costs.

We compare the performances of the Decentralized Cooperative Genetic Algorithm with three other strategies: OLB, Centralized GA and Decentralized Non-cooperative GA. It is shown that the algorithm clearly outperforms these methods. Decentralization and cooperation provide significantly better results of load-balancing and average processor utilization increase, as well as of total execution time minimization.

Furthermore, instead of employing simulated scenarios, we have validated our research in real-time environments by utilizing existing monitoring and job execution systems. The experiments show a high level of accuracy in the results obtained.

Future investigation would involve the extension of the algorithm towards classes of dependent tasks, as well as the incorporation of new features into the current framework (e.g. Recovery Service for task backup and migration). Also, the slow nature of the GA method and node dynamics in a Grid may lead to less suitable results for estimates of processing times. The solution would combine grid monitoring with prediction of status for the grid nodes [18].

References

1. Armstrong, R., Hensgen, D., Kidd, T.: The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. Procs. of the 7th IEEE HCW, pp. 79-87, 1998.
2. Braun, R.D. et al.: A Comparison Study of Static Mapping Heuristics for a Class of Meta-tasks on Heterogeneous Computing Systems. Procs. of the 8th HCW, pp. 15-29, 1999.

3. Beasley, D., Bull,D., Martin, R.: An overview of genetic algorithms: Part 2, research topics. University Computing, 15(4):170-181, 1993.
4. Cao, J. et al.: Grid load balancing using intelligent agents. Future Generation Computer Systems special issue on Intelligent Grid Environments: Principles and Applications, 2004.
5. Csaji, B., Monostori, L., Kadar, B.: Learning and cooperation in a distributed market-based production control system. Procs. of the 5th IWES, pp. 109-116, 2004.
6. Heymann, E., Fernndez, A., Senar, M.A., Salt J.: The EU-CrossGrid Approach for Grid Application Scheduling. LNCS, Vol. 2970, pp. 17-24, 2004.
7. Hou, E., Ansari N., Ren, H.: A genetic algorithm for multiprocessor scheduling. IEEE TPDS, 5(2):113-120, 1994.
8. Gentzsch, W.: Sun grid engine: Towards creating a compute power grid. Procs. of CCGrid'2001, pp. 35-36, 2001.
9. Goldberg, D.: Genetic algorithms in search, optimization and machine learning. Addison-Wesley, 1989.
10. Greene, W.: Dynamic load-balancing via a genetic algorithm. 13th IEEE International Conference on Tools with Artificial Intelligence, pp. 121-129, 2001.
11. Henderson, R.: Job scheduling under the portable batch system. Procs. of the JSSPP'95, LNCS, vol. 949, Springer, pp. 279-294, 1995.
12. Legrand, I.: End user agents: extending the intelligence to the edge in distributed service systems. Fall 2005 Internet2 Member Meeting, 2005.
13. Maheswaran, M. et al.: Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. JPDC, 59:107-131, 1999.
14. Mahmood, A.: A hybrid genetic algorithm for task scheduling in multiprocessor real-time systems. SIC Journal, 9(3), 2000.
15. Manimaram, G., Murthy, C.: An efficient dynamic scheduling algorithm for multiprocessor real-time systems. IEEE TPDS, 9(3):312-319, 1998.
16. Newman, H. et al.: Monalisa: A distributed monitoring service. CHEP 2003, 2003.
17. Page, A., Naughton, T.: Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. Procs of the 19th IPDPS, pp. 189a.1-189a.8, 2005.
18. Phinjareonphan, P., Bevinakoppa, S., Zeephongsekul, P.: An Algorithm to Predict Reliability of a Grid Node. 11th ISSAT International Conference on Reliability and Quality in Design, pp. 37-41, 2005.
19. Prodan, R., Fahringer, T.: Dynamic scheduling of scientific workflow applications on the grid: a case study. Procs. of ACM SAC 2005, pp. 687-694, 2005.
20. Ramamritham, K.: Allocation and scheduling of precedence related periodic tasks. IEEE TPDS, 4:382-397, 1993.
21. Schaffer, J., Eshelman, L.: On crossover as an evolutionarily viable strategy. R.K. Belew and L.B. Booker, Procs. of the 4th ICGA, pp. 61-68, 1991.
22. Seredynski, F., Koronacki, J., Janikow, C.: Distributed scheduling with decomposed optimization criterion: Genetic programming approach. LNCS; Vol. 1586. Procs. of the 11th IPPS/SPDP'99 Workshops, Springer, 1999.
23. Spears, W.: Crossover or mutation? L. Darrell Whitley, editor, Foundations of Genetic Algorithms, Morgan Kaufmann, pp. 221-237, 1993.
24. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. Grid Computing: Making The Global Infrastructure a Reality, Fran Berman, Anthony J.G. Hey, Geoffrey Fox, John Wiley, 2003.
25. Theys, M. et al.: Mapping Tasks onto Distributed Heterogeneous Computing Systems Using a Genetic Algorithm Approach. John Wiley, 2001.

26. Weichhart, G., Affenzeller, M., Reitbauer, A., Wagner, S.: Modelling of an agent-based schedule optimisation system. Procs. of the IMS International Forum, 2004.
27. Wu, A. et al.: An incremental genetic algorithm approach to multiprocessor scheduling. IEEE TPDS, 15(9):824-834, 2004.
28. Zhou, S.: Lsf: load sharing in large-scale heterogeneous distributed systems. Procs. of the Cluster Computing, 1992.
29. Zomaya, A., Teh, Y.-H.: Observations on using genetic algorithms for dynamic load-balancing. IEEE TPDS,12(9):899-911, 2001.

Solving Scheduling Problems in Grid Resource Management Using an Evolutionary Algorithm

Karl-Uwe Stucky, Wilfried Jakob, Alexander Quinte, and Wolfgang Süß

Forschungszentrum Karlsruhe GmbH
Institute for Applied Computer Science
P.O. Box 3640, 76021 Karlsruhe, Germany
{uwe.stucky, wilfried.jakob, alexander.quinte,
wolfgang.suess}@iai.fzk.de

Abstract. Evolutionary Algorithms (EA) are well suited for solving optimisation problems, especially NP-complete problems. This paper presents the application of the Evolutionary Algorithm GLEAM (**G**eneral **L**earning and **E**volutionary **A**lgorithm and **M**ethod) in the field of grid computing. Here, grid resources like computing power, software, or storage have to be allocated to jobs that are running in heterogeneous computing environments. The problem is similar to industrial resource scheduling, but has additional characteristics like co-scheduling and high dynamics within the resource pool and the set of requesting jobs. The paper describes the deployment of GLEAM in the global optimising grid resource broker GORBA (**G**lobal **O**ptimising **R**esource **B**roker and **A**llocator) and the first promising results in a grid simulation environment.

1 Resource Allocation in Grid Computing Environments

The task of resource scheduling is a well-known problem in industrial production planning. Machine resources are allocated to single jobs of a production process. For economic reasons, it is crucial to optimise this allocation in terms of workload, completion time and makespan. A number of projects deal with the scheduling problem in industrial production by utilising Evolutionary Algorithms (see for example [1]).

A production process can be described as a workflow. The simplest form contains jobs in a single chain, each job depending on its predecessor's completion only. Workflows are also capable of describing more complex dependencies, for example, jobs waiting for the completion of more than one predecessor or loops with sub-workflows processed for a given number of times. Parallel operation is possible when sub-workflows are independent of each other and their resources can be scheduled in parallel.

In computer science a similar problem arises in parallel and distributed computing and, in recent years, in grid computing. The latter field utilises computer resources, called grid resources in this context, across domain borders in networks. There are different types of grid resources, like computing power, storage capacity, software, or even external devices. The vision of a World Wide Grid – in analogy to the World Wide Web – requires resource scheduling similar to industrial production planning. A user's application corresponds to a production order and can be described as a workflow with properties comparable to production processes [2]. The elementary grid jobs require grid resources instead of machine resources.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1252 – 1262, 2006.
© Springer-Verlag Berlin Heidelberg 2006

On the other hand, there are differences to production planning that challenge developers of grid resource management systems:

- The jobs often require more than one resource, for example software licences or storage in addition to computing power. These resources must be co-allocated.
- Workflows may often be dynamic, which means that parts of the workflows are dependent on parameter values that are not known in advance.
- The "grid orders", namely the applications, are not known far in advance. They are started continuously by users, and immediately require suitable resources.
- The state of the grid changes more dynamically than in industrial production, when new resources are added or when resources fail or are otherwise removed.

The differences named here show that resource management in a grid is a highly dynamic process. It requires fast and flexible methods for resource allocation that can do continuous replanning. In [3] scheduling systems for grid computing are classified as either queuing systems, which schedule current jobs in queues with heuristic methods or planning systems, which schedule current and future jobs. The latter often introduce optimisation methods to the field of grid resource management.

This paper describes how the EA GLEAM [4, 5], a combination of Evolutionary Strategy [6] and real-coded Genetic Algorithms [7], is integrated in the global optimising grid resource broker GORBA [8]. The term "global" denotes that all waiting jobs are rescheduled with all resources available in the broker's scheduling domain. A new optimisation procedure is triggered by the dynamic changes described above.

The next section will give an overview of solutions for grid resource management tasks in related work and it will explain why GLEAM has been chosen as the optimisation method for the resource broker GORBA. Section 3 will start with a short presentation of the grid environment, into which GORBA is embedded, and proceed with a detailed description of GORBA and how the EA GLEAM is deployed for global optimisation. In Section 4 first results obtained with GORBA in a test environment that simulates the grid, the applications, and the grid resources will be given. A conclusion and an outlook on future work will complete this paper in the last section.

2 Methods for Resource Allocation in Grids

Continuous replanning and optimising of the allocation of n resources to m application jobs is an NP-complete problem [9]. Most systems are queuing systems or use simple heuristics for planning. In [10] and [11] taxonomies of grid resource management systems are presented. They show approaches to planning grid resource allocation, such as an economic method in Nimrod/G [12]. [13] follows another economic approach based on contracts between providers and users and focuses on costs as a critical link between both grid participant groups. Many aspects of grid resource management are also explained in [14], including overviews of important systems and highlighting of Quality-of-Service aspects. Promising planning methods deploy simulated annealing [15], tabu search [16] or genetic algorithms. [17] presents an overview of these methods and also discusses combinations.

EAs have already been applied successfully in the field of grid job scheduling. Aggarwal et al. [18] schedule jobs where direct acyclic graphs model dependencies. In

[19] a Genetic Algorithm is described for application-level scheduling that minimises the average runtime of jobs. The approach in [20] combines a Genetic Algorithm with historical scheduling results to generate start populations. Furthermore, Di Martino and Mililotti [21] deploy a Genetic Algorithm in a parallel version.

The GORBA tool discussed in this paper shall be integrated in the Campus Grid [22] (a local grid at our research centre). The objective is to establish schedules that optimise completion times, costs, and workload. The decision for GLEAM as the central optimising method was made for several good reasons:

- The references given above demonstrate that EAs have shown their applicability to scheduling tasks in industrial production and in grid computing before.
- GLEAM is a universally applicable tool for NP-complete problems. This universality has been demonstrated by numerous applications [4, 5, 23, 24].
- Some genetic operators in GLEAM are well suited for combinatorial optimisation.
- Although not utilised in GORBA up to now, GLEAM has a parallelisation option based on an integrated neighbourhood model.
- A more elaborated optimisation tool called HyGLEAM [25] has shown the usefulness of hybridisation with other methods (HyGLEAM is a memetic algorithm with local search methods). [17] has also emphasised the improvements by deploying hybrid methods.

3 Resource Brokering with GLEAM

A grid environment concept and the architecture of GORBA were described in earlier papers [8, 26]. Nevertheless, a brief introduction shall be given (see Fig. 1). An application and resource management system is embedded between two main interfaces:

- The Grid Application Interface to users and to administrators.
- Interfaces to third-party grid middleware, such as Globus services.

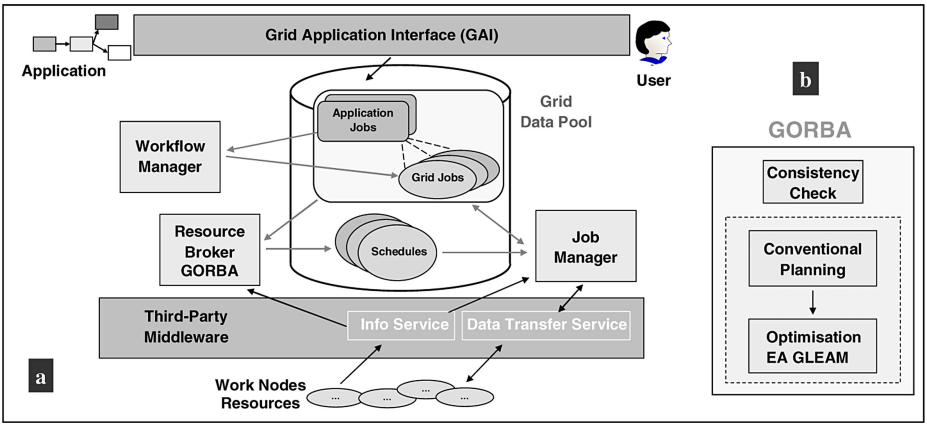


Fig. 1. Architecture of an application and resource system for grid environments (**a**) and the coarse architecture of the global optimising resource broker GORBA (**b**)

Applications are described as workflows, called application jobs, on an XML basis. The XML scheme expands GADL (**G**rid **A**pplication **D**efinition **L**anguage) of the Fraunhofer Gesellschaft [27] with requirements like maximum costs or time restrictions. The elementary units of an application job are called grid jobs. They describe their resource requirements, again XML-based (expanding the **G**rid **R**esource **D**efinition **L**anguage GRDL, a part of GADL). A capacity list is added to each resource that holds availability times and costs. A workflow manager decomposes application jobs into static workflows, which can be handled by the resource broker GORBA.

The connection to the third-party middleware is established at two points. XML resource data are provided by information services. The information is regularly updated. On the other side, a job manager submits and monitors jobs according to a schedule that was optimised by GORBA.

The global optimising resource broker GORBA actually is a hybrid method in the current version, because a conventional planning phase with simple heuristics precedes the EA optimisation with GLEAM. It is also capable of co-allocation. At present, CPU, software resources, and devices are handled.

GORBA starts with a consistency check of application and grid job data, for example, the smallest possible costs and makespan must be smaller than the user given cost and time limits. For conventional planning, the following strategies have been defined so far for the generation of an ordered list of grid jobs: shortest due date of the application job first, shortest user-estimated processing time of the application job first, and shortest user-estimated processing time of the single grid job first.

Currently, there are also three different strategies for the assignment of grid resources: always cheapest resource first, always fastest resource first, and cheapest or fastest resource first according to the requirements of the application job. The resource prices per hour and performance factors given by the administrator are used to calculate comparable resource prices.

The conventional planning phase is an important part of the whole brokering procedure, because

- firstly, it is a simple form of resource planning and therefore it can be the one and only method that is actually deployed, when time constraints do not allow for more sophisticated planning and
- secondly, it provides first results for the following phases. Especially for EAs, its solutions can be part of the start population, besides randomly generated individuals. It makes GORBA an elitist tool in this case, as the EA will likely improve solutions and never lead to a smaller quality.

Replanning is triggered when new jobs arrive, waiting jobs are cancelled, changes in the resource pool occur or execution times differ from planned times above a given threshold.

For the rest of this section GLEAM and its integration in GORBA as an optimising tool shall be explained. A thorough description of the features of GLEAM can be found in [5]. In GLEAM the analogues to genes are actions, which appear in application-specific types that are defined by a set of integer and/or real parameters. A population is built up of chains of actions (comparable to chromosomes) as individuals. Action chains can be arbitrarily subdivided into segments, which are the bases of some of the genetic operators. Action-based mutation operators allow for parameter

alterations, new parameterisations, or positional changes. Similar operators are available on the segment level and, moreover, for shifting segment borders, merging or separating segments. A special segment operator inverts the action order of a segment. Regarding recombination, GLEAM provides two types of crossover, 1-point and n-point operators, based on segment boundaries.

The gene model developed for GORBA assigns an action to each grid job. Resources are specified as integer parameters of an action, with the parameter value being an index in a list of alternatively usable resources for this job. There are as many parameters as resources have to be allocated to a particular job. The order of the actions in the chain is relevant, since it codes directly for the scheduling sequence. Thus, an individual represents a complete schedule of all jobs.

Positional mutation is essential for the GORBA application. It varies the main characteristic of the schedule, the grid job order. Besides coding the grid job order by the action sequence, which is a purely combinatorial scheme, a so-called permutation-based gene model was tested. It codes the grid job order as an integral position number, which is subjected to parameter mutation. It was found to have a lower performance than the combinatorial scheme (see also section 4).

The process of deciding between the combinatorial and the permutation-based scheme is a good example of the flexibility of GLEAM in adapting to specific application requirements. GLEAM provides interfaces to build application-specific action types and to establish application-specific gene models. Suitable genetic operators can be chosen from a built-in pool of operators or added to the system as necessary.

The evaluation of an individual requires the simulated execution of its schedule. Start times are assigned to the grid jobs one by one in the sequence given by the action chain. The earliest possible time is chosen depending on the availability of the resources specified for that grid job and the completion times of preceding grid jobs if any. The evaluation process focuses on completion times and costs of the application jobs. The former depends on CPU resource characteristics and the job, the latter simply on each resource required. Pareto optimisation has been discussed, but was discarded, as within an automatic process a single solution is required rather than a set of comparable good results (Pareto front). The overall fitness is calculated as a weighted sum of the relative makespan of all application jobs, the relative completion time per application job, the workload, and the relative cost per application job. It normalises the values of all criteria so that the resulting sum has to be maximised. Use of relative values instead of absolute ones makes the evaluation independent of the actual values of cost or time frames. This automatic readjustment of the fitness scale is necessary for an automated process of optimisation. As an example the equation for the relative job costs jc_{rel} is given here:

$$jc_{rel} = \frac{jc_{act} - jc_{\min}}{jc_{\max} - jc_{\min}} \tag{1}$$

jc_{act} is the actual costs, jc_{min} is calculated using the prices of the cheapest suitable resources, and jc_{max} is the minimum of the price using the most expensive resources and the user-given cost maximum. The quotient normally ranges between 0 (best) and 1 (expensive). Excessive costs are evaluated separately by penalty functions, which are also defined for time delays. Other terms of the evaluation function with relative

quantities are defined in an analogous manner. Additionally, the user can set a priority value for the relative assessment of time and costs for his application job.

4 Experimental Results

According to [28], a suitable workload model either is an extract of a mix of log files from different computer clusters or a synthetic workload with jobs of arbitrary size and time requirements. Another alternative exists between simple benchmark workloads, which are well suited for comparisons of different scheduling algorithms and more realistic and, therefore, detailed ones. In this case it was decided for synthetic and as realistic as possible benchmarks. They are constructed such, that the rates of dependencies and parallelism cover a wide range. For this purpose, these two rates are used and defined as follows for workflows without cycles:

$$rate_{par} = \frac{jobsum}{cplen} \text{ [29]} \qquad rate_{dep} = \frac{dep}{dep_{\max}} \tag{2}$$

with: *jobsum* sum of all grid job processing times.

cplen processing time of the longest critical path of all application jobs based on user estimated values.

dep sum of all dependencies of all grid jobs. Every grid job, which must be completed before the execution of a particular job, is counted.

$dep_{\max}$ the maximum possible number of dependencies is calculated for odd numbers as $(gjobs^2\text{-}1)/4$ and for even numbers of grid jobs as $gjobs^2/4$. The proofs for these calculations can be found in [30].

Based on this, the set of benchmark workloads is constructed as shown in Table 1, where *p* and *d* denote small rates of parallelism and dependencies, while *P* and *D* stand for greater ones. As the number of grid jobs to be scheduled is another significant parameter for the complexity of the workload, it is set to one of four fixed values for each scenario of parallelism and dependencies, for reasons of comparability. This results in different numbers of application jobs, as shown in Table 1. Despite our efforts to control and describe the complexity of a benchmark workload, there still is a property of great influence that is hard to determine prior to a schedule: the sufficient availability of suited resources. This limits the significance of the two rates and to some extend explains the differences of the results discussed later. The cost and time frames of the application jobs were chosen such that conventional planning violates at least one of them in all cases, but one, as it is shown in the *# Viol.* column of Table 1.

There are two questions to be answered by the experiments:

1. Can GLEAM overcome these cost and time overruns in general? The answer is yes, but the time required to reach convergence (measured as 500 generations without any offspring acceptance) varies between 10 minutes and 10 hours on a workstation equipped with an AMD Athlon 64 X2, 2.2 GHz CPU using only one of the two cores. This gives an impression of the varying complexity for the EA.
2. As the available planning time is limited to a few minutes to meet the needs of frequent rescheduling, the second question is: to what an extent can GLEAM overcome these overruns within a certain amount of time that has been chosen to be three minutes? The answer will be discussed in the rest of this section.

The *conv.p.* column of Table 1 shows the fitness to be maximised obtained by the best conventional planning without application of the penalty functions, which drastically reduce the fitness values. These are a measure of the overall quality of the schedule that should be improved by GLEAM. The next column (*min*) contains the minimum fitness values for violation-free schedules, followed by the GLEAM results, which are all averaged over 100 runs. The success rate is based on the runs exceeding the minimum fitness and also shown in Fig. 2, left part. For 50 to 100 grid jobs violation-free schedules can be expected in almost all cases. With greater numbers of grid jobs, the achievement of this goal can not be guaranteed. This is due to two reasons: more grid jobs to be scheduled do not only mean a greater complexity of the task, but also longer simulation times and, therefore, fewer evaluations within the given time frame of three minutes. The amount of evaluations drops from 1.2 million for 50 grid job schedules to 380 thousand for 200 grid jobs. This clearly limits the amount of processible schedules within a given time frame and computing power. But also in the cases without 100% success the overall quality of the schedules could be improved (see column *conv.p.* compared to column *GLEAM*) and the amount of violations could be reduced (not shown in the table).

Table 1. Important properties of the benchmark workloads and the results obtained by conventional planning (*conv.p.*) and the EA (*GLEAM*). The numbers of cost (c) and due time (t) violations of conventional planning are shown in column *# Viol.* The three fitness columns show the best fitness of conventional planning calculated as if no penalty function had to be applied, the minimum fitness (*min*) required for penalty-free schedules, and the EA results. *Rel. Std. Dev.* denotes the relative standard deviation of the EA fitness values obtained. The success rate indicates the fraction of EA runs with a better quality than the minimum fitness (*min*).

Acronyms	*rate* see (2)		# of Jobs		# Viol.	Fitness			Rel. Std.	Success
	par	*dep*	appl.	grid	c/t	conv.p.	min	GLEAM	Dev. [%]	[%]
pd	12.2	2.6	19	50	8/0	0.27	0.22	0.34	1.9	100
			29	100	12/0	0.23	0.19	0.30	2.2	100
			41	150	13/0	0.20	0.21	0.28	2.8	100
			52	200	12/0	0.23	0.20	0.29	2.7	100
Pd	39.1	1.8	8	50	3/1	0.08	0.17	0.26	2.5	100
			15	100	3/1	0.07	0.13	0.20	5.4	100
			23	150	5/1	0.12	0.15	0.16	18.2	67
			32	200	8/3	0.10	0.18	0.13	14.8	5
pD	12.4	3.6	7	50	0/0	0.13	0.13	0.16	4.8	100
			15	100	2/0	0.10	0.11	0.15	2.4	100
			21	150	0/2	0.06	0.10	0.11	14.4	70
			29	200	0/2	0.06	0.12	0.11	12.3	28
PD	25.0	10.3	5	50	1/0	0.04	0.18	0.20	1.5	100
			11	100	5/1	0.06	0.17	0.17	15.9	71
			16	150	5/2	0.08	0.15	0.10	23.9	5
			21	200	4/0	0.13	0.14	0.16	4.4	98

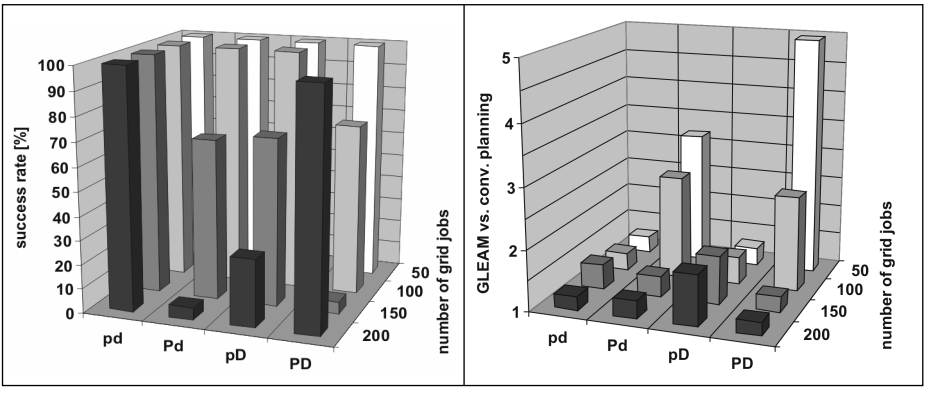


Fig. 2. On the left the fractions of GLEAM runs without cost and time overruns (success rate from Table 1) are shown for each benchmark scenario, while the improvement factor achieved as compared to conventional planning is displayed on the right

The right part of Fig. 2 reflects the improvement achieved by GLEAM as compared to the unpunished best conventional planning by comparing the fitness values of the "conv.p." and the GLEAM columns. At least, a small improvement was always possible, even with large amounts of grid jobs. The relative standard deviations of the fitness values obtained from the GLEAM runs indicate a reliable quality for those scenarios where GLEAM can avoid cost and time overruns.

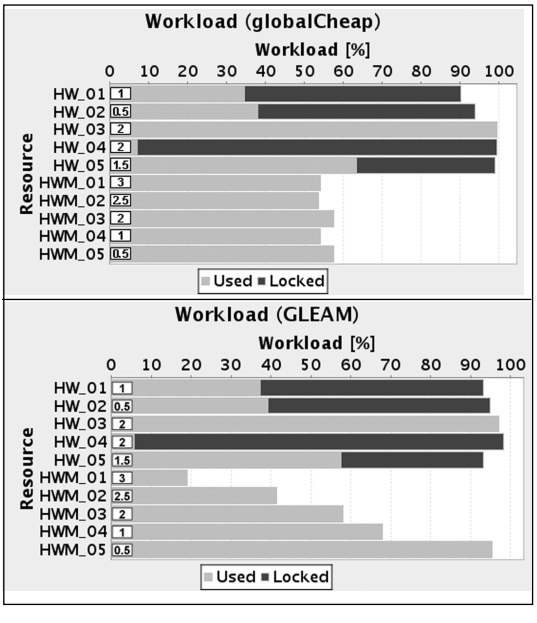


Fig. 3. Comparison of the workloads obtained from GLEAM (lower part) and the conventional planning variant using the cheapest resource first (upper part). The dark bars indicate times of resource unavailability. Used SW resources are omitted here.

To give an idea of the achievable improvements, Fig. 3 compares resource workloads of GLEAM and that conventional planning variant, which tries to use the cheapest resources first. There are two groups of alternatively usable resources (HW_*i* and HWM_*i*) with different prices as shown in the small boxes. GLEAM changes the schedule such that the cheaper resources are much more used, if possible.

These results were obtained using the combinatorial gene model described in section 3 and the standard version of GLEAM

without any application-specific add-ons. The only tuned strategy parameter was the population size, which was varied between 200 and 600. Experiments showed that the permutation-based gene model does not work well and, therefore, it is not used any further. This result underlines the importance of the two segment-based mutations of segment inversion and movement, as the effect of gene movements is covered by the permutation parameter.

5 Conclusion and Future Work

The concept of GORBA for optimised resource management in a dynamic grid environment was presented. It was shown that, despite the short time available for planning, an EA-based approach yields significant improvements of the schedules delivered by simple conventional planning heuristics. The investigated planning scenario was based on an empty grid. But in practice we are faced with replanning tasks, were only comparable small changes in either resources or the work load have to be dealt with. It can be derived from earlier applications of GLEAM that replanning using the knowledge of an old solution is much faster than planning from scratch.

These promising first results are the motivation for further enhancements and development. The gene model will be changed in such a way that the resource selection will be done by the heuristics already used for conventional planning instead of leaving the decision to evolution. So, evolution is responsible for the scheduling sequence only. Furthermore, local searchers will be added for combinatorial optimisation, the hope being that the resulting memetic algorithm will be as successful as its counterpart HyGLEAM was for parameter optimisation [25].

References

1. Nissen, V.: Quadratic Assignment. In: Bäck, T., Fogel, D., Michalewicz, Z. (eds.): Handbook of Evolutionary Computation, Oxford University Press, New York (1997) sect. G9.10.
2. Hoheisel, A., Der, U.: Dynamic Workflows for Grid Applications. Cracow Grid Workshop, 2003.
3. Hovestadt, M., Kao, O., Keller, A., Streit, A.: Scheduling in HPC Resource Management Systems: Queuing vs. Planning. Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) at GGF8, Seattle, WA, USA, June 24, 2003, LNCS 2862, 1-20.
4. Blume, C.: GLEAM - A System for Simulated “Intuitive Learning”. In: Schwefel, H.-P., Männer, R. (eds.): Proc. of PPSN I, LNCS 496, Springer, Berlin (1991) 48-54.
5. Blume, C., Jakob, W.: GLEAM – An Evolutionary Algorithm for Planning and Control Based on Evolution Strategy. Conf. Proc. GECCO 2002, Vol. Late Breaking Papers, (2002)
6. Rechenberg, I.: Evolutionsstrategie '94. Frommann-Holzboog Verlag, Stuttgart - Bad Cannstatt (in German) (1994).
7. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer, Berlin (1992).

8. Jakob, W., Quinte, A., Süß, W., Stucky, K.-U.: Optimised Scheduling of Grid Resources Using Hybrid Evolutionary Algorithms. Proc. 6th Int. Conf. on Parallel Processing and Applied Mathematics, Poznan, PL, 2005, Springer, LNCS 3911 (to be published).
9. Karp, R.M.: Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations, Sympos. Proc., Plenum Press, New York, 85-103 (1972)
10. Ali, A., Anjum, A., Mehmood, A., McClatchey, R., Willers, I.., Bunn, J., Newman, H., Thomas, M., Steenberg, C.: A Taxonomy and Survey of Grid Resource Planning and Reservation Systems for Enabled Analysis Environment. Proceedings of the 2004 International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2004,Wuhan Hubei, P.R. China, Sept. 13th-16th 2004.
11. Krauter, K., Buyya, R., Maheswaran, M.: A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing; International Journal of Software: Practice and Experience (SPE), ISSN: 0038-0644, Vol. 32, Issue 2, 135-164, Wiley Press, USA, February 2002.
12. Buyya, R., Murshed, M., Abramson, D., Venugopal, S.: Scheduling Parameter Sweep Applications on Global Grids: A Deadline and Budget Constrained Cost-Time Optimisation Algorithm. Softw. Pract. Exper. 2005, 35, 491–512.
13. Sample, N., Keyani, P., Widerhold, G.: Scheduling under Uncertainty: Planning for the Ubiquitous Grid. International Conference on Coordination Models and Languages, 2002.
14. Nabrzyski, J., Schopf, J.M., Weglarz, J. (eds.): Grid Resource Management – State of the Art and Future Trends. Kluver Academic Publishers, ISBN 1-4020-7575-8, 2004.
15. YarKhan, A., Dongarra, J.J.: Experiments with Scheduling Using Simulated Annealing in a Grid Environment. Proceedings M. Parashar (Ed.) Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Springer Verlag, Lecture Notes in Computer Science, November 2002 (Volume 2536), 232-242.
16. Glover F., Laguna, M.: Tabu Search. Kluwer Academic Publishers (1997).
17. Abraham, A., Buyya, R., Nath, B.: "Nature's Heuristics for Scheduling Jobs on Computational Grids"; Int. Conf. on Advanced Computing and Communications, 2000.
18. Aggarwal, M., Kent, R.D., Ngom, A.: Genetic algorithm based scheduler for computational grids. IEEE Conference Proceedings (High Performance Computing Systems and Applications, 2005. HPCS 2005), Vol 15-18, p. 209-215, 2005.
19. Gao, Y., Rong, H.Q., Huang, J.Z.: Adaptive grid job scheduling with genetic algorithms. Future Generation Computer Systems, Vol. 21, 2005, 151-161.
20. Song, S., Kwok,Y.-K., Hwang, K.: Security-Driven Heuristics and A Fast Genetic Algorithm for Trusted Grid Job Scheduling. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) – Papers, p. 65a, 2005.
21. Di Martino, V., Mililotti, M.: Sub optimal scheduling in a grid using genetic algorithms; Parallel Computing, 30, 2004, 553-565.
22. Schmitz, F., Schneider, O.: The CampusGrid test bed at Forschungszentrum Karlsruhe. Sloot, P.M.A. (Ed.), Advances in Grid Computing – EGC 2005, Amsterdam, NL, Feb. 14-16, 2005, Lecture Notes in Computer Science 3470, Springer, Berlin, 2005, pp. 1139-1142.
23. Blume, C., Gerbe, M.: Deutliche Senkung der Produktionskosten durch Optimierung des Ressourceneinsatzes. (in German) Automatisierungstechnische Praxis (atp) 36, Oldenbourg, München (1994) 25-29.
24. Jakob, W., Quinte, A., et al.: Opt. of a Micro Fluidic Component Using a Parallel EA and Simulation Based on Discrete Element Methods. In: Hernandez, S., et al.: Computer Aided Design of Structures VII, Proc. of OPTI'01, WIT Press, Southampton (2001) 337-346.
25. Jakob, W.: HyGLEAM – An Approach to Generally Applicable Hybridization of Evolutionary Algorithms. In: Merelo, J.J., et. al. (eds.): Conf. Proc. PPSN VII, LNCS 2439, Springer, Berlin (2002) 527–536.

26. Süß, W., Jakob, W., Quinte, A., Stucky, K.-U.: GORBA: Resource Brokering in Grid Environments using Evolutionary Algorithms. Proc. 17th IASTED Intern. Conference on Parallel and Distributed Computing Systems (PDCS), Phoenix, AZ., 14.-16.11.2005, S. 19-24.
27. Hoheisel, A., Der, U.: An XML-based Framework for Loosely Coupled Applications on Grid Environments; P.M.A. Sloot et al. (Eds.): ICCS 2003, 245-254, Springer-Verlag Berlin Heidelberg, 2003.
28. Tchernykh, A., Ramirez, J.M., Avetisyan, A., Kuzjurin, N., Grushin, D., Zhuk, S.: Two Levels Job-Scheduling Strategies for Computational Grid. Proc. 6th Int. Conf. on Parallel Processing and Applied Mathematics, 2005, Springer, LNCS 3911 (to be published).
29. Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling 5 (2002) 379-394.
30. http://www.iai.fzk.de/~suess/proof_for_gada_paper/

Integrating Trust into Grid Economic Model Scheduling Algorithm*

Chunling Zhu[1], Xiaoyong Tang[2,**], Kenli Li[2], Xiao Han[2], Xilu Zhu[2], and Xuesheng Qi[2]

[1] College of Computer Science & Technology, Huazhong University of Science & Technology, Wuhan 430074, China
[2] School of Computer and Communication, Hunan University, Changsha 410082, China
tang_313@163.com

Abstract. Computational Grids provide computing power by sharing resources across administrative domains. This sharing, coupled with the need to execute distrusted task from arbitrary users, introduces security hazards. This study mainly examines the integration of the notion of "trust" into resource management based on Grid economic model to enhance Grid security. Our contributions are two-fold: First, we propose a trust function which based on dynamic trust changing and construct a Grid trust model based on behavior. Second, we present trust-aware time optimization scheduling algorithm within budget constraints and trust-aware cost optimization scheduling algorithm within deadline constraints. The performance of these algorithms excels that of algorithm without considering trust via theory analysis and simulation experiment.

Keywords: Grid economic model; differential equation; scheduling algorithm.

1 Introduction

In computational Grid, for implementing secure and reliable high performance computing service, the study on how to support security infrastructure for grid is necessary. At present, almost computational Grid and their toolboxes all include certain secure techniques. Security Management Mechanism in Globus consists of GSI [1] and GSS-API. GSI supports many secure techniques, such as, single-sign, credentials, the collaboration between local secure strategy and secure strategy of whole system etc. GSI mainly points to secure the transport layer and application layer in network, and emphasizes on synthesizing present popularly secure techniques to Grid environment.

Although these secure techniques applied to Grid are developing more and more mature, when they are applied to Grid, there are many kinds of restrictions more or less. For example, systems usually require the resources in different management domains can be trusted by each other, users must be all legal, applications are totally harmless,

* Supported by the National Natural Science Foundation of China under Grant Nos. 60273075 and the Key Project of Ministry of Education of China under Grant No.05128.
** Corresponding author.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1263 – 1272, 2006.
© Springer-Verlag Berlin Heidelberg 2006

etc. These restrictions deeply frustrate the scale of Grid, users and applications. Similarly, they may increase the cost of Grid in running itself and implementing services, frustrate the development and applying into factual applications. At first, while actual computational Grid can span several management domains, and in these domains, the very high trust relationship must be supported each other. This is because sometimes for supporting powerful computing, computational resources in different domains usually should be shared, while this kind of resources sharing may lead to illegal users acquire much higher secure level to access to the resources that they have no rights to access to. Under this condition, the security of Grid may be threatened. Secondly, if illegal users run cockhorse in the environment of computational Grid, certain resources in it may be destroyed and all information in it will disappear forever.

For addressing these problems, this paper study a behavior-based trust model to implement the management of dynamical trust level[2-5], and establish the feasible and reliable trust mechanism based on trust function which dynamic express trust changing[4]. The trust model is integrated into Grid economic model which is an efficient Grid resource management and scheduling approach compares with levelized model and abstract owner model [6-10]. This study mainly focuses on the Grid economic model task scheduling based on trust model, and examines trust-aware time optimization scheduling algorithm within budget constraints (TTrust-DBC), trust-aware cost optimization scheduling algorithm within deadline constraints (CTrust-DBC).At last, the performance of these algorithms is excelled the algorithm without considering trust via theory analysis and simulation experiment.

This paper is organized as follows. Section 2, defines the notions of trust, reputation and outlines the trust function. The trust-aware Grid economic scheduling algorithms are presented in Section 3. The performance and the analysis of the proposed algorithms are examined in Section 4. And the conclusions and future work are given in section 5.

2 The Grid Trust Model

2.1 Definition of Trust and Reputation

The notion of trust is a complex subject relating to a *firm belief* in attributes such as reliability, honesty, and competence of the trusted entity. There is a lack of consensus in the literature on the definition of trust and on what constitutes trust management [2]. The definition of trust that we will use in this paper is as follows [2,3]:

> *Trust is the firm belief in the competence of an entity to act as expected such that this firm belief is not a fixed value associated with the entity but rather it is subject to the entity's behavior and applies only within a specific context at a given time.*

That is, the *firm belief* is a dynamic value and spans over a set of values ranging from *very trustworthy* to *very untrustworthy*. The *trust level* (TL) is built on past experiences and is given for a specific context. For example, entity *Y* might trust entity *X* to use its storage resources but not to execute programs using these resources. The TL is

specified for a given time frame because the TL today between two entities is not necessarily the same TL a year ago.

When making trust-based decisions, entities can rely on others for information pertaining to a specific entity. For example, if entity *X* wants to make a decision of whether to have a transaction with entity *Y*, which is unknown to *X*, *X* can rely on the reputation of *Y*. The definition of reputation that we will use in this paper is as follows [2,3]:

> *The reputation of an entity is an expectation of its behavior based on other entities' observations or the collective information about the entity's past behavior within a specific context at a given time.*

2.2 Computing Trust

The trust value of Grid entity *X* for *Y* is defined as following:

$$\begin{cases} \Gamma(X,Y,t) = \alpha \times H(X,Y,t) + \beta \times G(Y,t) \\ \alpha + \beta = 1 \qquad\qquad \alpha \geq 0, \beta \geq 0 \end{cases} \tag{1}$$

The trust relationship at a given time *t* between two entities, for example *X* for *Y*, expressed as $\Gamma(X,Y,t)$ is computed based on the direct relationship at time *t* between *X* for *Y,* expressed as $H(X,Y,t)$, as well as the reputation of *Y* at time *t* expressed as $G(Y,t)$. Let the weights given to direct trust and reputation relationships be α and β, respectively. The trust relationship is a function of direct trust and reputation. If the "trustworthiness" is based more on direct trust relationship with *X* for *Y* than the reputation of *Y*, α will be larger than β.

The direct trust relationship is varied by entities direct interaction, environment changing and the decay of itself as time going on.

1) The direct trust of entity *X* for *Y* will increase or decrease when *X* direct interact with *Y*. For example, if entity *Y* destroys the storage of *X* when *Y* uses *X*'s computing resources, and the trust of *X* for *Y* will decrease because *X* thinks *Y* is dishonesty. But as entity *Y* does no harm for *X*, the trust of *X* for *Y* will increase because *X* thinks *Y* worthy of trust.

2) The changing environment can affect the direct trust of *X* for *Y*. For an instance, if *X* has more idle storage resources, *Y* can access *X*'s storage resource, which equal to *X* trust *Y*. Otherwise the trust of *X* for *Y* will decrease as *X* does not have available storage resources.

3) As time go on, the direct trust itself decays.

Thus the direct trust function can express as following differential equation:

$$\frac{\partial H(X,Y,t)}{\partial t} = \xi D(X,Y,t) + \varphi E(X,Y,t) - \rho H(X,Y,t) \quad H(X,Y,0) = H_0(X,Y) \geq 0 \tag{2}$$

The direct interaction trust function of *X* for *Y* at a given time *t* is $D(X,Y,t)$ and $E(X,Y,t)$ represent the trust function of environment changing between two entities. The

parameter ξ , φ is positive—although we shall consider ξ =0, φ = 0 as a limiting case, ρ represent as decay rate and is positive too. The solution of (2) is:

$$H(X,Y,t) = H_0(X,Y)\exp(-\rho t) + \int_0^t [\xi D(X,Y,s) + \varphi E(X,Y,s)]\exp(-\rho(t-s))ds$$

As an essential unit of Grid entity in Grid economic model, he will try his best to get better brand image [13], namely everyone will increase the trust for him. This is the mainly contribution of his reputation. The other contribution of reputation is the trust recommendation by others and itself decay rate. So Grid entity *Y*'s reputation can express as following equation:

$$\begin{cases} \dfrac{\partial G(Y,t)}{\partial t} = \mu A(Y,t) + \nu B(Y,t) - \delta G(Y,t) & G(Y,0) = G_0(Y) \geq 0 \\ \dfrac{\partial B(Y,t)}{\partial t} = \sum \chi_i [P_i(Y,t) + Q_i(Y,t)] - \gamma B(Y,t) & B(Y,0) = B_0(Y) \geq 0 \end{cases} \quad (3)$$

The brand image trust function of Grid entity *Y* is expressed as $A(Y,t)$, and $B(Y,t)$ is the total trust recommendation by other Grid entities. The total trust recommendation affect by $P_i(Y,t)$ which is the recommendation trust function of $entity_i$ and $Q_i(Y,t)$ which is the environment changing trust function of $entity_i$. Because recommendation trust is based primarily on what other entities say about a particular entity, we introduced the *recommendation trust factor* χ_i to prevent cheating via collusions among a group of entities. Hence, χ_i is a value between 0 and 1, and will have a higher value if the recommender does not have an alliance with the target entity. In addition, we assume that χ_i is an internal knowledge that each entity has and is learned based on actual outcomes. In formula (3), μ , υ is positive and δ , γ is decay rate. The solution of equation (3) is:

$$\begin{cases} G(Y,t) = G_0(Y)\exp(-\delta t) + \int_0^t [\mu A(Y,s) + \nu B(Y,s)]\exp\{-\delta(t-s)\}ds \\ B(Y,t) = B_0(Y)\exp\{-\gamma t\} + \sum \int_0^t \chi_i [P_i(Y,s) + Q_i(Y,s)]\exp\{-\gamma(t-s)\}ds \end{cases}$$

3 Trust-Aware DBC Scheduling Algorithms in Grid Economic Model

3.1 The Related Work of Grid Economic Model DBC Scheduling Algorithms

Grid task scheduling is one of important part in Grid resource management system, the performance of algorithm is the key performance goal to evaluating Grid. The efficient Grid scheduling algorithm mainly including MCT, Min-min, Max-min, Sufferage and so on [11], but all of them are not suitable for Grid economic model resources management and scheduling. The efficient task scheduling algorithm in Grid economic model is time optimization, cost optimization under the limitation of deadline and budget (DBC) [6,7]. These resources scheduling mainly involves processing *N* users and each of them with *M* independent jobs (each with the same task specification, but a

different dataset) on *P* distributed computers where *N***M* is, typically, much larger than *P*. In Grid economic model, each user is constrained by deadline and budget. For computing resources, each is specified with specification performance and unit cost. Generally speaking, higher performance with higher unit cost. The objective of Grid economic model scheduling algorithm is how optimal scheduling *N* users on *P* computing resources and satisfy with the constrains of deadline and budget. Algorithm 1. outlines the time optimization scheduling algorithm which attempts to complete the experiment as quickly as possible, within the budget available (T-DBC) [7].

Algorithm 1. time optimization scheduling algorithm, within time and budget constraints (T-DBC)

1. For each resource, calculate the next completion time for an assigned job, taking into account previously assigned jobs and job consumption rate.
2. Sort resources by next completion time.
3. Assign one job to the first resource for which the cost per job is less than or equal to the remaining budget per job.
4. Repeat the above steps until all jobs are assigned.

The cost optimization scheduling algorithm attempts to complete the experiment as economically as possible within the deadline. A description of the core of the algorithm following Algorithm 2. (C-DBC) [7].

Algorithm 2. cost optimization scheduling algorithm, within time and budget constraints (C-DBC)

1. Sort resources by increasing cost.
2. For each resource in order, assign as many jobs as possible to the resource, without exceeding the deadline.

3.2 Overview of Trust-Aware Grid Economic Model Scheduling

In this section, we present two Grid economic model trust-aware scheduling algorithms to allocate the resources. In these algorithm, the clients belonging to different CDs[2,3] present the requests for task executions, different requests belonging to the same CD may be mapped onto different RDs[2,3]. The trust-aware algorithms presented here are based on the following assumptions: (a) scheduler is organized centrally, (b) tasks are mapped non-preemptively, and (c) tasks are indivisible (i.e., a task cannot be distributed over multiple machines).

Let EET[P_i, f(r_j)] represents the time of executing request r_j task on machine P_i, and ESC[P_i, r_j] be the *expected security cost* if the task of r_j is assigned to machine P_i. The ESC value is a function of the *trust cost* (TC) value obtained from ETS (Table 1) and the task under consideration. Finally, Let ECT[P_i] denotes the *expected completion cost* of machine P_i which is computed as the EET of r_j on machine P_i plus the ESC of r_j on machine P_i. The goal is to assign requests $R_v = \{ r_0, r_1, ..., r_{n-1} \}$ such that $\max_p$ (ECT[P_i]) is minimized where p is the number of machines.

3.2.1 Trust-Aware Time Optimization Scheduling Algorithm (TTrust-DBC)

Algorithm 3. outlines the core of trust-aware time optimization scheduling algorithm. The algorithm can be mainly divided into two phases, the first phase is computing the trust-aware overhead and next is optimal user task scheduling. The algorithm starts by computing the ESC in terms of the trust overhead which is the difference between the r_j requested TL and the offered TL by a machine P_i in RD_k. The trust overhead is an indicator of how well is the trust relationship between an RD and a CD. For example, if the trust overhead is 0, then the two parties completely trust each other. After that, the ECT table is initialized by first scheduling completion time. Then computing each requests of their the lowest completion time under the constrain of budget. Finally find request r_j with the lowest completion time and actually assign all of his task to the machines. For algorithm 3., we use the following equation to calculate the ESC table value**.**

$$S(TC) = EET(P_i, f(r_j)) \times (TC \times 15)/100 \tag{4}$$

Algorithm 3. trust-aware time optimization scheduling algorithm within budget constraints (TTrust-DBC)

(1) **for** all machines P_i **do**
(2) **for** all requests r_j in meta-request R_v **do**
(3) OTL = search the lowest TL among all activities involved in performing r_j on Pi
(4) TC = ETS[TL(r_j), OTL]
(5) ESC[P_i, r_j] = S(TC)
(6) **do until** (all requests in R_v are scheduled OR can't be scheduled by his deadline)
(7) ECT(P_i) = α_i
(8) **for** all r_j in meta-request R_v **do**
(9) **for** each task in r_j **do**
(10) **for** all machine P_i **do**
(11) ECT(P_i) = ECT(P_i) + EET(P_i, f(r_j)) + ESC(P_i, r_j)
(12) sort the machine P_i by his completion time
(13) **do until** (find a machine to allocate the task or can't be scheduled)
assume assign this task to the first machine and calculate r_j cost ,complete time
if cost >budget reassign task to next machine and recalculate r_j cost, complete time
enddo
(14) calculate r_j completion time = max(all task completion time of r_j)
(15) Find the request r_j with the minimum time and really assign all task of r_j
(16) Update the vectorα_i
(17) **enddo**

3.2.2 Trust-Aware Cost Optimization Scheduling Algorithm (CTrust-DBC)

A description of the core of trust-aware cost optimization scheduling algorithm in algorithm 4. The difference from algorithm 3 is how to calculate the computation cost and task schedule under the constrains of deadline. The computation cost is calculated by statement (12)Cost(P_i) = (EET(P_i, f(r_j)) + ESC(P_i, r_j)) × Price(P_i).

Algorithm 4. trust-aware cost optimization scheduling algorithm within deadline constraints(CTrust-DBC)

(11) $ECT(P_i) = ECT(P_i) + EET(P_i, f(r_j)) + ESC(P_i, r_j)$

(12) $Cost(P_i) = (EET(P_i, f(r_j)) + ESC(P_i, r_j)) \times Price(P_i)$

(13) sort the machine P_i by his cost that execute $f(r_j)$

(14) **do until (** find a machine to allocate the task or can't be scheduled)

assume assign this task to the first machine and calculate r_j cost ,complete time

if time >deadline reassign task to next machine and recalculate r_j cost ,complete time

enddo

(15) calculate r_j cost = totalcost / total time

(16) Find the request r_j with the minimum cost and really assign all tasks of r_j

4 Performance Evaluation

4.1 Analysis of the Trust-Aware Schemes

This paper propose two algorithm, the goal of TTrust-DBC is to minimize the makespan and CTrust-DBC is to minimize the total cost. We only analysis the performance of TTrust-DBC, where makespan is defined as the maximum among the available times of all machines after they complete the tasks assigned to them. Initiallyα_m=0. The scheduler assigns request r_j to machine p_i such that the scheduling criterion makespan is minimized. Let X_{kij} be the mapping function computed by the scheduler, where $X_{kij} = 1$ if $task_j$ of request r_i is assigned to machine p_k and 0, otherwise. The makespan: $\Lambda = \max_{P_k} \{\alpha_k\}$ [9], where α_k is the available time of machine p_k after complete all tasks assigned to it by the scheduler. According to the definition of makespan, the value of α_k is given by:

$$\alpha_k = \sum_{i=0}^{N} \sum_{j=0}^{M} \left[EET(P_k, r_{ij}) + ESC(P_k, r_{ij}) \right] \times X_{kij} \tag{5}$$

A given heuristic scheduling computes a value of X_{kij} such that the makespan is minimized. It should be noted that due to the non-optimality of the heuristics, the makespan value may not be the globally minimal one.

Theorem: The makespan obtained a TTrust-DBC scheduling is always less than or equal to the makespan obtained by the T-DBC scheduling that uses the same assignment heuristic.

Proof: Let the makespan obtained by the TTrust-DBC be

$$\Lambda_T^{N,M} = \max\{ \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left[EET(P_k, r_{ij}) + ESC(P_k, r_{ij}) \right] \} \times X_{kij}^T \tag{6}$$

Let the makespan obtained by the T-DBC be:

$$\Lambda_{UT}^{N,M} = \max\{\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\left[EET(P_k, r_{ij}) + ESC(P_k, r_{ij})\right]\} \times X_{kij}^{UT} \quad (7)$$

1) For N=1, i.e., for the first task

$$\Lambda_{T}^{1,M} = \max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\} \times X_{k1j}^{T}$$

$$\Lambda_{UT}^{1,M} = \max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\} \times X_{k1j}^{UT}$$

Suppose, $\Lambda_{T}^{1,M} > \Lambda_{UT}^{1,M}$ and thus we will have the following inequality:

$$\max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\} \times X_{k1j}^{T} > \max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\} \times X_{k1j}^{UT}$$

X_{k1j}^{T} was chosen to minimize $\max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\}$ and was computed by $(EET(P_k, r_{1j}) + ESC(P_k, r_{1j})) \cdot X_{k1j}^{UT}$ was chosen to minimize $\max\{\sum_{j=0}^{M-1}\left[EET(P_k, r_{1j}) + ESC(P_k, r_{1j})\right]\}$ and computed by $EET(P_k, r_{1j})$. The above inequality, implies another choice that further minimizes the sum exists that was not selected by the heuristic scheduling algorithm. This is a contradiction. Hence, $\Lambda_{T}^{1,M} \leq \Lambda_{UT}^{1,M}$.

2) Let $\Lambda_{T}^{i,M} \leq \Lambda_{UT}^{i,M}$ (i.e., assume the Ttrust-DBC scheme provides a smaller makespan after mapping i requests). Following the above process, we can show that $\Lambda_{T}^{i+1,M} \leq \Lambda_{UT}^{i+1,M}$

Therefore, by induction $\Lambda_{T}^{N,M} \leq \Lambda_{UT}^{N,M}$ □

4.2 Simulation Results and Discussions

Simulations were performed to investigate the performance of the trust-aware DBC scheduling algorithm. The resource allocation process was simulated using a discrete event simulator with request arrivals modeled using a Poisson random process. Experiment simulates 1000 Grid users, 500 of them have 10 tasks per user and the deadline is 1450s, cost budget is 300 units. The other 500 of them have 20 tasks per user and the deadline is 1500s,cost budget is 500 units. The size of each task is randomly generated from 1 to 40. And also simulates 5 computation resources, each has different performance and process cost. The RTL values were randomly generated from {1-6}

representing trust levels A to F, respectively. Whereas, the OTL values were randomly generated from {1-5} representing trust levels A to E, respectively.

Experiment simulates the execution of 100,200,…1000 users for each scheduling, respectively, and repeat 100 times. Figure 2 shows the result of trust-aware time optimization DBC algorithm (TTrust-DBC) compare with time optimization algorithm (T-DBC) without considering trust. When the load of system is low, the makespan of TTrust-DBC is similar as T-DBC. But as the system load is high, the optimization algorithm is TTrust-DBC and average improvement is 26.46%. Figure 3 show the benefit of integrating the trust notion into a cost optimization DBC algorithm (CTrust-DBC). As same as Figure 2, when the system with high load , the total cost of CTrust-DBC was reduced almost 20.64%.

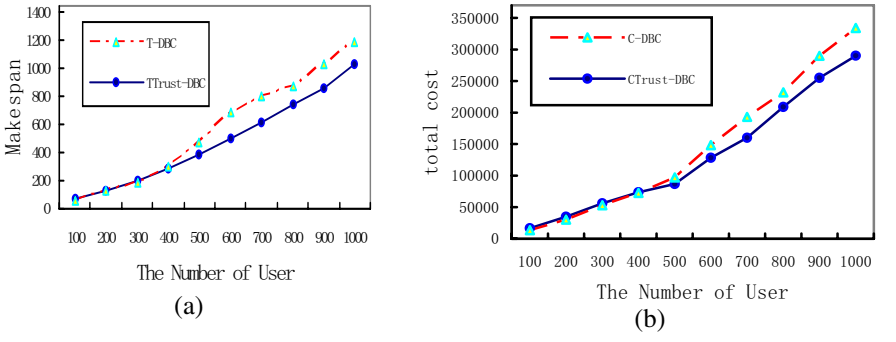


Fig. 1. (a) time optimization DBC scheduling algorithm simulation results (b) time optimization DBC scheduling algorithm simulation results

5 Conclusions and Future Research

Resource management plays a key role in Grid computing system, it is an efficient method to guarantee the security of the Grid. This paper examines the integration of the notion of "trust" into resource management such that the allocation process is aware of the security implications. First, we introduced a trust function based on dynamic reflect trust changing principle. Second, a behavior-trust model is proposed based on Grid resource management and scheduling. Third, we propose the trust-aware Grid economic model scheduling algorithm. At last, the theory analysis and simulations performed to evaluate the effectiveness of trust-aware scheduling algorithm indicate that the performance can be improved.

Since accurately computing trust value is a very difficult and complex problem. Several further issues remain to be addressed before the trust notion can be included in practical Grid. Some of these include techniques for managing and evolving trust in a large-scale distributed system, and mechanisms for determining trust values from ongoing transactions.

References

1. Butt. Ali Raza, Adabala, Sumalatha, Kapadia, H. Nirav, Figueiredo, J. Renato, Fortes, José. Grid-computing portals and security issues. Journal of Parallel and Distributed Computing, 2003, 63(10):1006~1014
2. F. Azzedin, Maheswaran. Integrating Trust into Grid Resource Management Systems. Proceedings of Parallel Processing 2002, International Conference on18-21 Aug. 2002 Page(s):47 ~ 54
3. F. Azzedin, M. Maheswaran. Towards trust-aware resource management in Grid computing systems. Cluster Computing and the Grid 2nd IEEE/ACM International Symposium CCGRID2002 21-24 May 2002 Page(s):419 ~ 424
4. S. Jorgensen, S. Taboubi, G. Zaccour. Retail promotions with negative brand image effects: Is cooperation possible?. European Journal of Operational Research, 2003,150(2):395~405
5. Gui Xiaolin, Xie Bing, Li Yinan, Qian Depei. Study on the behavior-based trust model in grid security system. Proceedings of Services Computing(SCC 2004), 2004 IEEE International Conference on 15-18 Sept. 2004 Page(s):506 ~ 509
6. R. Buyya, D. Abramson, J Giddy. High Nimrod/G: architecture for a resource management and scheduling system in a global computational grid. Proceedings of Performance Computing in the Asia-Pacific Region, 2000,283~289
7. R. Buyya, D. Abramson, S. Venugopal. The grid economy. Proceedings of the IEEE Volume 93, Issue 3, Mar 2005 Page(s):698~714
8. Chunlin Li, layuan Li. A distributed utility-based two level market solution for optimal resource scheduling in computational grid. Parallel Computing, 2005, 31(3-4):332~351
9. Keqin Li. Job scheduling and processor allocation for grid computing on metacomputers. Journal of Parallel and Distributed Computing, 2005,65(11):1406~1418
10. Chunlin,Li, Layuan,Li. The use of economic agents under price driven mechanism in grid resource management. Journal of Systems Architecture, 2004 50(9):521~535
11. He X. Sun, and von Laszewski. Qos guided min-min heuristic for grid task scheduling. Journal of Computer Science and Technology, 2003, 18, 442~451

QoS-Driven Web Services Selection in Autonomic Grid Environments

Danilo Ardagna[1], Gabriele Giunta[2], Nunzio Ingraffia[2], Raffaela Mirandola[1], and Barbara Pernici[1]

[1] Dipartimento Elettronica e Informazione, Politecnico di Milano, Italy
[2] Engineering Ingegneria Informatica S.p.A, R&D Lab, Italy

Abstract. In the Service Oriented Architecture (SOA) complex applications can be described as business processes from independently developed services that can be selected at run time on the basis of the provided Quality of Service (QoS). However, QoS requirements are difficult to satisfy especially for the high variability of Internet application workloads. Autonomic grid architectures, which provide basic mechanisms to dynamically re-configure service center infrastructures, can be be exploited to fullfill varying QoS requirements. We tackle the problem of selection of Web services that assure the optimum mapping between each abstract Web service of a business process and a Web service which implements the abstract description, such that the overall quality of service perceived by the user is maximized. The proposed solution guarantees the fulfillment of global constraints, considers variable quality of service profile of component Web services and the long term process execution. The soundness of the proposed solution is shown trough the results obtained on an industrial application example. Furthermore, preliminary computational experiments show that the identified solution has a gap of few percentage units to the global optimum of the problem.

1 Introduction

The recent trend towards shorter business cycles encourages the use of the emergent grid technologies and of the flexible service oriented development paradigm to meet the goal of construction and management of reliable and adaptable e-business applications at low cost. Our work aims to meet the requirements and the challenges posed by e-business applications by using the new emerging technological trends. Specifically, we intend to pursue the QoS-driven selection and composition of Web services for e-business applications in autonomic grid environments.

SOA paradigm foresees the creation of complex applications, described as business processes, from independently developed services that can be selected at run time. Usually, a set of functionally equivalent services exist, that is, services which implement the same functionality, but differ for non-functional characteristics, i.e., QoS properties. In this context, our goal is to discover the optimum mapping between each abstract Web service of a business process and a Web service which implements the abstract description.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1273–1289, 2006.
© Springer-Verlag Berlin Heidelberg 2006

However, QoS requirements are difficult to satisfy especially due to the high variability of Internet application workloads. Internet workloads can vary by orders of magnitude within the same business day [12]. Such variations cannot be accommodated with traditional allocation practices, but require autonomic computing self-managing techniques [15], which dynamically allocate resources among different services on the basis of short-term demand estimates.

This dynamic fulfillment of varying QoS requirements can be enhanced by grid computing. Grid middleware provides basic mechanisms to manage the overall infrastructure of a service center, implementing service differentiation and performance isolation for multiple Web services sharing the same physical resources and simplifying the re-configuration of the physical infrastructure.

In the literature, resource allocation in scientific grid workflows has been analyzed in depth [23,28]. However scientific workflows and composed e-business processes have different computing requirements, which pose diverse constraints on the re-configuration of the infrastructure. Indeed, in scientific grid, each task is computation intensive and is executed by a single computing resource. Vice versa, in e-business applications, the execution of a single Web service operation requires few CPU seconds and the high computation requirements are due to the high incoming workload. Furthermore, grid workflow and composed Web services have different characteristics: scientific grids introduce many tasks, generally in the order of thousands (the higher is the number the higher is the parallelism level of the workflow). Vice versa, e-business composed processes introduce a lower number of tasks since they correspond to high level application operations [9,25]; also the number of candidate resources for each task is very high (since correspond to available computing and storage resources) for scientific grids and moderate for e-business grids where resources are Web services candidate for the execution of high level operations. For this reasons resource allocation in scientific and e-business grids requires different approaches.

In this paper we present a reference framework to support the execution of Web services based e-business applications in autonomic grid environments. The problem of selection of Web services in composed services such that the QoS for the end user is maximized is formulated as a Mixed Integer Linear Programming (MILP) model. The formulation guarantees the fulfillment of global constraints and extends the work in [5] allowing the execution of stateful Web services. The devised solution is then applied to a case study derived from an industrial settings and the results obtained so far show the effectiveness of the proposed approach.

The paper is organized as follows. Section 2 briefly reviews related works. Section 3 describes the adopted case study. In Section 4 we describe the composed Web service specification and the quality model adopted. Section 5 is devoted to the description of the problem of resource allocation in grid environments, while Section 6 presents the numerical results obtained in the considered example application. Finally, conclusions are drawn in Section 7.

2 Related Work

Grid computing is proposed as an infrastructure providing transparent resource sharing between collaborating organizations [14]. Resource allocation and scheduling represents a key issue to deal with in grid environment. Several papers have been presented in the literature on this topic with a special emphasis on scientific computing applications. Resource management spans task-based algorithms that greedly allocate tasks to resources and workflow-based algorithms that search for an efficient allocation for the entire workflow [10]. Another aspect/dimension that distinguishes the various approaches concerns the type of resource manager considered: centralized or distributed [23] . A comprehensive survey of different workflow management systems and grid ongoing research projects is presented in [6].

Recently, QoS issues in Web services selection and composition have obtained great interest in the Web services research community. Different approaches have been followed so far, spanning the use of QoS ontologies [17], the definition of ad-hoc methods in QoS-aware framework [21,24], and the application of optimization algorithms [25,5,9].

As already discussed in the introduction, the adoption of grid infrastructure for the execution of e-business composed processes introduces different computing requirements and goals. These different requirements lead to diverse optimization objectives: in scientific grid several heuristics, such as genetic algorithms, GRASP, simulated annealing and tabu search, have been proposed to identify the optimum mapping between tasks and computing resources in order to minimize the workflow total execution time. Vice versa, in the Web service composition literature MILP models, and genetic algorithms [8] are applied. Furthermore, in this case, the optimization problem is multi-objective since several QoS parameters are considered.

In this paper the problem of selection of Web services in composed services will be formulated as a MILP problem. We extend the work in [5] by considering varying Web services QoS profiles, long term composed services execution, and by allowing the execution of stateful Web services. We extend our previous work [4] by providing computational experiment results to show the effectiveness of the proposed approach.

3 A Case Study: The Cost Allocation Process

The reference scenario is a Virtual Organization (VO) composed by a set of small and medium enterprises which share their computing resources for the execution of an ERP software. The software adopted is *AgileERP3*[1], an open-source ERP platform which introduces ISO compliant workflow processes and can use basic functionalities as lightweight Web services access. In this scenario, we restrict our attention to the area of the *cost accounting* functions whose goal is the computation of product costs. The followed costs accounting procedure is

[1] http://www.agilerp.it

based on Cost Center(CC), i.e., on the computation of the product costs and on their attribution to the center(s) incurring these costs.

Specifically, we focus on one of these functions, namely, the Indirect Costs Allocation Process (ICAP). This process assigns indirect costs to individual CCs. Indirect costs are costs that are associated with or caused jointly by two or more CCs, but that can not be directly assigned to each one of them individually. The main purpose of ICAP is to share the indirect costs charged to a single CC, called *Transfer Cost Center* (TCC), with different CC(s), called *Receiver Cost Center* (RCC).

The relationship among different CCs is represented by an *allocation cost hierarchy*, where each CC is defined as transfer or receiver, as it can transfer or receive costs from other CCs. The relationships defined by the allocation hierarchy can change during the time for economical reasons.

A TCC uses the *allocation rate* to calculate the percentage of indirect costs given to RCCs. Each TCC can allocate all or part of its own costs. In order to evaluate the percentage of costs charged to each RCC, a TCC can use one or more *cost drivers or allocation bases* to charge its own costs. Figure 1 shows an example of allocation hierarchy composed by a set of CCs ($C_{n.1}$, $C_{n.2}$, and $C_{n.3}$) that, at the same time, receive costs from the root (C_n) and transfer costs to the leaves ($C_{n.2.1}$, $C_{n.3.1}$).

As an example, let us consider the cost allocation of an electrical bill assigned to C_n with total value €1,500. The ICAP establishes that: (a) the electrical bill can be allocated from C_n (TCC) to $C_{n.1}$, $C_{n.2}$, $C_{n.3}$ (RCCs) according to the allocation hierarchy shown in Figure 1, (b) the costs percentage that TCC allocates to RCCs is determined by using the allocation rate as follows. If we assume that the allocation rate is equal to 1, then all the amount of the bill is transferred to the RCCs. For example, if $C_{n.1}$ cost driver is 0.1, $C_{n.2}$ cost driver is 0.2 and $C_{n.3}$ cost driver is 0.7, the cost allocated to each RCC is given by the amount received from TCC times each RCC cost drivers, i.e., $cost_{n.1}$ = €150, $cost_{n.2}$ = €300, and $cost_{n.3}$ = €1050. Since other RCCs exist, ($C_{n.2.1}$ and $C_{n.3.1}$), the ICAP repeats the steps described above by considering $C_{n.1}$, $C_{n.2}$, $C_{n.3}$ as TCCs and $C_{n.2.1}$, $C_{n.3.1}$ as RCCs.

ICAP steps are modeled by the UML activity diagram shown in Figure 1. The *BalanceVerification* task checks the balance for every CC in input. In case of error, the *NotificationError* task sends an error message to the end user. Otherwise, the ICAP starts and evaluates the cost allocation for every CC. The *Calculate Charge Balance* task, determines the amount of cost to be charged to a CC by the following equation:

$$\text{Charge Balance} = (\text{Accounting Balance} + \text{Total Cost Received}) \cdot (1 - \text{Allocation Rate})$$

As shown by the formula above, if the allocation rate is zero, the TCC cannot share costs with the RCCs, as all the costs are charged to it. In this case, the next CC of the allocation hierarachy is determined by a hierarchical navigation algorithm. On the contrary, the *CalculateTotalCostTransferred* task evaluates the total cost by the following equation:

$$\text{Accounting Balance} + \text{Total Cost Received} =$$
$$= \text{Charge Balance} + \text{Total Cost Transferred}$$

The *Allocate Costs* task evaluates the rate of total cost transferred to each RCC according to the *cost drivers*. A cost allocation ends when the navigation of the hierarchy terminates.

A cost center allocation ends when the navigation of the hierarchy terminates. The ICAP stops when all cost centers have been considered. The composed process will generate also a PDF and an Excel report file and the *Consolidate Costs* task will consolidate the balance.

As already stated, each of these functionalities is realized as a Web service and can be executed in each VO. The adoption of grid computing in this scenario is very promising since the re-configuration overhead is limited (the ERP software is available in every local grid). Furthermore, VOs can access computing resources of cooperating sites in order to manage peaks of loads obtained when the ICAP process is performed on a large number of CCs.

4 Business Process Specification and Quality Model

In our framework, a composite service is specified as a high-level business process in BPEL language in which the composed Web service is specified at an abstract level. In the following we refer to an *abstract Web service* as a *task* t_i, while Web services selected to be executed are called concrete Web services. Concrete Web services are registered with associated keywords and their WSDL specification in a service registry [7]. We assume that the registry stores also for each operation o of a concrete Web service j, the values of the n-th quality dimension $q^n_{j,o}(t)$ and the number of instances that can be executed during a specific time interval.

Some annotations are added to the BPEL specification in order to identify:

- global and local constraints on quality dimensions;
- the maximum number of iterations for cycles;
- the expected frequency of execution of conditional branches;
- user preferences, a set of normalized weights $\{\omega_1, \omega_2, \ldots, \omega_N\}$, $\sum_{n=1}^{N} \omega_n = 1$, indicating the user preferences with respect to n-th quality dimension;
- Web service dependency constraints.

Global constraints specify requirements at process level, while local constraints define quality of Web services to be invoked for a given task in the process. The optimization problem will consider statistically all of the possible execution scenarios of the composite service according to their probability of execution. The maximum number of iterations and frequency of execution of conditional branches can be evaluated from past executions by inspecting system logs or can be specified by the composite service designer. If an upper bound for cycles execution cannot be determined, then the optimization could not guarantee that global constraints are satisfied [25]. At compilation time, cycles are unfolded according to the maximum number of iterations. Finally, Web service dependency

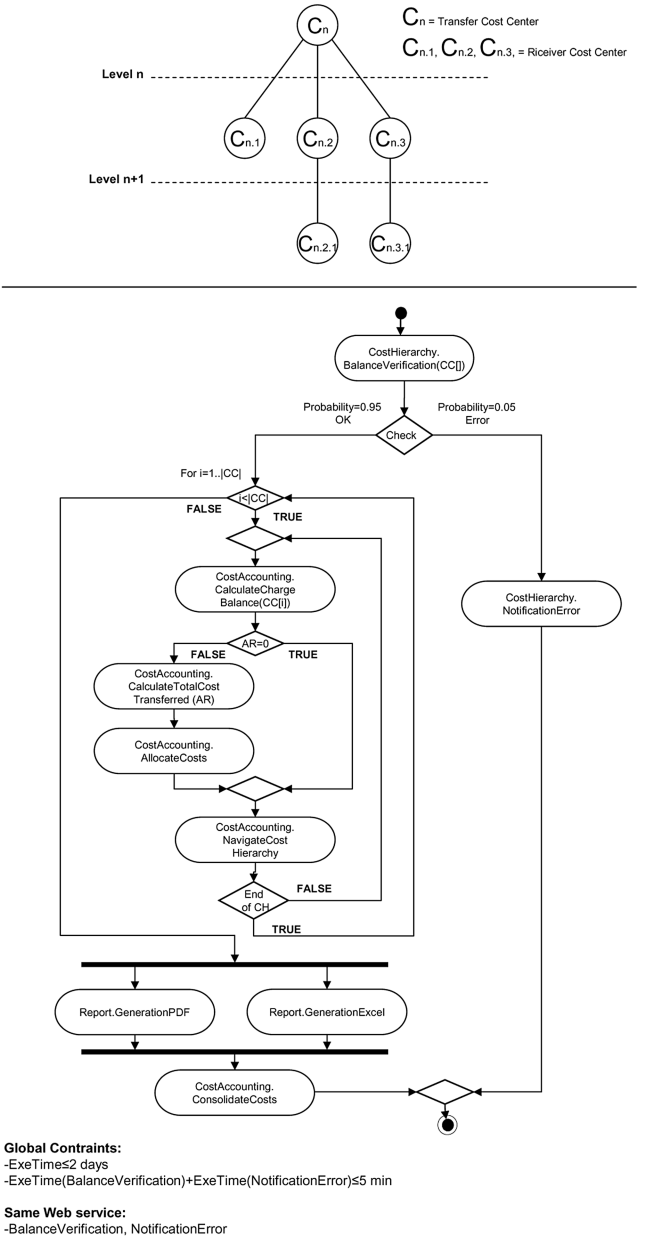


Fig. 1. The Indirect Costs Allocation Process

constraints impose that a given set of tasks in the process is executed by the same Web service. This type of constraints allows considering both stateless and stateful Web services in composed services execution. Constraints and BPEL annotations are specified by WS-Policy (see [3]).

In the reference example reported in Figure 1 the end user poses as global constraints that the ICAP process is executed within two days and, in case of error, the notification message is sent within 5 minutes. Furthermore, cycles number of iteration can be determined deterministically at invocation time. A dependency constraint imposes that the balance verification and notification error are performed by the same concrete Web service (for performance issues the notification error task can exploit the results of the balance verification).

QoS profiles follow a discrete stepwise function, that is periodic with period T (see Figure 2). As it will be discussed in Section 5.1, QoS profiles are obtained as result of local grids resource allocation. In the following the continuous time will be denoted by t, while the discrete time interval will be denoted by u. The discretization interval will be denoted by $\triangle$ and we assume that the QoS profile is constant in every interval of length $\triangle$. In autonomic systems $\triangle$ is about half an hour [2], while, if we assume that the incoming workload has a daily seasonal component, T is 24 hours.

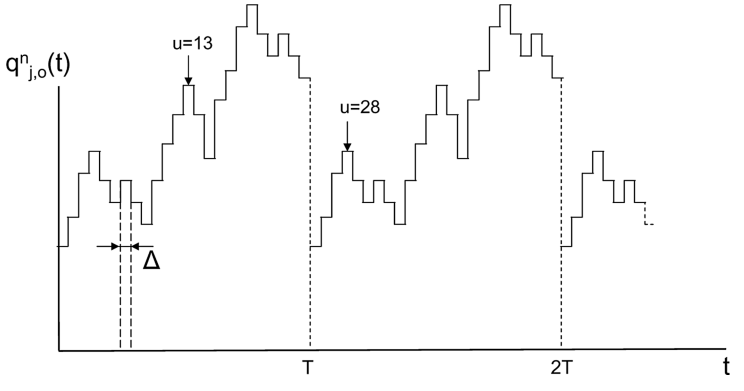


Fig. 2. Example of Periodic QoS Profile

In Figure 2, the time interval $u = 13$ and $u = 28$ are highligted, where $T = 24 \cdot \triangle$ and $\triangle = 1$ hour. The index u can range in $[1, U]$, where $U = \lceil E/\triangle \rceil$ and E indicates the execution time global constraint for the composed process. For the periodicity we have:

$$q^n_{j,o}(t) = q^n_{j,o}(t + uT) \quad u \in [1, U] \tag{1}$$

and

$$q^n_{j,o,u} = q^n_{j,o,u \bmod (T/\triangle)} \quad u \in [1, U] \tag{2}$$

The problem of maximization of QoS is multi-objective since several quality criteria can be associated with each operation o of a Web service j in the time interval u. We focus on Web services *execution time*, *availability*, *price*, and *reputation*. This set of quality dimensions have been the basis for QoS consideration

also in other approaches both in the Web service and grid community [25,20,19]. Quality dimensions can be also classified as positive and negative criteria. A quality attribute is positive (negative) if a higher value corresponds to a higher (lower) quality.

In the following N will indicate the number of quality dimensions of interest for the end user, the value of the quality dimensions of task t_i and of each operation invocation will be denoted by q_i^n and $q_{j,o,u}^n$, respectively. Finally, the execution time will be indexed by $n = 1$.

Note that, if the same service is accessible from the same provider, but with different quality characteristics (e.g. quality level), then multiple copies of the same service will be stored in the registry, each copy being characterized by its quality profile.

Finally we denote with $\mathcal{N}_{j,o,u}$ the number of instances of operation o of Web service j which can be executed in the time interval u; we assume that the Web services execution is supported by limited resources.

5 The Reference Grid Environment

In our framework, VOs' resources are represented by concrete Web services which are physically deployed and executed by multiple *Local Grids*. A VO can use concrete Web services that are located in different VO sites, to execute a particular abstract composed service. Local Grid includes also: a *Service Registry*, a *Local Resource Allocation* module and a *Broker* (see Figure 3). The registry stores the WSDL specification, the QoS profile and the number of instances which can be executed in the time interval u for every operation. The broker receives composed Web service execution requests from VO members and external users, consults the local and remote registries and determines the execution plan (i.e. abstract to concrete Web service assignment) for the composed service execution. Local grid resources are reserved according to the execution plan identified by the broker. VOs and the end user establish Service Level Agreement (SLA) contracts for the service provisioning and $\mathcal{N}_{j,o,u}$ is updated accordingly.

Resource management introduces two different optimization problems which corresponds to the VOs (providers) and users perspectives: (i) each VO would like to maximize the SLA revenues and the use of physical resources, (ii) the end user is interested in the maximization of the QoS of the composed service execution.

SLA revenues maximization is performed locally by the local resource allocation modules, while QoS maximization is evaluated by brokers. Note that brokers of different local grids can collaborate to identify the optimum abstract to concrete Web service mapping and a composed service can be executed by concrete Web services located in different local grids.

This paper focuses on the maximization of the QoS for the end user which will be discussed in depth in Section 5.2. Local grid resource management has been presented in previous works [26,2] and will be briefly summarized in Section 5.1.

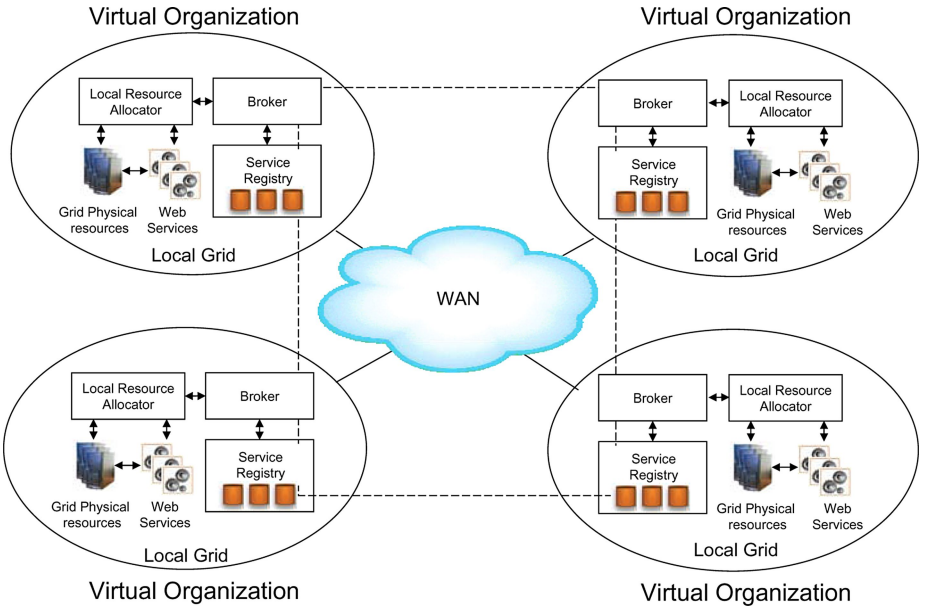


Fig. 3. Grid Reference Framework

5.1 Local Resource Allocation

Each VO needs to allocate local grid physical resources to different Web service operation invocations in order to maximize the revenues from SLA, while minimizing resource management costs. One of the main issues is the high variability of the incoming request workload which can vary by orders of magnitude within the same business day [12]. It is difficult to estimate workload requirements in advance, and planning the capacity for the worst-case scenario is either infeasible or extremely inefficient. In order to handle workload variations, many service centers have started employing autonomic techniques [16], that allow the dynamic allocation of physical resources among different Web services invocations on the basis of short-term demand estimates. The goal is to meet the application requirements while adapting the physical infrastructure. The adoption of grid computing is very promising in this application area, since basic mechanisms to provide resource virtualization, service differentiation, performance isolation, and dynamic re-configuration of the physical infrastructure are implemented by the grid middleware layer [14,1].

The local resource allocation is performed periodically with period $\triangle'$ (e.g., 10-30 minutes [27,2]) on the basis of a short-term workload prediction (see Figure 4a). Note that $\triangle'$ is lower than $\triangle$, the discretization time interval adopted to model the QoS in the service registry. The short term predictor forecasts the number of Web services invocations for the next control interval denoted as $\hat{\mathcal{N}}_{j,o}(t)$. The local resource allocator uses also some low level information provided by the grid monitoring infrastructure in order to identify requests of

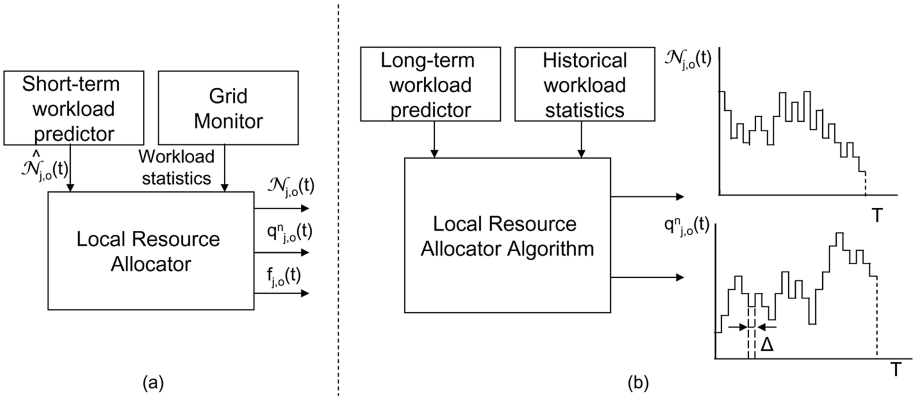


Fig. 4. Local Resource Allocation

different Web services operations and to estimate requests service times (i.e., the CPU and disk time required by the physical infrastructure to execute each operation).

In order to maximize revenues from SLA, the local resource allocator determines the fraction of capacity assigned to each Web service invocation $f_{j,o}(t)$, relying on the virtualization mechanism and performance isolation provided by the grid infrastructure. The local resource allocator employs also an admission control scheme [22] that may reject requests in order to guarantee that the QoS requirements are met, or to avoid service instability caused by capacity restrictions. Overall, the local resource allocator determines the number of Web service operation invocations $\mathcal{N}_{j,o}(t) \leq \hat{\mathcal{N}}_{j,o}(t)$ allowed for the next control interval, its corresponding QoS profile $q^n_{j,o}(t)$, and the fraction of capacity of the physical grid infrastructure $f_{j,o}(t)$ devoted to its execution.

As we discussed in [2], the local resource allocator algorithm can be used with a long-term workload predictor and historical workload statistics from systems log (see Figure 4b) in a simulation environment in order to determine $\mathcal{N}_{j,o}(t)$ and the quality profile on the long term. If we assume that the incoming workload has a daily periodic component, as frequently happens in practice [18], $\mathcal{N}_{j,o}(t)$ and $q^n_{j,o}(t)$ are also periodic and can be described with granularity $\triangle > \triangle'$ in a service registry.

Note that, resource allocation is performed at local grid level since a global resource allocation scheme which determines $\mathcal{N}_{j,o}(t)$ and $q^n_{j,o}(t)$ for the whole grid infrastructure introduces a high overhead [13]. Furthermore, by implementing a local resource allocation policy VOs have a greater control of their own physical infrastructure.

5.2 QoS Maximization for the End-User

Requests of execution of composed Web service from VOs or external users are submitted to grid brokers specifying the preferences (weights) and the set of

local and global constraints. A broker solves the Web Service Selection (WSC) problem which consists in finding the optimal mapping between process tasks and Web service operations.

In the following, Web services will be indexed by j while operations will be indexed by o. We will indicate with WS_i the set of indexes of Web services ws_j candidate for the execution of task t_i; with OP_j the set of indexes of operations implemented by Web service ws_j, and with $ws_{j,o}$ the invocation of operation $o \in OP_j$ of Web service ws_j. Let be I the number of tasks of the composed service specification and J the number of candidate Web services. We assume that cycles are unfolded according to the maximum number of iterations. For the sake of simplicity in the following definitions we assume that a composite service is characterized by a single initial task t_1 and a single end task t_I:

- *Execution Path.* A set of tasks $\{t_1, \ldots, t_i, \ldots, t_I\}$ such that t_1 is the initial task, t_I is the final task and no t_{i_a}, t_{i_b} belong to alternative branches. Execution paths will be indexed by k and denoted by ep_k. We will indicate with $\mathcal{A}_k$ the set of indexes of tasks included in the execution path and with K the number of different execution paths arising from the composed service specification. Note that an execution path can include parallel sequences (see Figure 5, where the ICAP process cycles are unfolded by assuming that only one CC is considered and the hierarchy is navigated in one step). The evaluation of the n-th quality dimension along the k-th execution path will be denoted as $q^n(k)$.
- *Global Plan.* The global plan is a set of ordered triples $\{(t_i, ws_{i,o}, x_i)\}$, which associates every task t_i to a given Web service operation invocation $ws_{j,o}$ at time instant x_i and satisfies local and global constraints for all execution paths.

Note that, the set of execution paths of an activity diagram identifies all the possible execution scenarios of the composite service. The optimization problem will consider all of the possible execution scenarios according to their probability of execution, which can be evaluated by the product of the frequency of execution of branch conditions included in execution paths and annotated in the BPEL specification. Under these definitions, a local constraint can predicate only on properties of a single task. Vice versa, global constraints can predicate on quality attributes of an execution path or on a subset of tasks of the activity diagram, for example a set of subsequent tasks. The WSC problem is formulated as a mixed integer linear programming problem. The decision variables of our model are the followings:

- $y_{i,j,o,u} \in \{0, 1\}$ is equal to 1 if the task t_i is executed by Web service $j \in WS_i$ with the operation $o \in OP_j$ during the time interval u, 0 otherwise;
- $w_{i,u} \in \{0, 1\}$ is equal to 1 if the task t_i is executed during time interval u, 0 otherwise;
- $x_i \in \mathbb{R}^+$ indicates the time instant in which task t_i is executed.

The goal of the WSC problem is to maximize the average aggregated value of QoS. The average is obtained by considering all of the possible execution

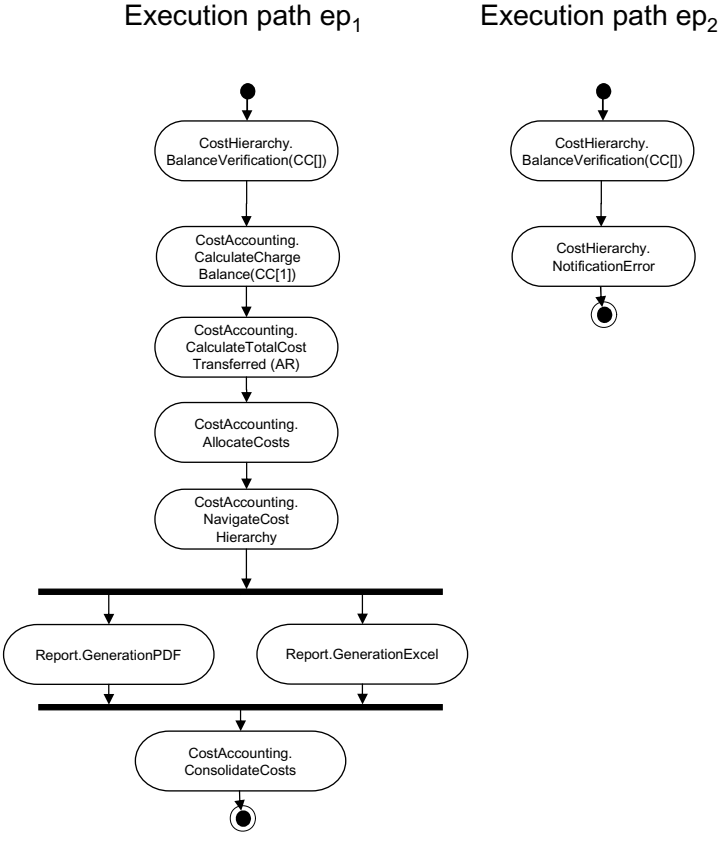


Fig. 5. Execution Paths

scenarios, i.e., all of the execution paths arising from the composed service specification, and their probability of execution $freq_k$. As we have discussed in [5] the aggregated value of QoS can be obtained by applying the Simple Additive Weighting (SAW) technique, one of the most widely used techniques to obtain a *score* from a list of dimensions. The WSC problem can be formulated as:

P1) $$\max \sum_{i=1}^{N} \sum_{j \in WS_i} \sum_{o \in OP_j} \sum_{u=1}^{U} \gamma_{i,j,o,u} \cdot y_{i,j,o,u}$$

$$\sum_{j \in WS_i} \sum_{o \in OP_j} \sum_{u=1}^{U} y_{i,j,o,u} = 1 \qquad i = 1, \ldots, I \tag{3}$$

$$w_{i,u}(u-1)\triangle \leq x_i \qquad i = 1, \ldots, I; u = 1, \ldots, U \tag{4}$$

$$u\triangle w_{i,u} + E(1 - w_{i,u}) \geq x_i \qquad i = 1, \ldots, I; u = 1, \ldots, U \tag{5}$$

$$w_{i,u} = \sum_{j \in WS_i} \sum_{o \in OP_j} y_{i,j,o,u} \qquad i = 1, \ldots, I; u = 1, \ldots, U \tag{6}$$

$$y_{i,j,o,u} \leq \mathcal{N}_{j,o,u} \qquad i = 1, \ldots, I; \forall j \in WS_i \quad \forall o \in OP_j;\ u = 1, \ldots, U \tag{7}$$

$$q_i^n = \sum_{j \in WS_i} \sum_{o \in OP_j} \sum_{u=1}^{U} q^n_{j,o,u \bmod (T/\Delta)} \cdot y_{i,j,o,u} \quad i = 1, \ldots, I; n = 1, \ldots, N \tag{8}$$

$$x_{i_b} - (q^1_{i_a} + x_{i_a}) \geq 0 \qquad \forall t_{i_a} \rightarrow t_{i_b} \tag{9}$$

$$\tag{10}$$

$$q^n(k) = \sum_{i \in \mathcal{A}_k} \tilde{\gamma}_i^n(k) \cdot q_i^n \qquad k = 1, \ldots, K; n = 1, \ldots, N \tag{11}$$

$$x_I + q_I^1 \leq E \tag{12}$$

$$q^n(k) \ [\geq | \leq] \ Q^n \qquad k = 1, \ldots, K; n = 2, \ldots, N$$

$$x_i \in \mathbb{R}^+ \qquad i = 1, \ldots, I;$$

$$q_i^n \in \Re^+ \qquad i = 1, \ldots, I; n = 1, \ldots N,$$

$$q^n(k) \in \Re^+ \qquad k = 1, \ldots, K; n = 1, \ldots N$$

$$y_{i,j,o,u}, w_{i,u} \in [0, 1] \qquad \forall i, j, o, u$$

The objective function is a linear combination of the decision variables $y_{i,j,o,u}$, where the coefficients $\gamma_{i,j,o,u}$ depend on execution paths frequency of execution ($freq_k$), end user preferences ($\{\omega_n\}$) and are obtained by the normalization process. Constraints family (3) garantees that each task t_i is assigned to only one concrete $ws_{j,o}$ and its execution can start in a specific time interval u. Constraints families (4) and (5) relate variable x_i and variable $w_{i,u}$. If $w_{i,u}$ is set to 1 then x_i value belongs to u interval, otherwise x_i can assume any value in the range [0,E]. For example if $w_{i,5} = 1$ then we can obtain from constraint (4) $x_i \geq 4\triangle$ and from constraint (5) $x_i \leq 5\triangle$; on the other hand if $w_{i,5} = 0$ we obtain $x_i \geq 0$ and $x_i \leq E$. Constraints family (6) relates variables $w_{i,u}$ and $y_{i,j,o,u}$, indeed if the task t_i is executed by invoking in interval u the operation o of Web service j, i.e. $y_{i,j,o,u} = 1$, then $w_{i,u}$ is raised to 1. Constraints family (7) guarantees that the number of parallel Web service invocation operations that can be executed in the same interval u must be lower or equal to the number of available invocation instances. Constraints family (8) expresses the quality of every task in term of the quality of the selected service. Note that for constraints (3) and (6) there is only one operation invocation in a specific interval and hence a task quality value is given by the quality value of the selected Web service operation. Constraint family (9) represents precedence constraints for subsequent tasks in the activity diagram. If a task t_{i_b} is a direct successor of task t_{i_a} (indicated as $t_{i_a} \rightarrow t_{i_b}$), then the execution of task t_{i_b} starts after task t_{i_a} termination. Constraints family (11) evaluates the quality value of the composed service along the k-th execution path. Coefficients $\tilde{\gamma}_i^n(k)$ depend on the composition rule of the quality parameter. For example the price of the composed service is given by the sum of the prices of the Web service operation invocations, i.e. $\tilde{\gamma}_i^n(k) = 1$. Vice versa (see [25]), the reputation is evaluated as the average of the reputation of the Web service operation invocations, i.e. $\tilde{\gamma}_i^n(k) = 1/|\mathcal{A}_k|$.

Constraint (12) guarantees that the execution time of the composed process is less or equal to the execution time global constraint E (from Section 4 we assume that the composed process has a single start and a single end task, hence the execution time of the composed process is given by the sum of the

starting time of the last task t_I and its corresponding execution time). Finally, constraints familiy (13) represents the global constraints to be fulfilled for the remainder quality dimensions, the inequality is $\geq$ ($\leq$) for positive (negative) quality parameters.

Problem P1) can include Web service dependency constraints which can be formulated as follows. If two task t_{i_a}, t_{i_b} must be executed by the same Web service, then the following constraint families are introduced:

$$\sum_{o \in OP_j} \sum_{u=1}^{U} y_{i_a,j,o,u} = \sum_{o \in OP_j} \sum_{u=1}^{U} y_{i_b,j,o,u} \quad \forall j \in WS_{i_a} \cap WS_{i_b};$$

$$\sum_{o \in OP_j} \sum_{u=1}^{U} y_{i_a,j,o,u} = 0, \forall j \in WS_{i_a} \setminus WS_{i_b};$$

$$\sum_{o \in OP_j} \sum_{u=1}^{\lceil E/\triangle \rceil} y_{i_b,j,o,u} = 0, \forall j \in WS_{i_b} \setminus WS_{i_a}.$$

Local constraints can predicate on properties of a single task and can be included in the model as follows. For example if the designer requires that the execution time of task t_{i_a} has to be less or equal than a given value $\overline{E}_a$, then the following constraint is introduced:

$$\sum_{j \in WS_{i_a}} \sum_{o \in OP_j} \sum_{u=1}^{U} q^1_{j,o,u} y_{i_a,j,o,u} \leq \overline{E}_a \tag{13}$$

Problem P1) has integer variables and linear constraints. In [5] we have shown that the problem of selection of Web services when the quality profile is constant with respect to the time variable, is equivalent to a Multiple choice Multiple dimension Knapsack Problem which is NP-hard, hence P1) is NP-hard.

6 Experimental Results

Experimental analyses have been conducted by implementing a prototype optimization tool based on *CPLEX*, a state of the art integer linear programming solver. The approach has been tested on a wide set of randomly generated instances considering the ICAP composed process described in Section 3. The number of iterations of the inner cycle has been varied between 0 and 16 with step 2. Hence, the number of tasks has been varied between 5 and 69 with step 8. The number of candidates Web services operations per task has been varied between 5 and 20 with step 5. The execution time global constraints has been varied between 24 and 72 with step 24 hours and $\triangle$ has been set equal to one hour.

Quality of services values of candidate Web services have been randomly generated according to the values reported in the literature (see [5]). Availability values were randomly generated assuming a uniform distribution in the interval 0.95 and 0.99999. Reputation was determined in the same way but considering the range [0.8, 0.99]. As in [11], we assume that the execution time have a

Gaussian distribution. We assumed that the price of each service invocation was proportional to service reputation and availability and inverselly proportional to the execution time (i.e., the higher is the execution time of a Web service operation invocation, the lower is the price). Finally, the set of weights ω_i was random generated and weights were adjusted to sum 1. Analyses have been performed on a 3 GHz Intel Pentium IV Workstation.

The optimization execution time varies with the size of the problem instance. Since the broker has to solve the WSC problem efficiently, CPLEX execution time is limited to one minute. CPLEX can find the global optimum of the WSC problem within one minute for small problem size instances. For other instances anyway the gap between the approximate solution obtained in one minute and the global optimum, which can be obtained by CPLEX sometimes in several hours, is few percentage points. Table 1 reports, as an example, the gap between the (approximate) solution obtained in one minute and the global optimum for 108 random generated problem instances of different size. In this case the maximum gap is 8.3% while on average the gap is 1.5%.

Table 1. Percentage gap between the approximate and global optimum solutions

	E=24h				**E=48h**				**E=72h**			
	Num. of candid. WS operations				**Num. of candid. WS operations**				**Num. of candid. WS operations**			
Num. of tasks	5	10	15	20	5	10	15	20	5	10	15	20
5	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0.7	0	2.1	1.2	1.8	0	3.2	2.1	2.9
29	0.7	1.5	1.8	0.8	3.9	1.6	2.9	1.2	1.5	0.8	1.6	2.3
37	2.9	2.7	1.5	2.0	1.3	2.1	5.1	3.2	1.4	3.3	2.9	1.7
45	2.1	2.3	1.6	1.7	2.1	3.2	2.6	2.8	1.1	3.1	2.2	2.3
53	0.6	0.7	0.5	1.9	2.1	1.6	1.9	1.6	2.0	2.3	1.6	1.8
61	3.3	0	8.3	1.9	0.9	1.2	1.7	1.2	1.8	2.1	1.4	0.9
69	3.1	1.4	2.4	2.3	2.7	1.9	2.1	0.9	1.4	1.7	1.3	2.9

7 Conclusions

This paper presents a framework for the development of e-business applications built on autonomic grid computing infrastructure where the service selection and composition is performed to guarantee that the overall quality perceived by the user is maximized. Results has shown that the MILP model formulation can be solved efficiently by a state of the art linear solver which can obtain in a limited time an approximate solution within few percentage point units of the global optimum. Our short/medium term goal includes the implementation of the proposed case study to validate the proposed approach on a real environment

setting. The long term goal is the realization of a framework where different kind of selection and composition methods are provided; in such a way the user can choose the methods best suited for his application and quality requirements.

Acknowledgements

The work reported in this paper has been partially supported by the DISCORSO FAR Italian Project. Thanks are expressed to Giuliana Carello and Marco Trubian for many fruitfull discussions on optimization issues. We are grateful to Engineering Ingegneria Informatica S.p.A. and especially to Antonio Filograna, Silvio Sorace and Giuseppe Vella for technical support.

References

1. The egee Project (Enabling Grid for E-Science). http://public.eu-egee.org/test/.
2. J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian. Resource management in the autonomic service-oriented architecture. In *ICAC 2006 Proc.*, 2006. In Press.
3. D. Ardagna, C. Cappiello, P. Plebani, and B. Pernici. A Framework for Describing and Supporting Adaptive Context-aware Web Services. Politecnico di Milano Technical Report 2006.48 http://www.elet.polimi.it/upload/ardagna/Tech2006-48.pdf, June 2006.
4. D. Ardagna, S. Lucchini, R. Mirandola, and B. Pernici. Web services composition in autonomic grid environments. In *Grid and P2P Worflow Workshop Proc.*, 2006. In press.
5. D. Ardagna and B. Pernici. Global and Local QoS Guarantee in Web Service Selection. In *BPM 2005 Workshops Proc.*, pages 32–46, 2005. Nancy.
6. D. Berlich, M. Kunze, and K. Schwarz. Grid computing in Europe: from research to deployment. In *CRPIT '44: Proc. of the 2005 Australian workshop on Grid computing and e-research*, pages 21–27, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
7. D. Bianchini, V. D. Antonellis, B. Pernici, and P. Plebani. Ontology-based methodology for e-Service discovery. *Information Systems*, 31:361–380, 2006.
8. P. A. Bonatti and P. Festa. On optimal service selection. In *WWW 2005 Proc.*, pages 530–538, 2005. Chiba.
9. G. Canfora, M. Penta, R. Esposito, and M. L. Villani. QoS-Aware Replanning of Composite Web Services. In *ICWS 2005 Proc.*, 2005.
10. J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGRID 2003 Proc.*, Jul. 2003.
11. S. Chandrasekaran, J. A. Miller, G. Silver, I. B. Arpinar, and A. P. Sheth. Performance Analysis and Simulation of Composite Web Services. *Electronic Market: The Intl. Journal of Electronic Commerce and Business Media*, 13(2):120–132, 2003.
12. J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *SOSP 2001 Proc.*, pages 103–116, 2001. Banff.
13. L. Chunlin and L. Layuan. A distributed utility-based two level market solution for optimal resource scheduling in computational grid. *Parallel Comput.*, 31(3+4):332–351, 2005.

14. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Intl. J. of Supercomputer Applications*, 2001.
15. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 1(31):41–50, 2003.
16. Z. Liu, M. Squillante, and J. L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proc. of ACM Eletronic Commerce Conference*, October 2001.
17. E. M. Maximilien and M. P. Singh. A Framework and Ontology for Dynamic Web Services Selection. *IC*, 8(5):84–93, Sept./Oct. 2004.
18. D. Menascé, V. Almeida, and L. Dowdy. *Performance by Design.* Prentice Hall, 2003.
19. D. Menasce and E. Casalicchio. QoS in Grid Computing. *IEEE Internet Computing*, July–Aug 2004.
20. M. Ouzzani and A. Bouguettaya. Efficient Access to Web Services. *IEEE Internet Comp.*, 37(3):34–44, 2004.
21. C. Patel, K. Supekar, and Y. Lee. A QoS Oriented Framework for Adaptive Management of Web Service Based Workflows. In *Proc. of DEXA 2003*, volume 2376 of *LCNS*, pages 826–835. Springer-Verlag, 2003.
22. H. G. Perros and K. H. Elsayed. Call Admission Control Schemes : A Review. *IEEE Magazine on Communications*, 1996.
23. J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.
24. T. Yu and K. J. Lin. A Broker-Based Framework for QoS-Aware Web Service Composition. In *Proc. of 2005 IEEE Int'l Conf. on e-Technology, e-Commerce and e-Service*, Mar. 2005.
25. L. Zeng, B. Benatallah, M. Dumas, J. Kalagnamam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Trans. on Soft. Eng.*, May 2004.
26. L. Zhang and D. Ardagna. SLA based profit optimization in autonomic computing systems. In *ICSOC 2004 Proc.*, pages 173–182, 2004. New York.
27. L. Zhang and D. Ardagna. SLA Based Profit Optimization in Autonomic Computing Systems. In *ICSOC 2004 Proc.*, pages 173–182, 2004.
28. L. J. Zhang and L. Bing. Requirements driven dynamic services composition for web services and grid solutions. *Journal of Grid Computing*, 2(2):121–140, 2004.

Autonomous Layer for Data Integration in a Virtual Repository*,**

Kamil Kuliberda[1,4], Radoslaw Adamus[1,4], Jacek Wislicki[1,4], Krzysztof Kaczmarski[2,4], Tomasz Kowalski[1,4], and Kazimierz Subieta[1,3,4]

[1] Technical University of Lodz, Lodz, Poland
[2] Warsaw University of Technology, Warsaw, Poland
[3] Institute of Computer Science PAS, Warsaw, Poland
[4] Polish-Japanese Institute of Information Technology, Warsaw, Poland
{kamil, radamus, jacenty, tkowals}@kis.p.lodz.pl,
kaczmars@mini.pw.edu.pl,
subieta@pjwstk.edu.pl

Abstract. The paper describes a self-operating integration mechanism for virtual repository's distributed resources based on object-oriented databases in a grid architecture. The core architecture is based on the SBA theory and its virtual updateable views. Our virtual repository transparently processes heterogeneous data producing conceptually and semantically coherent results. Our integration apparatus is explained by two examples with different types of fragmentations – horizontal and vertical. A transparent integration process exploits a global index mechanism and an independent virtual P2P network for a communication between distributed databases. Researches presented in the paper are based on a prototype integrator which is currently under development.

1 Introduction

Recently, a term *virtual repository* becomes increasingly popular in database environments as a mean to achieve many forms of transparent access to distributed resources. A virtual repository user is not aware of an actual data form as he or she gets only needed information and the best shaped for a particular use. Among many other new concepts in modern databases this branch is evolving and developing very quickly as a definite answer from science to business needs. We already know many potential applications of a dynamic data integration technologies like e-Government, e-University or e-Hospital. In a modern society data must be accessible from anywhere, at any time, regardless of a place is stored in and other similar aspects. There are many approaches which try to realize this idealistic image. Some involve a semantic data description and an ontology usage extended by logic-based programs that try to understand users needs, collect data and transform it to a desired form (RDF, RDFQL, OWL). Other commercial systems like Oracle-10G offer a flexible execution of distributed queries but they are still limited by a data model and

* This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST.

** This work has been supported by European Social Fund and Polish State in the frame of "Mechanizm WIDDOK" programme (contract number Z/2.10/II/2.6/04/05/U/2/06).

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1290 – 1304, 2006.
© Springer-Verlag Berlin Heidelberg 2006

languages not sufficient for distributed queries, in which programming suffers from inflexibility, complexity and many unpredictable complications. Our novel system offers all necessary features but it keeps programming very simple. The main idea is focused on P2P networks as a model for connecting clients and repositories combined with a powerful viewing system [2] enabling users to see data exactly in a required form [8]. This viewing system also allows an easy-to-operate mechanism for managing various data fragmentation forms as a consistent virtual whole.

The rest of the paper is organized as follows. Section 2 presents the idea of a data grid based on a virtual repository. Section 3 presents a virtual network for a data grid. Section 4 demonstrates the implementation and some examples. Section 5 concludes.

2 Distributed Data in Virtual Repository

The main difficulty of the described virtual repository concept is that neither data nor services can be copied, replicated and maintained in the global schema, as they are supplied, stored, processed and maintained on their autonomous sites [6, 7]. What is even more important, resources should be easily pluggable into the system as well as users can appear and disappear unexpectedly. Such a system by similarity to an electric grid was called a grid database or a data grid [1, 9].

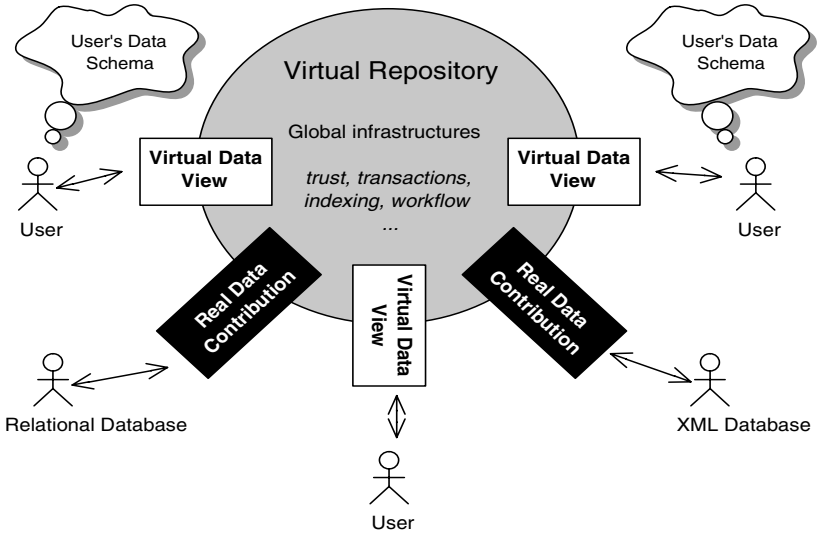


Fig. 1. The concept of a virtual repository. Users work with their own and favorite view of resources not knowing the real data or services.

However, a grid database has some limitations that significantly distinguish it from an electrical grid:

- a user may have many providers of the same service;
- a service provider can be not the same as a connection provider;
- a trust policy is more extended and more complex.

Because of these reasons a user must exactly describe his or her needs in terms of business contracts and a service provider must be able to evaluate possibilities of fulfilling these demands. Figure 1 shows the general concept. A user may plug into a virtual repository and use resources according to his or her needs, assigned privileges and their availability. In the same way resource providers may plug in and offer data or services. We may say that Internet and World Wide Web accompanied with HTML, search engines and web services work according to that idea. But we also know that searching for information, whose results are stable and repeatable, is very difficult. This environment is not sufficient for databases and effective programming tasks. A virtual repository must be achieved by an additional layer of a middleware which will supply a full transparency of resources, providers and clients [8]. The goals of the approach are to design a platform where all clients and providers are able to access multiple distributed resources without any complications concerning data maintenance and to build a global schema for the accessible data and services.

Currently, our team is working on a data grid solution developed under the international eGov-Bus project (contract no. FP6-IST-4-026727-STP). The project objective is to research, design and develop technology innovations which will create and support a software environment providing user-friendly, advanced interfaces supporting "life events" of citizen or enterprises – administration interactions transparently involving many government organizations within the European Community [16]. It can be achieved only if all existing government and para-government database resources (heterogeneous and redundant) are accessible as a homogeneous data grid. We aim to integrate existing data stores (a cost of software unification and data migration would be extremely high and a time of such an operation unacceptably long) represented by various data models and stores (e.g. relational, object-oriented, semistructured ones).

2.1 Resources Description

We separate two different data schemata. The first one is a description of resources acceptable by a virtual repository and it is called a *contributory schema.* A virtual repository must not accept *any* data but only data that it "knows" how to integrate. Another reason for limitations in this area is a consortium, which is establishing a virtual repository. It also has certain business goals and cannot accept *any* data from *any* provider. We assume that a virtual repository is created for a certain use and certain institutions [3, 5].

The other description is called a *grid schema* or a *user schema.* It describes data consumed and services used by clients. The task of a virtual repository system is to transform data from a contributory schema into a user schema performing also necessary integration and homogenization. This task may be done with different tools based on data semantics, business analysis performed by experts or any other programming techniques. In our solution we propose a view-based system with a very powerful programming and query language. The views are able to perform any data transformation and they support updates without any limitation common in other systems. We emphasise that it is the best solution for transparent data integration not possessing drawbacks of other systems.

2.2 System Architecture

The global infrastructure is responsible for managing our grid contents through access permissions, discovering data and resources, controlling location of resources, indexing whole grid attributes. The realization challenge is a method of combining and enabling free bidirectional processing of contents of local clients and resource providers participating in the global virtual store [8].

Each resource provider possesses a view which transforms its local share into an acceptable contribution. It is called *a contributory view*. It connects to the repository and may immediately share data. Providers may also use extended wrappers to existing DBMS systems [6, 7].

Similarly, a client uses *a grid view* to consume needed resources in a form acceptable for his or her applications. This view is performing the main task of data transformation and its designer must be aware of data fragmentation, replication, redundancies, etc. [3, 5] This transformation may be described by an *integration schema* prepared by business experts or being a result of an automatic semantic-based analysis. Nevertheless, we insist that in real business cases a human interference is always necessary at this stage. The problem is how to allow transparent plugging in new resources and incorporate them into existing and working views. This question is discussed in the next section.

3 Virtual Network for Distributed Resources

The idea of a database communication in a grid architecture relies on unbound data processing between all database engines plugged into a *virtual repository*. Our approach depicts not only an architecture of the network with its features, but also additional mechanisms to ensure effortless:

- users joining,
- transparent integration of resources,
- trust infrastructure for contributing participants.

The general architecture of a virtual network concept solves above three issues though middleware platform mechanisms designed for an easy and scalable integration of a community of database users. The middleware platform creates an abstraction method for a communication in a grid community. The solution creates an unique and simple database grid, processed in a parallel peer-to-peer architecture [8].

The basic concept of a transport platform is based on a well known *peer-to-peer* (P2P) architecture. Our investigations concerning distributed and parallel systems like Edutella [10], OGSA [11] bring a conclusion that a database grid should be also independent from TCP/IP stack limitations, e.g. firewalls, NAT systems and encapsulated private corporate restrictions. The network processes (such as an access to the resources, joining and leaving the grid) should be transparent for the participants. Because grid networks (also computational grids) operate on a parallel and distributed architecture, our crucial principle is to design a self-contained virtual network with P2P elements.

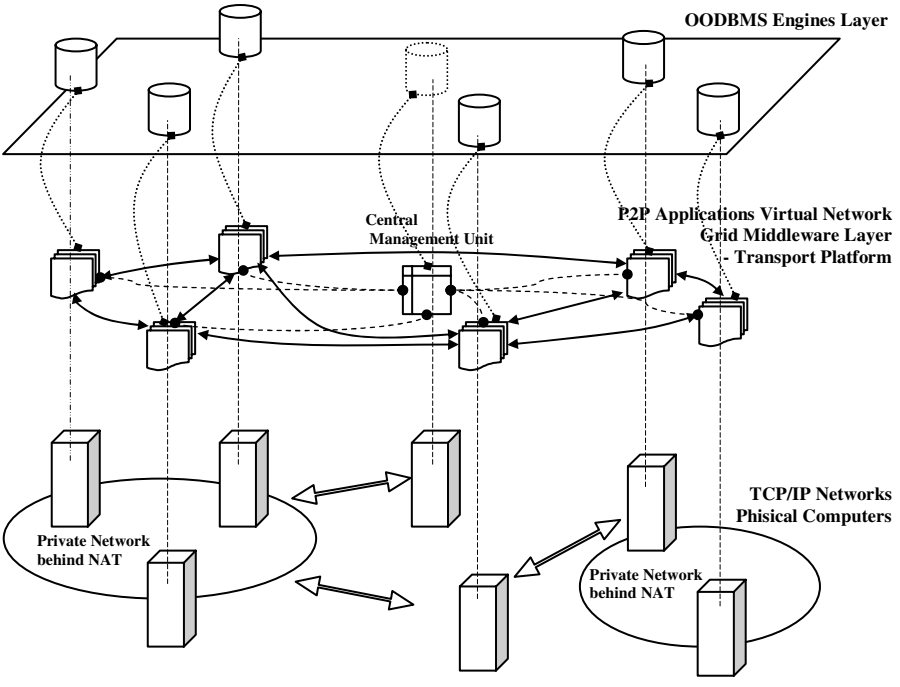


Fig. 2. Data grid communication layers and their dependencies

User's grid interfaces – in this proposition database engines – are placed over the (P2P) network middleware. For users DBMSs work as heterogeneous data stores, but in fact they are transparently integrated in the virtual repository. Users can process their own local data schemata and also use business information from the global schema available for all contributors. This part of a data grid activity is implemented above user applications available through database engines and SBQL query language [12, 13], see *OODBMS Engines Layer* in Figure 2.

In such an architecture, databases connected to the virtual network and peer applications arrange unique parallel communication between physical computers for an unlimited business information exchange.

3.1 P2P Transport Layer

The transport platform is based on JXTA Project [14, 15] as a core for a virtual network. It implements a part of P2P mechanism and provides the complete solution of a centralized and a decentralized P2P network structure. The virtual network has a centralized model and supports a management module for network mechanisms and grid processes (also a trust infrastructure). The transport platform is presented in Figure 2 as *P2P applications virtual network*. The P2P virtual network solution is a layer built on top of TCP/IP network. Its goal is to separate TCP/IP networks and make them transparent and independent of the grid community. The P2P virtual network has a complex structure containing two main blocks: JXTA Core and JXTA

Services with six additional virtual network protocols. The technical reference on how to build a P2P network using open JXTA platform is well explained in [15].

Peer-to-peer transport platform applications operate on a TCP/IP stack integrated with every operating system environment. A JXTA application implementation utilizes the Java Framework, so the peer applications of a virtual network can be used on any operating system supporting Java technology.

3.2 Central Management Unit

Our virtual network has a centralized architecture whose crucial element is a *central management unit* (CMU) – see Figure 2. In the virtual network there can exist only one CMU peer, its basic function is a responsibility for lifetime of data grid, besides it manages the virtual repository integrity and resource accessibility. Inside the P2P network level CMU is responsible for creating and managing the grid network – it means that CMU creates a *peer group* which is dedicated to linking data grid participants. CMU also maintains this peer group.

The peer group specifies the basic features of a trust infrastructure model for a database interconnection. This part of a trust infrastructure is located in the lower level (the virtual network layer) referring to the OODBMS Engines Layer rather than a direct user interface. In this case a realization consists of binding of a networking policy between an OODBMS engine and a peer application. In the current implementation we are focusing on credentials for a contribution of privileged databases in the data grid through indexing their unique identifiers inside a CMU's index. Each peer connected to the virtual network is indexed in CMU (see chapter 4).

3.3 Communication Peers and DB Engines – Implementation Details

For regular grid contributors the virtual network is equipped with participant's communication peers. They are interfaces to the virtual repository for OODBMS user's engines. Each database has its unique name in local and global schemata which is bound with the peer unique name in the virtual network. If a current database IDs are stored in the CMU, a user can cooperate with the peer group and process information in the virtual repository (according to the trust infrastructure) through a database engine with a transparent peer application. Peer unique identifiers are a part of P2P implementation of JXTA platform [14, 15].

A peer contains a separate embedded protocol for a local communication with an OODBMS engine and separate ones for cooperating with an applicable JXTA mechanism and the whole virtual network. All exceptions concerning a local database operation, a virtual network availability and a TCP/IP network state are handled by a peer application responsible for a local part of a grid maintenance. Please notice that in one local environment there are residing two separate applications (a P2P virtual network application and a database engine) composing (in grid aspects) one logical application [8].

Figure 3 presents the logical interoperability of grid components concerning the OODBMS layer and the virtual network layer. Communication can be established between:

1. Applications in the virtual network. It concerns the virtual network management and the virtual network processing (JXTA-XML datagrams). Connections in this layer rely on the protocol provided by the JXTA platform. For a communication inside the virtual network we use the multithreaded JXTA Socket layer and XML formulated datagrams for an information flow. We use different datagrams to exchange information in the P2P network and to manage peers.
2. A database implemented under ODRA (our prototype OODBMS) and a virtual network peer application used by internal XML-based protocol instances (see Figure 3). A connection between these applications implements a protocol, which exploits the JXTA communication stub and the TCP/IP socket implementation in .NET C#. Because the OODBMS prototype engine is currently implemented in .NET C#, it has forced a creation of a bridge protocol between JXTA and C#. The approach based on the bridge brings additional benefits, such as flexibility and openness for new functionalities and resource kinds, in particular, other DBMS-s, other P2P networks, other programming languages, etc.

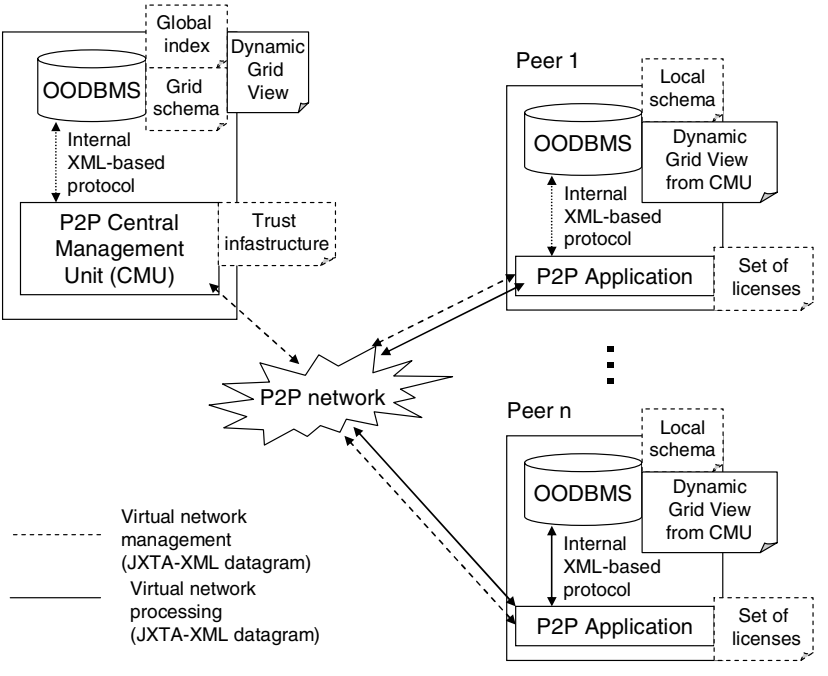


Fig. 3. Logical architecture of virtual network

All user operations on the virtual repository concern not only a local part of data stores, but also remote resources. User requests are packaged into XML-based protocol messages (equipped with some additional information about a source, a destination, etc.). The packages are sent through an XML-based protocol from a DBMS to an appropriate peer application. After receiving a XML datagram by a suitable peer application it is again wrapped according to the JXTA protocols into

JXTA-XML datagram. Such datagrams should be sent from a source peer to its destination peer application in the virtual network. A destination peer is responsible for decomposing a datagram and sending appropriate requests to a target DBMS. Notice that XML datagrams which come from a database engine have as source and destination attributes unique identifiers of participating databases. In the virtual network each database has an additional identifier as its native peer name. The current database is recognized in the virtual network by a string like *DB_name@Peer_name*.

4 Transparent Integration Via Viewing System and Global Index

The presented virtual repository based on a P2P networking must be permanently updated and this is the most important aspect of our virtual repository operation. For an easy management of the virtual repository's content we have equipped the CMU with a *global index* mechanism which covers technical networking details, management activities and it is also a tool for efficient programming in a dynamically changing environment. The global index not necessarily has to be really centralized as there are many ways to distribute its tasks. The system needs this kind of an additional control and it does not matter how it is organized. Its tasks are:

- controlling grid views available for users,
- keeping information on connected peers,
- keeping network statistics,
- registering and storing information on available resources.

The global index is a complex object which can be accessed with the SBQL syntax (as a typical database object) on every database engine plugged into the virtual repository. This means that we can evaluate queries on its contents. There is one substantial difference from processing typical virtual repository objects – as a result of an expression evaluation CMU can return only an actual content of index, like a list of online grid participants. The global index is the basic source of knowledge about the content of the virtual repository. Basing on indexed information referring the views' system we can easily integrate any remote data inside the virtual repository. If data have been indexed already, it can be transparently processed without any additional external interference. The *global index* has a specified structure which is a reflection of a *global schema* and it contains mainly additional objects for characterizing a type of a data fragmentation. These objects are dynamically managed through the views' systems whenever a virtual repository contents undergoes a change (e.g. when a resource joins or disconnects the virtual repository its local view cooperates with the global index). The global index keeps also dependencies between particular objects (complexity of the objects, etc.) as they are established in the *global schema*.

As a simple example of how the *global index* works we present grid objects named `Employee` which are registered in the central index. The grid controls two remote DBs (`DB_Krakow, DB_Lodz`), which map their objects named `Person` as the mentioned `Employee` objects in the grid schema (according to contribution views). Inside the global index list, each `Employee` object will contain two attributes as indexed location of Employee resources – the names of remote DBs – `DB_Krakow`

and DB_Lodz. Notice that the global index is a dynamic structure, thus any change in grid's available participants list will alter its content.

Each indexed object in the global index is equipped with a special object called *HFrag* (horizontal fragmentation) or *VFrag* (vertical fragmentation). Each of them keeps a special attribute named *ServerName*, whose content is a *remote object* – an identifier of a remote data resource (see Figure 6 and 8). If any new resource appears in a virtual repository, there will be added a suitable *ServerName* into the global index automatically with appropriate information about the resource.

Accessing the remote data can be achieved with calling the *global index* with:

```
GlobalIndex.Name_of_object_from_global_scheme.(Name_of_subobject).
HFrag_or_VFrag_object.ServerName_object;
```

Because every change of the virtual repository's content is denoted in the global index, accessing data in this way yields to be the only correct one.

Since every reference to a remote object must explicitly contain its location (a server it resides on), such a procedure would be too complicated for grid attendants. Moreover, it would not accomplish with transparency requirements and would complicate an automation of multiple resources integration process. Thus, we have decided to cover this stage together with automation of integration process behind a special procedure exploiting the updateable object views mechanism [4]. The process is described in the next chapter.

4.1 Examples of Transparent Accessibility of Remote Objects

Figure 4 depicts two examples of a distributed database schema to visualize our approach to data integrators. The first example realizes a transparent integration process concerning a horizontal fragmentation (Figure 4a). The complex objects Doctor and Ward are placed on three different servers, but the data structure for each sever is the same (see Figure 5), and the corresponding global index content is presented in Figure 6.

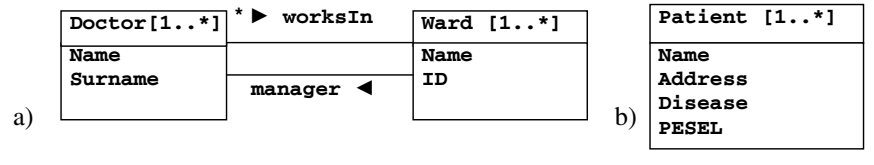


Fig. 4. Distributed databases schemata

The situation where data is horizontally fragmented in distributed resources forces merging all data as a single virtual structure, transparently achieved by all grid's clients. This process can be done by updateable object views [4] and the *union* operator. In our solution this operator is not needed, because virtual objects like *DoctorGrid* (and *WardGrid*) from a view definition procedure (see an example below) simply return a *bag* of identifiers to remote *Doctor* (or *Ward*) objects structured inside identically. The logical schema of this operation is presented in Figure 5.

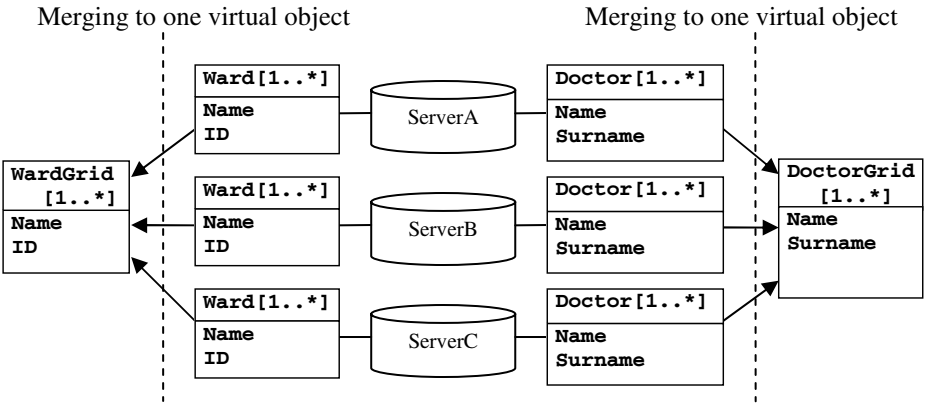


Fig. 5. Integration of distributed databases (with identical data structure) into one virtual structure for virtual repository

The updateable object views [4] action in integrating a horizontal fragmentation is presented by a following query evaluation (we retrieve names of all grid doctors who are working in the "cardiac surgery" ward):

(DoctorGrid ***where*** *WorksIn.WardGrid.Name = "cardiac surgery").Name;*

The *DoctorGrid* and *WardGrid* virtual objects definitions (through updateable object views [4]) are as follows:

```
create view DoctorGridDef {
virtual_objects DoctorGrid {
        return (GlobalIndex.Doctor.HFrag.ServerName).Doctor as doc};
                                    // return remote not fragmented objects doc
on_retrieve do {return deref(doc)}; //the result of retrieval virtual objects
        create view NameDef {
        virtual_objects Name {return doc.Name as dn}; //creating the virtual
                                    //subobjects Name of object DoctorGrid
        on_retrieve do {return deref(dn)}; //the result of retrieval virtual objects
        };

        create view SurnameDef {//…};

        create view worksInDef {
        virtual_pointers worksIn { return doc.WorksIn as wi};
        on_retrieve do {return deref(wi)};
        };
};
create view WardGridDef {
virtual_objects WardGrid {
        return (GlobalIndex.Ward.HFrag.ServerName).Ward as war};
on_retrieve do {return deref(war)};

        create view NameDef {
```

```
        virtual_objects Name {return war.Name as wn};
        on_retrieve do {return deref(wn)};
        };
        create view IDDef { //…};
};
```

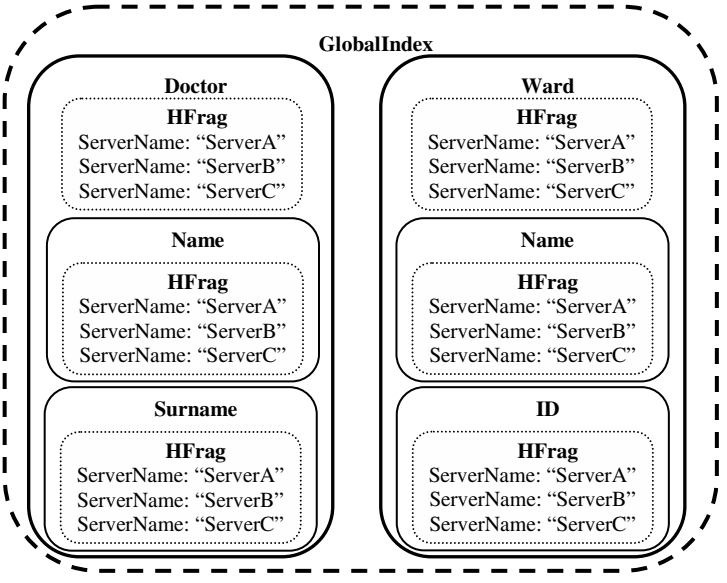


Fig. 6. The contents of CMU global index for example of horizontal fragmentation

Please notice, that for well keeping the original data schema inside a grid consentaneous to resource databases, the view must contain procedures for retrieving every subobject in an original objects' hierarchy. Therefore the views procedures, from above example, include subviews. These subviews also describe the methods of processing the called data.

The similar case concerns the GlobalIndex object, where the content also brings back the objects hierarchy like in participating databases, but in reality this is only a map for the indexing mechanism and the content is the information where the original objects and subobjects are physically stored.

Such an approach is always required where distributed resources and grid schema must have the identical representation of their contents.

The other example depicts a transparent integration process concerning a vertical fragmentation, which is more complicated, because we must join different data structures stored on physically separated servers. For solving this problem we use the *join* operator with a join predicate specific for an appropriate integration of distributed objects. The database schema (according to global schema) is presented in Figure 4b, where the complex object Patient is placed on three different servers where the data structure of stored objects is different (see Figure 7), according to this the global index content is presented on Figure 8.

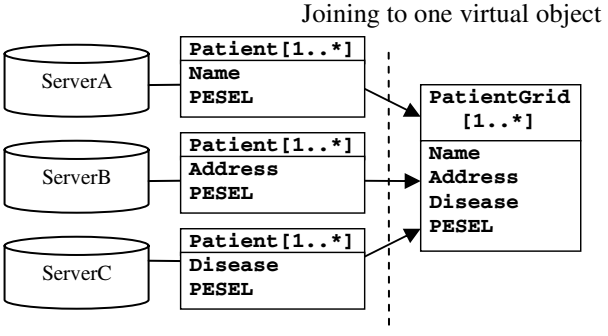


Fig. 7. Integration of distributed databases (with different data structure) into one virtual structure for virtual repository

The conclusion about a grid structure from the above example is that each server participating the virtual repository has a differential structure of stored data except for the PESEL object which has an identical content on each server. We utilize the knowledge about the object PESEL and its content to make "join" on the fragmented Patient object. The PESEL attribute is an unique identifier (predicate) for joining distributed objects into a virtual one.

This integration for a vertical fragmentation can be exemplified with a query evaluation: we retrieve names of all grid patients who suffer from cancer):

*(PatientGrid **where** Disease = "cancer").Name;*

The PatientGrid virtual objects definitions (through updateable object views [4]) are following:

```
create view PatientGridDef {
virtual_objects PatientGrid {
        return { (((GlobalIndex.Patient.VFrag.ServerName).Patient as pat).(
        ((pat where exist(Name)) as pn) join
        ((pat where exist(Address)) as pa where pa.PESEL = pn.PESEL) join
        ((pat where exist(Disease)) as pd where pd.PESEL = pn.PESEL)).(
                pn.Name as Name, pn.PESEL as PESEL, pa.Address as Address,
                pd.Disease as Disease)) as Patients };
                // return remote not fragmented objects Patients
on_retrieve do {return deref(Patients)}; //the result of retrieval virtual objects

        create view PatNameDef {
        virtual_objects Name {return Patients.Name as PatN};
        on_retrieve do {return deref(PatN)};
        };

        create view PatDiseaseDef {
        virtual_objects Disease {return Patients.Disease as PatD};
        on_retrieve do {return deref(PatD)};
        };
};
```

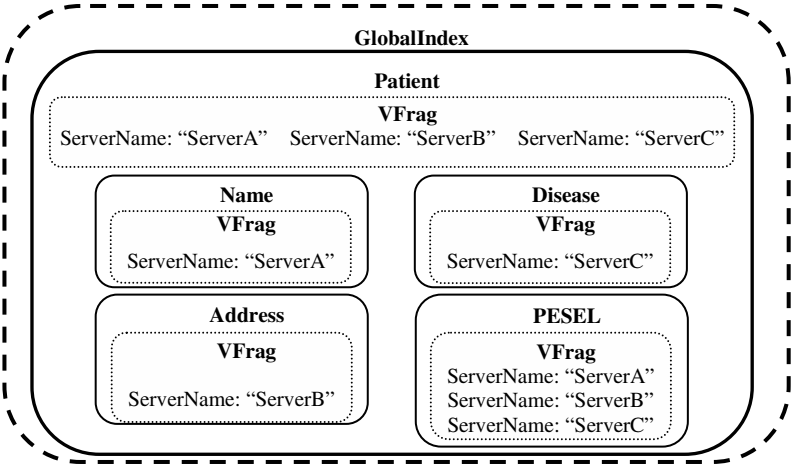


Fig. 8. The contents of CMU global index for example of vertical fragmentation

In the above examples we create virtual objects explicitly, what implies that a grid designer must be aware of fragmented objects in the grid schema. He or she does not need any knowledge of the fragmentation details, but must know which objects are fragmented. The rest of the integration process is executed automatically through SBQL syntactic transformations. Basing on the presented approaches it is easy to define an integration mechanism of objects fragmented both horizontally and vertically. In order to do so, we must combine them into a specific model. Also, we are able to extend our proposal for mixing fragmentation for different resources where some resources can store an object with a few different subobjects. Current object also can store simultaneously identical subobjects in different resources as a replication of the data. For this approach we have prepared the GlobalIndex object and we have equipped it with *HFrag* and *VFrag* subobjects. Our research aims to extend the global index structure with some information about a replication and redundant resources for reach transparency in a grid over such data structures and dependencies.

5 Conclusions and Future Work

We have presented a generic approach to a transparent integration of distributed data in a virtual repository mechanism. Our solution utilizes a consistent combination of several technologies, such as P2P networks developed on the ground of JXTA, a SBA object-oriented database and its query language SBQL with virtual updateable views. Our preliminary implementation solves a very important issue of independence between technical aspects of distributed data structure managing (including additional issues such as participants' incorporation, resource contribution) and a logical virtual repository content scalability (a business information processing). We expect that the presented methods of integration of fragmented data will be efficient and fully scalable. We also expect that due to the power of object-oriented databases and SBQL

such a mechanism will be more flexible than other similar solutions. The prototype is fully implemented and preliminarily tested.

The mentioned grid architecture is described in details in [6, 7, 8]. The presented P2P transport layer (also depicted as a grid/transport middleware) is independent of an application composition which provides a transparent communication between databases (grid data resources). Database's UI and engines are equipped with a separate protocol for a connection with P2P applications (at the transport layer level) – the details are described in [8] and in Figure 3.

Currently we are working on extending the presented idea to achieve a generic integration process through new functionalities like managing the mixed form of fragmentations, data replicas and redundancies of resources.

References

1. Foster I., Kesselman C., Nick J., Tuecke S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum, June 22, 2002.
2. Kaczmarski K., Habela P., Kozakiewicz H., Subieta K.: Modeling Object Views In Distributed Query Processing on the Grid. OTM Workshops 2005, Springer LNCS 3762, 2005, pp.377-386
3. Kaczmarski K., Habela P., Subieta K.: Metadata in a Data Grid Construction. 13th IEEE International Workshops on Enabling Technologies (WETICE 2004), IEEE Computer Society 2004, pp. 315-316
4. Kozankiewicz H.: Updateable Object Views. PhD Thesis, 2005, http://www.ipipan.waw.pl/~subieta/, Finished PhD-s
5. Kozankiewicz H., Stencel K., Subieta K.: Implementation of Federated Databases through Updateable Views. Proc. EGC 2005 - European Grid Conference, Springer LNCS 3470, 2005, pp.610-619
6. Kuliberda K., Wislicki J., Adamus R., Subieta K.: Object-Oriented Wrapper for Relational Databases in the Data Grid Architecture. OTM Workshops 2005, Springer LNCS 3762, 2005, pp.367-376
7. Kuliberda K., Wislicki J., Adamus R., Subieta K.: Object-Oriented Wrapper for Semistructured Data in a Data Grid Architecture. 9th International Conference on Business Information Systems 2006, Klagenfurt Austria, Proceedings in Lecture Notes in Informatics (LNI) vol. P-85, GI-Edition 2006, pp.528-542
8. Kuliberda K., Kaczmarski K., Adamus R., Błaszczyk P., Balcerzak G., Subieta K.: Virtual Repository Supporting Integration of Pluginable Resources, 17th DEXA 2006 and 2nd International Workshop on Data Management in Global Data Repositories (GRep) 2006, Proceedings in IEEE Computer Society, to appear.
9. Moore R., Merzky A.: Persistent Archive Concepts. Global Grid Forum GFD-I.026. December 2003.
10. Nejdl W., Wolf B., Qu C., Decker S., Sintek M., Naeve A., Nilsson M., Palmer M., Risch T.: EDUTELLA, a P2P networking infrastructure based on RDF. Proc. Intl. World Wide Web Conference, 2002.
11. Open Grid Services Architecture, Data Access and Integration Documentation, http://www.ogsadai.org.uk
12. Subieta K.: Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, 2004 (in Polish)

13. Subieta: Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL). http://www.ipipan.waw.pl/~subieta, Description of SBA and SBQL, 2006
14. The JXTA Project Web site: http://www.jxta.org
15. Wilson B.: JXTA Book, http://www.brendonwilson.com/projects/jxta/
16. eGov-Bus, http://www.egov-bus.org

An Instrumentation Infrastructure for Grid Workflow Applications*

Bartosz Balis[1], Hong-Linh Truong[3], Marian Bubak[1,2], Thomas Fahringer[3], Krzysztof Guzy[1], and Kuba Rozkwitalski[2]

[1] Institute of Computer Science AGH, Krakow, Poland
{balis, bubak}@agh.edu.pl, kubaroz@gmail.com
[2] Academic Computer Centre – CYFRONET AGH, Krakow, Poland
[3] Institute of Computer Science, University of Innsbruck, Austria
{truong, tf}@dps.uibk.ac.at

Abstract. Grid workflows are very often composed of multilingual applications. Monitoring such multilingual workflows in the Grid requires an instrumentation infrastructure that is capable of dealing with workflow components implemented in different programming languages. Moreover, Grid workflows introduce multiple levels of abstraction and all levels must be taken into account in order to understand the performance behaviour of a workflow. As a result, any instrumentation infrastructure for Grid workflows should assist the user/tool to conduct the monitoring and analysis at multiple levels of abstraction. This paper presents a novel instrumentation infrastructure for Grid services that addresses the above-mentioned issues by supporting the instrumentation of multilingual Grid workflows at multiple levels of abstraction using a unified, highly interoperable interface.

Keywords: grid, monitoring, instrumentation, legacy applications.

1 Introduction

Grid workflows based on modern service-oriented architecture (SOA) are often multilingual, i.e. combine java-based services with invocations of legacy code. Furthermore, workflows can be analyzed at different levels of abstraction, ranging from the entire workflow, to workflow activites, to code regions of invoked applications. The Grid and SOA with their promise of interoperable inter-enterprise integration further increase workflows' heterogenity in the sense of programming languages and execution environments.

Instrumentation of Grid workflows is a required step in order to collect monitoring data for debugging or performance analysis purposes. However, due to the aforementioned Grid workflows' characteristics, instrumentation poses both a conceptual and a technical challenge. The user needs a unified approach to specify instrumentation and monitoring requests for different pieces of his/her workflow, as well as a common representation of monitoring results, regardless of underlying implementation languages and execution environments. Thus, not only the monitoring of workflows in the Grid

* The work described in this paper is partially supported by the EU IST-2002-511385 project K-WfGrid and by the Polish SPUB-M grant.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1305–1314, 2006.
© Springer-Verlag Berlin Heidelberg 2006

requires different instrumentation techniques but also interactions between various services involved in the instrumentation and monitoring have to use a well-defined, highly interoperable interface. Often these requirements are conflicting and cannot be easily unified into a single framework. This paper presents a novel instrumentation infrastructure for Grid services that addresses the above-mentioned issues by supporting the dynamically enabled instrumentation of multilingual Grid workflows at multiple levels of abstraction using a unified, highly interoperable interface.

The rest of this paper is organized as follows: Section 2 discusses concepts of instrumentation at multiple levels. Section 3 presents languages describing instrumented applications and instrumentation requests. Section 4 details instrumentation techniques. We present experiments in Section 5. Section 6 outlines the related work, followed by a summary of the paper and an outlook to the future work in Section 7.

2 Multiple Levels of Instrumentation

Grid workflows, in our view, introduce multiple levels of abstraction including *workflow, workflow region, activity, invoked applicaton* and *code region* [1]. To understand the performance behaviour of Grid workflows, it is necessary to analyze and correlate various performance metrics collected at these levels. Therefore, performance monitoring and analysis tools for Grid workflows have to operate at different levels and to correlate performance metrics between these levels. To provide performance metrics at workflow, workflow region and activity levels, the tools have to conduct the monitoring and measurement by instrumenting the enactment engine (EE) which is responsible for executing workflows. On the other hand, for analyzing metrics at invoked applications and code regions, the tools have to instrument invoked applications.

We instrument the enactment engine by statically inserting sensors into its source code in order to monitor execution behaviour of workflows and workflow activities. For applications invoked within workflow activities, we apply dynamically enabled instrumentation. That is the instrumentation is conducted before the runtime and the measurement is dynamically enabled at runtime. Even though dynamically enabled instrumentation mostly supports measuring performance metrics of program units and function calls, but not of arbitrary code regions, for Grid workflow applications, we believe that measuring the performance at a level of program unit and function call should be enough. Due to the complex and distributed nature of Grid workflows, we believe that when analyzing the performance of workflow applications, most users are interested in observing the performance at the level of workflow, program unit and function call, rather than at loop or statement levels. The invoked applications, e.g., Grid service operations or legacy scientific programs, are also diverse and can be multilingual. Therefore, we must have a common infrastructure that allows us to instrument such applications. While we could reuse existing instrumentation techniques, we have to provide a common interface for performing the instrumentation of different types of applications.

One of important goals of our work is to support various forms of legacy code (libraries, forked jobs, parallel applications). Supporting legacy code is important, since today most mature applications are written in Fortran, C or C++ and rather adapted to the Grid than re-engineered to Java-based services.

3 Program Representations and Instrumentation Requests

As discussed in the previous section, the instrumentation system has to support multiple levels of abstraction and to work with diverse and multilingual applications. Therefore, we need various, quite different, instrumentation techniques, each suitable for a specific type of applications and abstraction level. Besides the detailed instrumentation techniques, we must address interface between the instrumentation requestor and the instrumentation system. The interface between them answers two basic questions: (1) how the structure of multilingual applications are represented?, and (2) how the instrumentation requests are specified? The main desire in answering these questions is that the requestor should treat different applications implemented in different languages using a unified means. To this end, we have to provide a neutral way to represent objects to be instrumented and requests used to control the instrumentation. Our approach is to use SIR [2] for representing applications and to develop XML-based instrumentation requests based on that representation.

Standardized Intermediate Representation: We have developed an XML-based representation named SIRWF (Standardized Intermediate Representation for Workflows) to describe the structure of invoked applications of workflow activities; SIRWF is developed based on SIR [2]. The main objective is to express information of different types of applications, which is required by instrumentation systems, using a single intermediate representation. The SIR represents most interesting information for instrumentation, such as program units and functions, while shields low level details of the application from the user. SIRWF, a simplified version of SIR, currently can represent invoked applications and code regions (at program unit and function call levels) of workflows.

Workflow Instrumentation Request language (WIRL): Given application structure represented in SIRWF, we develop an XML-based language for specifying instrumentation requests named WIRL (Workflow Instrumentation Request Language). WIRL, based on IRL [3], is an XML-based language. A WIRL request consists of experiment context and instrumentation tasks. Experiment context (e.g., activity identifier, application name, computational node, etc.) identifies applications to be instrumented and includes information used to correlate monitoring data to the monitoring workflow. Instrumentation tasks specify instrumentation operations. Examples of instrumentation tasks can be a request for all instrumented functions within an application, to enable or disable an instrumented code, etc. The current requests include (1) `ATTACH` to attach/prepare the instrumentation of a given application, (2) `GETSIR` to get SIR of the application, (3) `ENABLE` to enable/instrument a code region, (4) `DISABLE` to disable/deinstrument a code region, and (5) `FINALIZE` to finish the instrumentation.

Code region identifiers are obtained from the list of functions provided by the instrumentation system. In an instrumentation request, performance metrics as well as user-defined events can be specified. Performance metrics are defined by the metric ontology [1] while user-defined events include event names with associated event attributes (names and values). For the instrumentation of user-defined events, the request also allows the client to specify the location to which the event probes should be inserted, e.g., before or after a function call.

Further description of monitoring and instrumentation interfaces as well as data representations can found in [4].

4 Instrumentation Infrastructure

Fig. 1 depicts a basic architecture in which entities involved in instrumentation and monitoring of a sample workflow application are shown. In this figure, a workflow enactment engine *GWES* (Grid Workflow Execution Service) [5] creates a workflow activity (Activity 1) which in turn invokes an MPI application. *GEMINI* [6] is the monitoring and instrumentation infrastructure which implements and adapts various instrumentation techniques and systems in a single and unified framework. GEMINI executes instrumentation and data subscription requests and notifies subscribers about new data. *Portal* is the user interface from which the user submits workflows, as well as controls the instrumentation, performance monitoring and analysis. The figure puts emphasis on how the events related to the execution of the workflow flow from various distributed locations: GWES, an invoked application running in a workflow activity, and a legacy MPI application. The framework needs to collect those pieces of data, correlate the collected data, and present them to clients, e.g. in the portal.

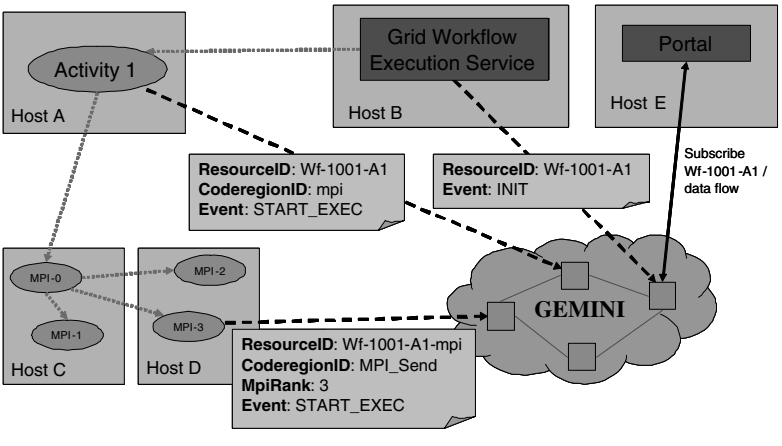


Fig. 1. Instrumentation framework architectuire

Fig. 2 presents the course of action in a basic instrumentation and monitoring scenario (MPI part of the application being omitted for simplification). In this scenario, after the user submits a workflow to GWES, GWES initializes the workflow, enacts the consecutive activities and invokes *invoked applications* performing the task of activities. GWES also sends some events to GEMINI concerning workflow status (not all shown in the figure). Within GWES, every single workflow or activity instance is associated with a unique ID. When an invoked application of an activity instance is started, GWES sends the unique ID of the activity instance to the invoked application (see section 4.2). All IDs associated with activity instances can be obtained from GWES through the Portal. When the invoked application starts its execution, it registers itself to GEMINI by

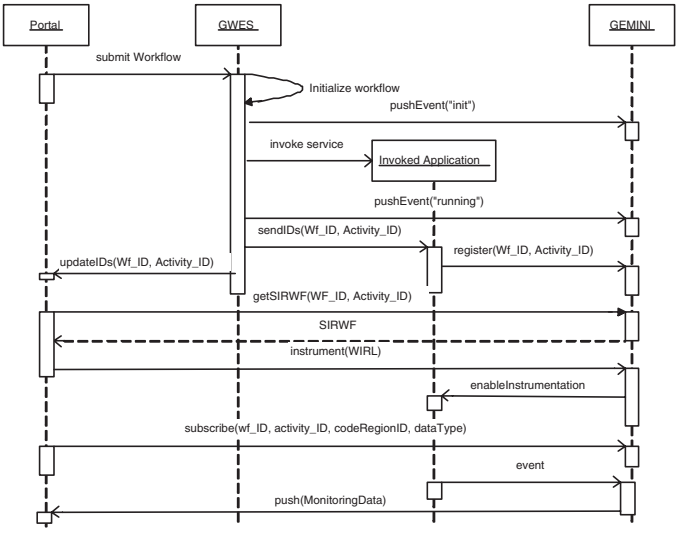


Fig. 2. Multiple levels of instrumentation scenario

passing its unique ID. Meanwhile, by using the ID, the user can select invoked applications of interest and then request SIRWF of the selected invoked application in order to perform the monitoring and measurement. Based on the SIRWF, the user can select code regions and specify interesting metrics to be evaluated. A WIRL request will be created and sent to GEMINI which activates the measurement. In order to receive the data, the user specifies a PDQS (Performance Data Query and Subscription) request and sends it to GEMINI. When the execution of the invoked application reaches the instrumentation code, events containing monitoring data are generated and monitoring data is sent to GEMINI which in turn pushes the data to the subscriber (user).

4.1 Instrumentation at Multiple Levels of Abstraction

Instrumentation of Workflow Execution Service: Monitoring data at the level of workflow, workflow region and activity can be obtained from the GWES. GWES controls and executes the workflow and its activities so it can provide a number of events relevant to the execution of workflows such as events indicating the workflow and activity state changes (*initiated, running, complete*, etc.) Currently GWES is manually instrumented by inserting sensors, which collect monitoring data, into the source code. Applying only static instrumentation to GWES should be sufficient because only a few places in the GWES code have to be instrumented and as the overhead of invoking the probe functions is minimal, the instrumentation can be permanently installed and active.

Instrumentation of Invoked Applications and Code Regions of Java-based Services: In doing this, we currently employ byte-code instrumentation techniques and dynamically control the measurement process at runtime. Sensors are inserted into the byte-code using the BCEL tool [7]. At the same time SIRWF is created and saved with the

class. Instrumented versions of classes are placed in different locations than the original ones. Instrumentation of an individual code region is conditional. Dynamic activation and deactivation of the instrumentation amounts to changing the value of a condition variable at runtime.

Instrumentation of Invoked Applications and Code Regions of Legacy Code: The possibility to use legacy code in services is twofold: (1) as legacy libraries invoked by means of JNI (Java Native Interface) calls or (2) as legacy jobs submitted from within a service. The first case can be handled in the same way as instrumentation of code regions described in Section 4.1, only tools and libraries for legacy code have to be developed. In the latter case, the legacy jobs invoked from services are often parallel applications, for example computationally intensive simulations. In our framework, monitoring of such applications will be handled by an external monitoring system, the OCM-G [8]. To be integrated seamlessly with the GEMINI, OCM-G has to provide SIRWF for legacy jobs, be able to handle WIRL-based instrumentation requests and represent monitoring in GEMINI data representation. To this end, we have substantially extended OCM-G, for example by introducing a SIRWF generation, and a GEMINI OCM-G-sensor for integrating OCM-G with GEMINI in terms of interfaces and data representation. The detailed process of OCM-G adaptation, instrumentation, and SIRWF generation for legacy programs is presented in [9].

Selection of Instrumentation Scope: A general problem in low-level monitoring and instrumentation of applications is how to select the proper parts of the applications to be instrumented and taken into account in monitoring. The application usually consists of hundreds of code units (classes, functions, files and libraries) of which many are irrelevant for the user, since they are not part of application's logic but belong to external libraries, etc. This problem is valid both for Java applications and e.g. C applications. The method, which we currently employ, is to explicitly specify the subset of code units (e.g. jars, classes or C-files) that are of interest and only those will be instrumented and included in SIRWF.

4.2 Passing Correlation Identifier to Invoked Applications

Monitoring data concerning a single workflow is generated in distinct places, mainly within GWES and invoked applications. In order to correlate those different pieces of data together, and to be able to correlate data requests with the incoming monitoring data, we need to associate the monitoring data with a unique ID, a 'signature'. In our case, this ID constitutes the combination of workflow and activity IDs, both generated by GWES. Obviously those IDs must also be passed to invoked applications which are web service operations.

The described problem is a case of a general issue of passing *execution context* to web services. However, Web services currently do not support any context information. For pure, stateless web services, the only information passed to an invoked service is via operation's parameters. There are a few possibilities to pass context information, as discussed in [10], for example: (a) by extending service's interface with additional parameter for context information, (b) by extending the data model of the data passed to a service, (c) by inserting additional information in SOAP header, and reading it from

within the service. An additional option is to use the state provided by a service, but this only works for services that support state, e.g. WSRF-based ones.

We have used the SOAP-header approach since it is transparent and does not require the involvement of services' developers. Thus, when invoking services, GWES inserts additional information to SOAP headers which is accessed in the services by instrumentation initialization functions to obtain workflow and activity identifiers.

4.3 Organizing and Managing Monitoring Data

The instrumentation supports gathering monitoring data at multiple levels; data collected within an experiment (e.g., of a workflow) is determined through IDs. Currently the framework does neither support merging data from different levels nor provide a storage system managing that data, although part of monitoring data is temporarily stored in distributed GEMINI monitoring services. The main goal of the framework is to support online performance monitoring and analysis so that the monitoring data is just returned to clients (e.g., performance analysis services) via query or subscription and the clients will analyze the monitoring data.

5 Experiment

We have implemented a prototype of our instrumentation infrastructure that currently supports, among others, generation of SIRWF, static insertion of probes for Java and C applications, and dynamic control (activation/deactivation) of probes based on SIRWF representation. In this section we present an experiment to illustrate our concepts.

In our experiment, we use GWES to invoke a service for numerical integration. The actual computation is done by a parallel MPI job invoked from the service which performs a parallel adaptive integration algorithm. The monitoring and instrumentation of

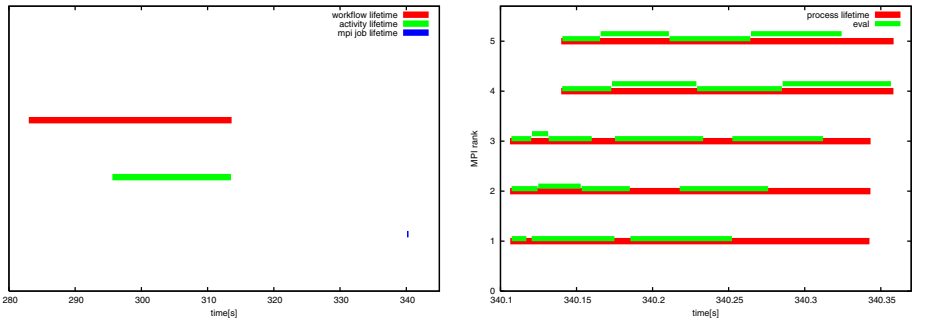
```
<sirApp><experiment>...</experiment>
 <sir><unit type="subroutine"
    name="numeric.pa_integrate">
    <coderegion type="call" id="1">
     <calee name="mpi_integrate"/>
    </coderegion></unit>
   <unit type="subroutine"
                name="mpi_integrate">
    <coderegion type="call" id="2">
      <calee name="main"/>
    </coderegion></unit>
   <unit type="function" name="main">
    <coderegion type="call" id="3">
      <calee name="MPI_Send"/>
    </coderegion>
      ...
    <coderegion type="call" id="14">
      <calee name="eval"/>
    </coderegion>
      ...
</unit></sir></sirApp>
```

(a)

```
<wirl><request name="ENABLE">
 <experiment>
   <wfInstanceID>wf1</wfInstanceID>
   <pu><activityID>wf1_a1</activityID>
       <invokedAppId>wf1_a1_ia1
       </invokedAppId>
   </pu>
 </experiment>
 <task><coderegion name="main" id="2"/>
   <coderegion name="eval" id="14"/>
    <events location="BEFORE">
      <event>
        <eventname>BEGIN_EXEC</eventname>
      </event>
    </events>
    <events location="AFTER">
      <event><eventname>END_EXEC</eventname>
      </event>
    </events>
  </task>
</request></wirl>
```

(b)

Fig. 3. Instrumentation: (a) SIRWF (simplified) and (b) WIRL (simplified)



(a) (b)

Fig. 4. Monitoring results: (a) Workflow perspective (b) Legacy perspective

the MPI application is realized by the OCM-G monitoring system which is integrated with GEMINI, as described in [9].

In our experiment, we wanted to obtain an event trace showing the consecutive stages of execution of the workflow – from the GWES events to the execution of code regions in the MPI application; we were interested in parts in which the actual computation was performed. After the application has been started, it was suspended so that required measurements could be set up from the beginning. The SIRWF obtained for the invoked application is shown in Fig. 3 (a). The `experiment` section identifies a particular runtime part of the workflow (e.g. an invoked application, a legacy job). Within invoked application `numeric.pa_integrate`, a subroutine named `mpi_integrate`. This subroutine actually invokes a legacy application written in MPI. The legacy code consists of a function named `main` and other functions such as `eval` and `MPI_Send`. In the monitoring view, we are interested in monitoring the following calling chain: `mpi_integrate` → `main` → `eval`. Using information in the SIRWF, we decide to instrument the `numeric.pa_integrate` invoked application, the `main` functions and the calls to `eval` function in all MPI processes. Part of the corresponding WIRL submitted to GEMINI is shown in Fig. 3. As GWES is statically instrumented, events from GWES are always reported to GEMINI.

When the instrumentation has been set up, we send a subscription request to GEMINI to obtain all events related to the monitored workflow. We show the results in a form of bars representing the time span of different execution units. Fig. 4 (a) shows a comparison between the execution of the entire workflow, the single activity that has been executed, and the MPI job executed from the activity. As we can see, the delay related to workflow submission is enormous in comparison to the actual computation. Fig. 4 (b) uses a different scale to show the time breakdown for individual code regions inside the MPI processes. Multiple calls to the `eval` function are due to the iterative process of adaptive integration.

6 Related Work

Techniques like source code, binary or dynamic instrumentation are well-known and widely used such as in TAU [11], Paradyn [12], Dyninst [13]. Many efforts have been

spent on Java byte-code and dynamic instrumentation, e.g., [14,15,16,17]. The main aspect in which our work differs from these tools is that we focus on developing extensible and interoperable instrumentation interfaces, e.g. WIRL and SIRWF, for performance measurement of Grid applications and on integrating various instrumentation mechanisms for Grid applications. While the above-mentioned tools also target to large scale, long-run applications, they support and are developed on high performance parallel systems where issues of heterogeneity, loosely integration, multilingual applications, and interoperability are not centered points. PPerfGrid [18] geographically collects and exchanges performance data of parallel programs by using Grid services but it does not address the instrumentation and monitoring of Grid workflow-based applications. To our best knowledge, there is no instrumentation infrastructure that supports multiple levels of instrumentation for multilingual Grid workflows.

The APART working group proposes the SIR as a high level means to describe structure of applications to be instrumented by using XML [2]. However, APART SIR represents only single applications, not workflows. We adopt SIR, simplifying and using it for our instrumentation. There are works on proposing XML-based requests for instrumentation such as MIR[19] and MRI [20]. However, those requests are used for instrumentation of individual invoked applications, rather than for workflows.

AKSUM tool supports dynamic instrumentation of distributed Java applications [15]. The instrumentation also uses SIR and MIR. However, it neither supports Grid- nor workflow-based applications. Moreover, the instrumentation is limited to Java code only while our instrumentation supports legacy code (e.g., C/Fortran) as well.

7 Conclusion and Future Work

The contribution of this paper is a novel framework for instrumentation of multilingual and Grid-based workflows. The instrumentation and monitoring of complex, multilingual, workflow of Grid services are handled seamlessly in an integrated system. By employing GEMINI, the concepts of SIR and WIRL, and various instrumentation techniques, we are able to provide a unified and comprehensive view on the execution of workflow applications at various levels.

At the moment we have a working prototype of our infrastructure. We are working on full integration of service-level and legacy-level instrumentation, full implementation of SIRWF, transparency of application monitoring, and a fully reliable passing of workflow and activity IDs to all parts of the workflow application. Moreover, the scalability and robustness of the system need to be evaluated.

References

1. Truong, H.L., Fahringer, T., Nerieri, F., Dustdar, S.: Performance Metrics and Ontology for Describing Performance Data of Grid Workflows. In: Proceedings of IEEE International Symposium on Cluster Computing and Grid 2005, 1st International Workshop on Grid Performability, Cardiff, UK, IEEE Computer Society Press (2005)
2. Seragiotto, C., Truong, H.L., Fahringer, T., Mohr, B., Gerndt, M., Li, T.: Standardized Intermediate Representation for Fortran, Java, C and C++ Programs. Technical report, Institute for Software Science, University of Vienna (2004)

3. Truong, H.L., Fahringer, T.: SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid. Scientific Programming **12** (2004) 225–237 IOS Press.
4. Truong, H.L., Balis, B., Bubak, M., Dziwisz, J., Fahringer, T., Hoheisel, A.: Towards Distributed Monitoring and Performance Analysis Services in the K-WfGrid Project. In: Proc. PPAM 2005 Conference, Poznan, Poland, Springer (2006) 157–163
5. Neubauer, F., Hoheisel, A., Geiler, J.: Workflow-based Grid Applications. Future Generation Computer Systems **22** (2006) 6–15
6. Balis, B., Bubak, M., Dziwisz, J., Truong, H.L., Fahringer, T.: Integrated Monitoring Framework for Grid Infrastructure and Applications. In: Innovation and the Knowledge Economy. Issues, Applications, Case Studies, Ljubljana, Slovenia, IOS Press (2005) 269–276
7. BCEL: Byte Code Engineering Library (2006) http://bcel.sourceforge.net.
8. Balis, B., Bubak, M., Radecki, M., Szepieniec, T., Wismüller, R.: Application Monitoring in CrossGrid and Other Grid Project s. In: Grid Computing. Proc. Second European Across Grids Conference, Nicosia, Cyprus, Springer (2004) 212–219
9. Baliś, B., Bubak, M., Guzy, K.: Fine-grained Instrumentation and Monitoring of Legacy Applications in a Service-Oriented Environment. In: Proc. 6th Intl Conference on Computational Science – ICCS 2006. Volume 3992 of LNCS., Reading, UK, Springer (2006) 542–548
10. Brydon, S., Kangath, S.: Web Service Context Information. (2005) https://bpcatalog.dev.java.net/nonav/soa/ws-context/index.html.
11. Sheehan, T., Malony, A., Shende, S.: A runtime monitoring framework for tau profiling system. In: Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments(ISCOPE's 99), San Franciso (1999)
12. Miller, B., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer **28** (1995) 37–46
13. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching. The International Journal of High Performance Computing Applications **14** (2000) 317–329
14. Guitart, J., Torres, J., Ayguad, E., Labarta, J.: Java instrumentation suite: Accurate analysis of java threaded applications (2000)
15. Seragiotto, C., Fahringer, T.: Performance Analysis for Distributed and Parallel Java Programs. In: Proceedings of IEEE International Symposium on Cluster Computing and Grid 2005, Cardiff, UK, IEEE Computer Society Press (2005)
16. Factor, M., Schuster, A., Shagin, K.: Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 288–300
17. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: PLDI. (2001) 168–179
18. Hoffman, J.J., Byrd, A., Mohror, K., Karavanic, K.L.: Pperfgrid: A grid services-based tool for the exchange of heterogeneous parallel performance data. In: IPDPS, IEEE Computer Society (2005)
19. Seragiotto, C., Truong, H.L., Fahringer, T., Mohr, B., Gerndt, M., Li, T.: Monitoring and Instrumentation Requests for Fortran, Java, C and C++ Programs. Technical Report AURORATR2004-17, Institute for Software Science, University of Vienna (2004)
20. Kereku, E., Gerndt, M.: The Monitoring Request Interface (MRI). In: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006). (2006)

A Dynamic Communication Contention Awareness List Scheduling Algorithm for Arbitrary Heterogeneous System*

Xiaoyong Tang**, Kenli Li, Degui Xiao, Jing Yang, Min Liu, and Yunchuan Qin

School of Computer and Communication, Hunan University, Changsha 410082, China
tang_313@163.com

Abstract. Task scheduling is an essential aspect of parallel process system. Most heuristics for this NP-hard problem assume fully connected homogeneous processors and ignore contention on the communication links. Actually, contention for communication resources has a strong influence on the execution time of a parallel program in arbitrary network topology heterogeneous system. This paper investigates the incorporation of contention awareness into task scheduling. The innovation is the idea of dynamic scheduling edges to links, which we use the earliest communication finish time search algorithm based on shortest-path search algorithm. The other novel idea proposed in this paper is scheduling priority based on recursive rank computation on heterogeneous arbitrary architectures. The comparison study, based on randomly generated graphs, shows that our scheduling algorithm significantly surpass classic and static communication contention awareness algorithm, especially for high data transmission rate parallel application

Keywords: list scheduling; arbitrary network topology heterogeneous system; DAG; communication contention; parallel algorithm.

1 Introduction

Heterogeneous system has become widely used for scientific and commercial applications such as high-definition television, medical imaging, or seismic data process and weather prediction. These systems require a mixture of general-purpose machines, programmable digital machines, and application specific integrated circuits [1].A heterogeneous system involves multiple heterogeneous modules connect by arbitrary network and interact with one another to solve a problem. More and more evidence shows that scheduling parallel tasks is a key factor in obtaining high performance in such system. The common objective of scheduling is to map tasks onto machines and order their execution so that task precedence requirements are satisfied and a minimum schedule length (makespan).

A popular representation of a parallel application is the directed acyclic graph (DAG) in which the nodes represent application tasks and the directed arcs or edges

* Supported by the National Natural Science Foundation of China under Grant Nos. 60273075 and the Key Project of Ministry of Education of China under Grant No.05128.
** Corresponding author.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1315 – 1324, 2006.
© Springer-Verlag Berlin Heidelberg 2006

represent inter-task dependencies, such as task's precedence. As the problem of finding the optimal schedule is NP-complete [2]. The most common heuristic for DAG scheduling is the traditional list scheduling algorithm. These algorithms may be broadly classified into the following four categories:

- Task-duplication-based (TDB) scheduling [3].
- Bound number of processors (BNP) scheduling [4].
- Unbounded number of clusters (UNC) scheduling [5].
- Arbitrary network topology (ANP) scheduling [6-9].

The basic idea of list scheduling is to assign priorities to the tasks of the DAG and place the tasks in a list arranged in descending order of priorities. A task with a higher priority is scheduled before a task with a lower priority and ties are broken using some method, such as his child's priority. However, most list scheduling algorithms were designed for homogeneous system and not very suitable for heterogeneous system.

Several variant list scheduling algorithms have been proposed to deal with the heterogeneous system, for example mapping heuristic (MH) [8], dynamic-level scheduling (DLS) algorithm [1], levelized-min time (LMT) algorithm [10], Critical-Path-on-a-Machine (CPOP) algorithm and heterogeneous earliest-finish-time (HEFT) algorithm [9,11]. The HEFT algorithm significantly outperforms the DLS algorithm, MH, LMT and CPOP algorithm in terms of average schedule length ratio, speedup, etc [9,11]. The HEFT algorithm selects the task with the so-called highest upward rank value at each step and assigns the selected task to the processor which minimizes its earliest finish time with an insert-based policy. When computing the priorities, the algorithm uses the task's mean computation time on all processors and the mean communication rates on all links.

However, most of these works assume that the processors are fully connected, which means that there is no communication content. Actually, most heterogeneous system can't meet this condition and their processors can be linked by an arbitrary network topology (ANP). In this field, very few algorithms have proposed to meet arbitrary processor network [6-9,11] and incorporate link contention in the heuristic scheduling [6,7]. However, as Macey and Zomaya [12] showed, the awareness of link contention is significant to produce accurate and efficient schedules. In literature [6,7], Oliver Sinnen and A. Sousa propose a communication contention-awareness algorithm and their experiments show that their algorithm surpass significantly than no contention-awareness classic algorithms.

Nevertheless, their shortest path search algorithm is naturally based on static shortest path. As some tasks schedule to processors, static shortest path can not be the best route because communication inevitably occur in some links and the data arrive time may be postponed. On the other hand, his scheduling priorities simple use bottom level which is efficiently for fully connected homogeneous system and not suit for arbitrary network topology heterogeneous system.

This paper investigates a dynamic communication contention awareness list scheduling algorithm and implemented by dynamic earliest communication finish time search algorithm which find shortest path by considering the link communication complete time and scheduling link based on insertion policy. The task scheduling priority derive form HEFT rank value [11] and his mean communication time is computed by all shortest path communication time. This paper, we first discuss a dynamic

earliest communication finish time search algorithm, then propose the serialization dynamic communication contention awareness scheduling algorithm.

The rest of the article is organized as follows: In the next section, we define the task scheduling problem and the related terminology. Section 3 proposes the dynamic communication contention-awareness scheduling algorithm. In Section 4, the simulation experimental results are presented and analysed. The article concludes with Section 5.

2 Task Scheduling Problem

2.1 Scheduling System Model

A scheduling system model consists of an application, a target computing environment, and a performance criteria for scheduling. Generally, An application is represented by a directed acyclic graph G=<*V,E*>,where *V* is the set of *v* tasks that can be executed on any of the available processors; $E \subseteq V \times V$ is the set of directed arcs or edges between the tasks representing the dependency between the tasks. For example, edge $e_{i,j} \in E$ represents the precedence constraint such that task v_i should complete its execution before task v_j completes its execution. A task may have one or more inputs. When all its inputs are available, the task is triggered to execute. After its execution, it generates its outputs. The weight w(v_i) assigned to node v_i represents its computation cost and the weight w($e_{i,j}$) assigned to edge $e_{i,j}$ represents its communication cost.Fig.1(a) shows an example DAG with the assigned node and edge weights.

A task with no parent node in the DAG is called an *entry task* and a task with no child node in the DAG is called an *exit task*. Without loss of generality, we assume that the DAG has exactly one entry task v_{entry} and one exit task v_{exit}. If multiple exit tasks or entry tasks exist, they may be connected with zero time-weight edges to a single pseudo-exit task or a single entry task that has zero time-weight.

The topology of the target system is modeled as an undirected graph G_T=<*P,L*>,where *P* is a finite set of *p* vertices and *L* is a finite set of *l* undirected edges or arcs. A vertex p_i represents the processor *i* and an undirected edge l_{ij} represents a bi-directional communication link between the incident processors p_i and p_j. A weight w(p_i) assigned to a processor p_i represents its relative execution speed and a weight w(l_{ij}) assigned to a link l_{ij} represents its relative communication speed. Fig. 1(b) shows an example of arbitrary heterogeneous system graph.

2.2 Heterogeneous System Task Scheduling Problem

As computation and communication cost is different in arbitrary network topology heterogeneous system. The average execution cost $\overline{w_i}$ of a task v_i is defined as [11].

$$\overline{w_i} = (\sum_{j=1}^{p} w(v_i) / w(p_j)) / p \tag{1}$$

The link communication speed between processors p_m and p_n is defined by all possible links' speed satisfied with $q_{m,n}$

$$q_{m,n} = \max(all\ link\ \min(\sum w(l_{ij}))) \tag{2}$$

All of these link can achieve by DFS or BFS [13]. Before scheduling, average communication costs are used to compute priority. The average communication cost $\overline{c_{i,j}}$ of an edge $e_{i,j}$ is defined by.

$$\overline{c_{i,j}} = (\sum_{m=1}^{p-1} \sum_{n=m+1}^{p} w(e_{ij}) / q_{m,n}) / (p \times (p-1)/2) \tag{3}$$

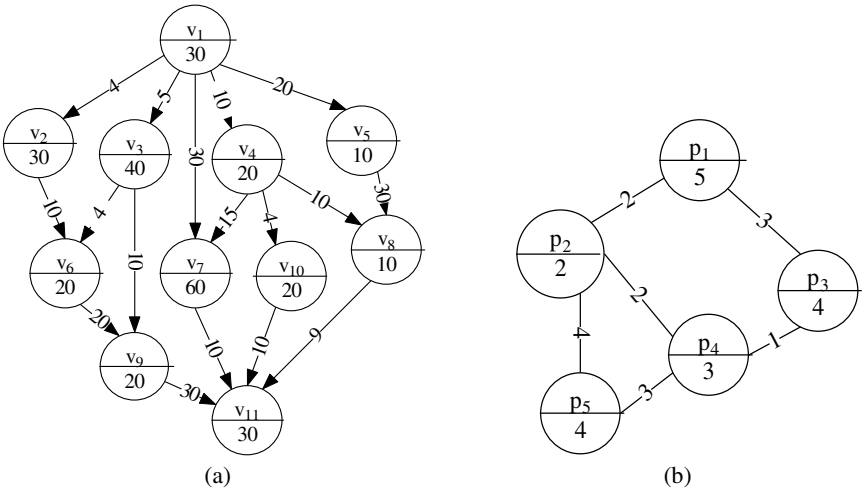


Fig. 1. (a) A parallel application task graph (b) An arbitrary heterogeneous system graph

Before presenting the objective function, it is necessary to define the EST and EFT attributes, which are derived from a given partial schedule. EST(v_i,p_j) and EFT(v_i,p_j) are the earliest execution start time and the earliest execution finish time of task v_i on processor p_j, respectively. For the entry task v_{entry}.

$$EST(v_{entry}, p_j) = 0 \tag{4}$$

For the other tasks in the graph, the EST and EFT values are computed recursively, starting from the entry task, a shown in (5) and (6), respectively. In order to compute the EFT for a task v_i, all immediate predecessor tasks of v_i must have been scheduled.

$$EST(v_i, p_m) = \max\left\{Available(v_i, p_m), \max_{v_j \in pred(v_i)} (EFT(v_j, p_n) + c_{i,j})\right\} \tag{5}$$

$$EFT(v_i, p_m) = EST(v_i, p_m) + w(v_i) / w(p_m) \tag{6}$$

Where $c_{i,j}$ is the communication cost of the edge $e_{i,j}$, which transferring data from task v_i(scheduled on p_m) to task v_j(scheduled on p_n),is computed by dynamic communication contention awareness search algorithm that will be detailed discussed in section 3.2. When both v_i and v_j are scheduled on the same processor $c_{i,j}$ becomes zero since

we assume that the intraprocessor communication cost is negligible when it is compared with the interprocessor communication cost. The pred(v_i) is the set of immediate predecessor tasks to task v_i and Available(v_i,p_m) is the earliest time at which processor p_m is ready for task execution. As using insertion-based policy [5],the time between EST(v_i,p_m) and EFT(v_i,p_m) is also available. The inner the earliest block in the EST equation returns the *ready time*, i.e, the time when all data needed by v_i has arrived at processor p_m. After all tasks in a graph are scheduled, the schedule length (i.e, overall completion time) will be the actual finish time of the exit task v_{exit}, thus the schedule length (which is also called *makespan*) is defined as.

$$makespan = EFT(v_{exit}) \tag{7}$$

The *objective function* of the task-scheduling problem is to determine the assignment of tasks of a given application to processors such that its scheduling length is minimized.

3 Dynamic Communication Contention Awareness List Scheduling Algorithm

Before introducing the details of scheduling algorithm, we introduce the graph attributes used for setting the task priorities. then discuss dynamic communication contention awareness list scheduling algorithm. At last, a parallelization algorithm is proposed.

3.1 Task Priority

Tasks are ordered in our algorithm by their scheduling priorities based on ranking [11],which computed by mean computation cost and mean communication const is defined in (1),(3).The rank of a task v_i is recursively defined by.

$$rank(v_i) = \overline{w_i} + \max_{v_j \in succ(v_i)} (\overline{c_{i,j}} + rank(v_j)) \tag{8}$$

Where succ(v_i) is the set of immediate successors of task v_i, $\overline{c_{i,j}}$ is the average communication cost of edge $e_{i,j}$, and $\overline{w_i}$ is the average computation cost of task v_i. The rank is computed recursively by traversing the task graph upward, starting from the exit task. For the exit task v_{exit}, the rank value is equal to

$$rank(v_{exit}) = \overline{w_{exit}} \tag{9}$$

Basically, rank(v_i) is the length of the critical path from task v_i to the exit task, including the average computation cost of task v_i.

3.2 Dynamic Communication Contention Awareness Link Search Algorithm

The basic underlying idea is to treat the communication edges in the same way as the nodes of the DAG: the edges are scheduled to the communication links in the same way the nodes are scheduled to the processors [6,7].

Corresponding to the node, we define LST($e_{i,j}$, $l_{m,n}$) to be $e_{i,j}$'s start time on link edge $l_{m,n}$ and LFT($e_{i,j}$, $l_{m,n}$) its finish time. Also, Available($l_{m,n}$) is defined as the available time vector of a communication edge $l_{m,n}$. While the start time of a node is constrained by the data ready time of its incoming communication, the start time of an edge is restricted by the finish time of its origin node. The scheduling of an edge differs further from that of a node, in that an edge might be scheduled on more than one link. A communication between two nodes, which are scheduled on two different but not adjacent processors, utilizes all links of the communication route between the two processors. The edge representing this communication must be scheduled on each of the involved links. Most of today's parallel system network's routing algorithm or policy is cut-through and store-forward [6,14].If all edges are available ,communication happens on all links of a route at the same time. But for arbitrary network topology heterogeneous system, to achieve all edges of a link are available is not high efficient method. This paper introduce an earliest finish communication time search algorithm based on shortest path search [13] and link scheduling with insertion policy [9,15].

Let <l_1,l_2,...l_k> be a path link $l_{m,n}$ between two processors p_m and p_n with k edges of a topology graph $\mathbf{G_T}$ that represents a communication link and the communication traverses the links in the order of the path. The earliest start time of the edge $e_{i,j}$ scheduled to the first link l_1 of route and its finish time may be described as

$$\begin{cases} LST(e_{i,j}, l_1) = \max(EFT(v_i), available(l_1)) \\ LFT(e_{i,j}, l_1) = LST(e_{i,j}, l_1) + w(e_{i,j}) / w(l_1) \end{cases} \tag{10}$$

w(l_1) is the edge l_1's communication speed, *available*(l_1) is satisfied with the time between LST($e_{i,j}$, $l_{m,n}$) and LFT($e_{i,j}$, $l_{m,n}$) is the earliest available time block on l_1.

The earliest start time and finish time of the edge $e_{i,j}$ on all subsequent link edge l_x and l_{x+1} is defined by the following equations:

$$\begin{cases} LST(e_{i,j}, l_{x+1}) = \max\{LST(e_{i,j}, l_x), Available(l_{x+1})\} \\ LFT(e_{i,j}, l_{x+1}) = \max\{LFT(e_{i,j}, l_x), LST(e_{i,j}, l_{x+1}) + w(e_{i,j}) / w(l_{x+1})\} \end{cases} \tag{11}$$

with x=1,2,...,k-1.Thus, on a subsequent link, the edge $e_{i,j}$ can only scheduled on the earliest available time block of l_{x+1}.The goal of algorithm is to find the earliest LFT($e_{i,j}$, l_k) path. The earliest finish communication time search (EFTS) algorithm is given in Algorithm 1. The time-complexity is $O(p^2)$.

Algorithm 1. EFTS($e_{i,j}$,p_m, p_n)

1. Initial all nodes of target system into set V, let the earliest finish communication time nodes set S=Φ, vector D[k] is computed by equation:

$$D[k] = \begin{cases} LFT(e_{i,j}, l_{m,k}) & \text{if } p_m \text{ connect } p_k \\ \infty & \text{other} \end{cases}$$

2. select p_x with D[x]=Min{D[k] | $p_k \in$ V-S} and let S=S$\cup$ {x}, if p_x= p_n then output the link
3. compute D[k] value use equation (11) from p_m to V-S, if LFT($e_{i,j}$, $l_{x,k}$) < D[k] then D[k] = LFT($e_{i,j}$, $l_{x,k}$)
4. repeat 2,3.

3.3 List Scheduling Algorithm

The consideration of communication contention list scheduling algorithm is outlined in algorithm 2, which is for an arbitrary topology network heterogeneous system, has two major phases: *a task prioritizing phase* for computing the priorities of all tasks and *a processor selection phase* for selecting the tasks in the order of their priorities and scheduling each selected task on its "best" processor, which minimizes the task's finish time.

Task Prioritizing Phase. This phase requires the priority of each task to be set with the rank value, which is based on mean computation and mean communication costs. The task list is generated by sorting the tasks by decreasing order of rank. Tie-breaking is done randomly. It can be easily shown that the decreasing order of rank values provides a topological order of tasks, which is a linear order that preserve the precedence constraints.

Processor Selection Phase. As the algorithm adopt insertion-based policy which considers the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on a processor. Thus, a task v_i after v_j scheduled may not be executed after v_j. In computing communication delay time phase, we also use insertion-based policy which means a edge $e_{i,j}$ is scheduled on a communication edge may transfer before others already-scheduled on it. Additionally, scheduling on this idle time slot should preserve precedence constraints. In algorithm 2,this goal is achieved by use Available(v_i,p_m) and Available(l_x).Like general list scheduling algorithm, this algorithm search processor p_n which has the earliest finish time and assign task v_i to process p_n. The novel characteristic is communication cost which based on dynamic communication contention awareness computing algorithm.

Algorithm 2. List Scheduling

1. Set the computation costs of tasks and communication costs of edges with mean values use equation (1),(3) respectively
2. Compute rank value for all tasks by traversing application graph, starting from the exit task.
3. Sort the tasks in a scheduling list by non-increasing order of rank value.
4. **while** there are unscheduled tasks in the list **do**
5. Select the first task, v_i, from the list for scheduling
6. **for** each processor p_k in the processor set ($p_k \in$ P) **do**
7. **for** each processor p_x which execute task v_j and $v_j \in$ pred(v_i) **do**
8. **if** $p_x = p_k$ **then** communication time is zero
else Use Algorithm 1. compute communication time
9. Search the most data transfer delay time
10. Compute the earliest finish time on p_k use equation (5),(6).
11. Assign task v_i to process p_n with minimize EFT(v_i,p_n).
12. **endwhile**.

4 Simulation Experiment Results

In this section, we use simulation approach to evaluate the performance of the proposed dynamic communication contention-aware scheduling. which we call dynamic

algorithm. The objective is to investigate the schedule length compared to static algorithm which proposed by Oliver Sinnen and Leonel [6] and classic algorithm such as HEFT [11], DLS [8]. In this paper, we consider randomly generated application graphs for testing the algorithms. The comparison is intended not only to present quantitative results but also to qualitatively analyse the results and to suggest explanations, for a better insight in the overall scheduling problem.

4.1 Randomly Generated Application Graphs

For the generation of random graphs, which are commonly used to compare scheduling algorithms [7,9,11], two fundamental characteristics of the DAG are considered: (i) the communication to computation ratio (CCR) and (ii) the average number of edges per node.The CCR is defined as the sum of all communication costs (that is the weight of all edges) divided by the sum of all computation costs: CCRs of 0.1, 1 and 10 are used to simulate low, medium and high communication, respectively. For the average number of edges per node, we utilised values of 1 and 5.Graphs were generated for all combinations of the two above parameters with the number of nodes ranging between 100 and 1000, in steps of 100. Every possible edge(DAGs are acyclic) was created with the same probability, calculated based on the average number of edges per node. To obtain the desired CCR for a graph, node weights are taken randomly from a uniform distribution [0.1;1.9] around 1, thus the average node weight is 1. Edge weights are also taken from a uniform distribution, whose mean depends on the CCR and on the average number of edge weights (e.g. the mean is 5 for CCR=10 and 2 edges per node). The relative deviation of the edge weights is identical to that of the node weights. Every set of the above parameters was used to generate several random graphs in order to avoid scattering effects. The results presented below are the average of the results obtained for these graphs.

4.2 The Target System

To investigate arbitrary network topology (i.e., processor connectivity), we used four different topologies in the experiments:(Ⅰ)8-processor ring, (Ⅱ) 16-processor bus topology which not only represent a SMP system, but also topologies of cluster computing, e.g. workstations connected via a hub (or a switch), (Ⅲ) 8-processor fully connected network, and (Ⅳ)16-processor randomly structured topology. The random topology was generated such that the degree of each processor ranged from two to five. The four topologies are connected by 5 edges with different communication speed random from 1 to 4 and randomly connected among them.

4.3 Randomly Application Performance Results

The performance of the algorithms were compared with respect to various graph characteristics. The first set of experiments compares the performance and cost of the algorithms with respect to various graph sizes (see Fig.2).As CCR=0.1,the communication cost is less than the computation cost, the task is relative independent. Our dynamic algorithm is not significantly better than HEFT, because its scheduling priority is derivate from HEFT and communication contention is few. But the static algorithm is worse

than DLS,HEFT, Dynamic, for his schedule priority based on less efficient bottom_level, which has very good performance in fully connected homogeneous system and not suitable for arbitrary network topology heterogeneous system. As CCR=1,which computation cost equal to communication cost, our dynamic algorithm has much improvement than HEFT,DLS and static algorithm. Because there exists some communication contention, but not very seriously. On the other hand, static algorithm also better than DLS, this improvement contribute to its communication contention awareness method. When CCR=10,the communication cost more than computation cost, the communication contention become very seriously. The dynamic algorithm shows significantly improvement than static about 7.5%,HEFT 18.8%,DLS 24.7%.

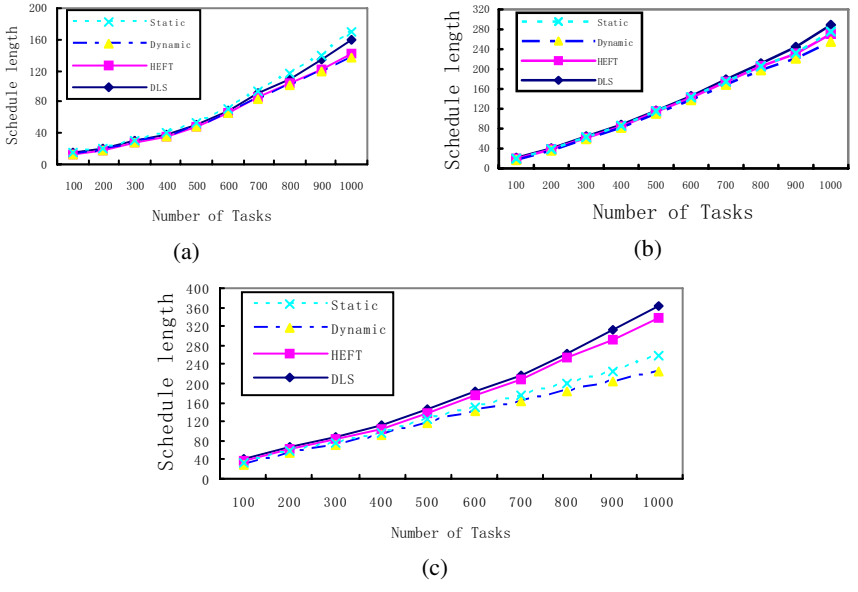


Fig. 2. Experiments in the first solution target system (a) CCR=0.1 (b) CCR=1 (c) CCR=10

5 Conclusions and Future Work

This paper was dedicated to the incorporation of communication contention awareness into task scheduling. The ideal system model of classic task scheduling does not capture communication contention for communication resources. Therefore, an arbitrary network topology heterogeneous system model was proposed that abandons the assumptions of a fully connected network and concurrent communication. Communication Contention awareness is achieved by scheduling the edges of the DAG onto the links of the target system which using dynamic communication earliest finish time search algorithm. To improve list scheduling accuracy and efficiency, list scheduling priorities are achieved by recursive rank computation on heterogeneous arbitrary architectures. Simulation results demonstrated the significantly improved accuracy and efficiency of the schedules.

This work represents our first and preliminary attempt to study a very complicated problem. Future studies in this area are twofold. First, the involvement of the processors in communication should be investigated in order to further improve task scheduling's accuracy and efficiency. Second we plan to study a more accurately computation of list scheduling priority method.

References

1. G.C. Sih, E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous machine architectures. *IEEE Trans. Parallel Distrib. Systems* , 4 (2) (1993) 175–187
2. M.R. Gary, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, CA, 1979.
3. I. Ahmad, Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Systems* 9 (9) (1998) 872–892.
4. M.K. Dhodhi, I. Ahmad, A. Yatama, I. Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing system. *J. Parallel Distrib. Comput.* 62 (9) (2002) 1338–1361
5. D. Kim, B.G. Yi. A two-pass scheduling algorithm for parallel programs. *Parallel Comput.* 20 (6) (1994) 869–885
6. Sinnen O., Sousa L.A. Communication contention in task scheduling. *IEEE Trans. Parallel Distrib. Systems*, Vol. 16, No. 6, June 2005 Page(s):503 - 515
7. Sinnen Oliver, Sousa, Leonel. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing,* Volume:30, Issue:1, January,2004, pp.81-101
8. H. El-Rewini, T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.* 9 (2) (1990) 138–153
9. Liu,G.Q, Poh,K.L, Xie,M. Iterative list scheduling for heterogeneous computing. *Journal of Parallel and Distributed Computing,* Volume: 65, Issue:5, May,2005,pp.654-665
10. 10.M. Iverson, F. Ozuner, G. Follen. Parallelizing existing applications in a distributed heterogeneous environment. in: *Proceedings of Heterogeneous Computing Workshop*, 1995, pp. 93–100
11. H. Topcuoglu, S. Hariri, M.-Y. Wu. Performance-effective and low complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Systems* 13 (3) (2002) 260–274
12. B.S. Macey, A.Y. Zomaya, A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. in: *Parallel Processing Symposium*, 1998, Proceedings of IPPS/SPDP 1998, 1998, pp. 538 –541
13. T.H. Cormen, C.E. Leiserson, R.L. Rivest. Introduction to Algorithms. MIT Press, 1990.
14. D.E. Culler and J.P. Singh. Parallel Computer Architecture. Morgan Kaufmann Publishers, 1999.
15. O. Sinnen, L. Sousa. Exploiting unused time slots in list-scheduling considering communication contention. in: *Euro-Par 2001 Parallel Processing, Lecture Notes in Computer Science*, vol. 2150, Springer-Verlag, 2001, pp. 166–170.

Distributed Provision and Management of Security Services in Globus Toolkit 4

Félix J. García Clemente, Gregorio Martínez Pérez, Andrés Muñoz Ortega, Juan A. Botía Blaya, and Antonio F. Gómez Skarmeta

Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia, Spain
{fgarcia, gregorio, skarmeta}@dif.um.es, {amunoz, juanbot}@um.es

Abstract. Globus Toolkit version 4 (GT4) provides a set of new services and tools, completing a first step towards the migration to web services of previous Globus Toolkit versions. It provides components addressing basic issues related to security, resource discovery, data movement and management, etc. However, it still lacks from advanced management frameworks that may be linked to GT4 services and thus providing a system-wide view of these services. This is especially true in the case of security related services. They need to be properly managed all across different organizations collaborating in the grid infrastructure. This paper presents the major achievements when designing and developing a semantic-aware management framework for GT4 security services.

Keywords: GT4 security services management, web service-oriented management, semantic-aware policy checking and validation.

1 Introduction and Motivation

Globus Toolkit version 4 (GT4) provides a range of new services and features, including a first robust implementation of Globus Web Services components, allowing GT4 to complete the first stage of migration to web services that began with GT3.

The management of grid services is a complex task, with a high number of issues to be considered. Really useful grid infrastructures should offer a number of tools for managing such services. Their purpose should be that of facilitating the management tasks of grid users and administrators. They may also provide some added-value services as those related with checking the validity of grid management rules, performing proactive monitoring, etc. thus enabling automatic and dynamic management frameworks. Even though these features are of real need for grid infrastructures, the Globus toolkit does not provide any particular solution to distributed grid management [7]. However, several active working groups and projects are starting to address some of these issues. For example, some important solutions are provided by KAoS and PERMIS. KAoS [13] is a set of platform-independent services, allowing fine-grained policy-based management of grid services on the Globus platform. By providing an interface between the Globus grid

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1325 – 1335, 2006.
© Springer-Verlag Berlin Heidelberg 2006

and KAoS, it enables the use of KAoS mechanisms to manage GSI (Grid Security Infrastructure)-enabled grid services. Other solution is PERMIS [11] that is a Privilege Management software that uses the principles of Role-Based Access Control (RBAC) to deploy an authorisation service into a Globus container.

In the same line, the POSITIF (Policy-based Security Tools and Framework) [12] EU IST project proposes the design and implementation (as contribution to the open-source community) of a framework for managing security policies for the grid scenario based on GT4. Moreover, POSITIF framework meets certain requirements that make it different to other existing solutions. These requirements are the integration of web-services in the whole management cycle (from the definition task to the enforcement and monitoring processes) and the use of semantic-aware management languages enabling different added-value features, such as detection and resolution of conflicts existing between different grid management policies specified on the basis of decision rules.

This paper is structured as follows. First, the section 2 presents the main ideas behind the specification of semantically-rich policies in the POSITIF framework and how policies are represented. Then, we present how POSITIF framework has been linked with GT4 in section 3, along with the way security policies have been defined and the tools developed to evaluate this proposal in a real GT4 scenario. Finally, we conclude this paper with our remarks and statements of direction in section 4.

2 Representation of Semantic-Aware Security Policies

This section provides an overview of the main ideas behind the specification of semantically-rich policies in the POSITIF framework. Additional details can be found at [3] and [12].

- *Use of a standard information model.* The use of information models eases the task of building efficient and well-integrated security management systems. In this sense, the POSITIF representation is based on a well-recognized standard such as CIM (Common Information Model) [1], thus enabling the management of security services in a uniform manner. The CIM model is independent of any encoding or particular specification. However, for an information model to be useful, it has to be mapped into some implementation. In this sense, CIM can be mapped to multiple structured specifications. As part of our semantic-aware security policy model we have proposed the specification of CIM using OWL [14]. In this sense, authors presented a proposal for expressing CIM objects in OWL, named CIM-OWL, in [4].
- *Policies like horn-like rules.* Policies will be defined by managers as *if-then* rules. In consequence, authors have proposed the use of Semantic Web Rule Language (SWRL) [15] which extends the set of OWL axioms to include a high-level abstract syntax for Horn-like rules that can be combined with an OWL knowledge base. A useful restriction in the form of the rules is to limit antecedent and consequent atoms to be named classes, where the classes are defined purely in OWL. Adhering to this format makes it easier to translate rules to or from future or existing rule systems, including Prolog and Jena [8]. In fact, both the rule-based inference engine from the Jena framework and the Pellet reasoner have

been the ones used as part of our deployment in POSITIF. More information on this proposal from authors can be found at [4, 9].

- *Separation of system description and policy description*. POSITIF separates the formal definition of the desired level of security and the target information system. In this sense, two different languages have been formally defined, named SDL (System Description Language) and SPL (Security Policy Language). SDL describes networked systems and applications conveying the topology of the system, the functionality of each element, and the security capabilities of each component of the system using an ontology based on the CIM information model, while SPL offers to security policy administrators the possibility to specify both high-level policies and low-level policies using horn-like SWRL rules.
- *Reasoning capabilities.* POSITIF incorporates semantic expressiveness into the management information specifications to ease the tasks of validating and reasoning about the policies which definitely help in handling the security management tasks (e.g., conflict resolution). POSITIF management representations are realized in the form of an ontology using OWL that eases to perform useful reasoning tasks. This is due to the fact that OWL is based on description logics. This simple and yet powerful kind of first order-like logic allows to perform reasoning tasks not only on individuals but also on the structure of the base information model which holds the instances, with efficient and sound algorithms. More information on the proposal from authors can be found at [9].
- *Integration and interoperability with other frameworks.* The use of standards (such as the CIM information model or Web Services over HTTP/HTTPS) eases the integration of the POSITIF framework with other existing management frameworks. This can be of real interest in order to integrate the results presented in this paper with any other approaches coming from any of the different GGF working groups or any other research works as those mentioned before.

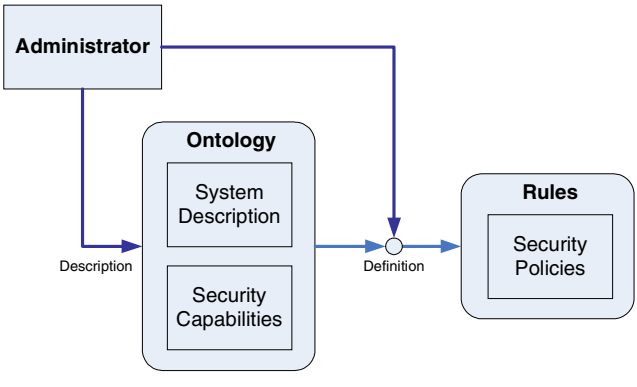


Fig. 1. Specification of security policies

Figure 1 shows our approach for the specification of security policies. The administrator describes the system and its security capabilities using an ontology based on the CIM information model. Then, the administrator uses these ontology

elements within the body and the head of horn-like rules that really represent the security policies governing the behaviour of the system being managed.

3 Security Policy Management for GT4 Services

Globus Toolkit 4 provides effective resource management for the grid-computing environment. It also includes security services for providing secure access to these resources, and then it needs security policy services to determine when the resources can be accessed (or not) from any other entity taking part of the grid infrastructure. In this sense, POSITIF represents a good complement to the GT4 system, providing a wide range of security policy management capabilities that rely on platform-specific enforcement mechanisms, and thus helping grid managers to define the right policies that should be applied and enforced in the grid components.

By providing an interface between GT4-based grids and the POSITIF framework, we enable the use of POSITIF mechanisms to manage Grid Security Infrastructure (GSI) [6] enabled grid services. GSI (also called GT Security) is the basis for GT4 security layer and it is composed of a set of command-line tools to manage certificates, and a set of Java classes to easily integrate security into web services. GSI offers programmers features such as transport-level and message-level security, authentication through X.509 digital certificates, several authorization schemes, credential delegation and single sign-on, and different levels of security (i.e., container, service, and resource).

GSI is the only component of the GT4 that we use in the integration. The interface is a POSITIF plug-in, which we called *GT4 Security Plug-in*. The plug-in itself is a grid plug-in that gives grid clients and services the ability to check the action to perform on the basis of current policies. Figure 2 shows the basic architecture.

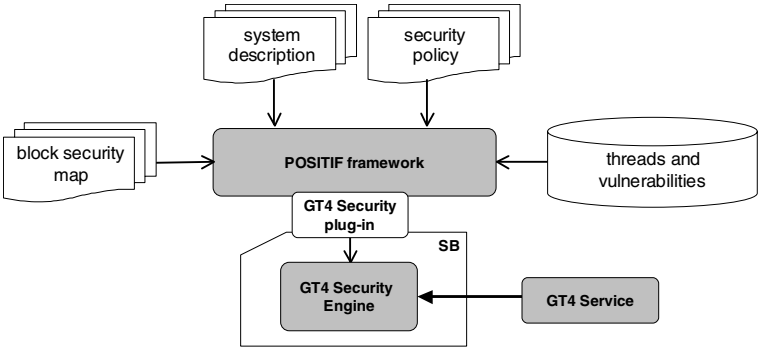


Fig. 2. GT4 security plug-in

This plug-in links grid services to the POSITIF framework. Currently GT4 Security plug-in has been deployed as an authorization interceptor because GT4 makes it easier this way. In the future, when GT4 supports new interceptors for other types of policies (e.g., privacy) the plug-in will be adding this new functionality easily thanks to the extensibility of the POSITIF framework.

3.1 Enforcing Area

GT Security includes an Authorization Framework that allows for a variety of authorization schemes, including a grid-mapfile access control list, an access control list defined by a service, a custom authorization handler, and access to an authorization service via the SAML protocol. *GT4 Security plug-in* uses a custom authorization handler.

GT4 Security plug-in is a middleware authorization package that enables any service with POSITIF authorization logic by developing interceptors and enabling them in a security descriptor. What this means is that any request for an authorization decision made by a Globus service is sent to an independent package - i.e. POSITIF framework via *GT4 Security plug-in*. POSITIF framework is responsible for managing a policy associated with a GT4 service.

GT4 authorization configuration involves setting a chain of authorization schemes (also known as Policy Decision Points (PDPs)). When an authorization decision needs to be made the PDPs in the chain are evaluated in turn to arrive to a permit or deny decision. The combining rule for results from the individual PDPs is currently "deny overrides"; that is, all PDPs have to evaluate to permit for the chain to finally evaluate as permit.

GT4 Security plug-in includes two interceptors: POSITIFAuthzPDP and POSITIFAuthzPIP (see Figure 3). POSITIFAuthzPDP creates the methods to implement the GT4 PDP interface [2] that must be implemented by all PDPs in the interceptor chain. A PDP is responsible for making decisions on whether a subject is allowed to invoke a certain operation or not. The subject may contain public or private credentials holding attributes collected and verified by the POSITIFAuthzPIP.

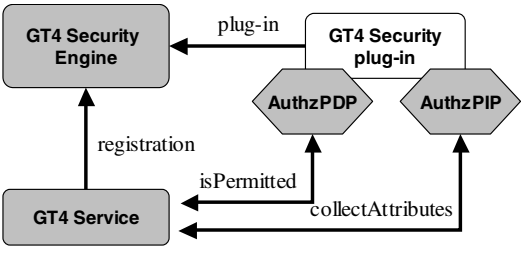


Fig. 3. GT4 security plug-in implements authorization PDP and PIP

The following example shows a security descriptor with the correct authorization chain configured. In the next example, the values chosen for the scope are pdpscope and pipscope:

```
<authz value="pipscope: org.globus.positif.authorization.POSITIFAuthzPIP
pdpscope: org.globus.positif.authorization.POSITIFAuthzPDP"/>
```

It is necessary to configure the interceptors with the URL of the web service to connect to the POSITIF enforcement area. We assume that the PDPConfig object picks up the configuration information from the service deployment descriptor. This is an example.

```
<parameter name="pipscope-authzService"
        value="https://umu.positif.org:8443/wsrf/services/POSITIFAuthorizationService"/>
```

3.2 Expressing Semantic-Aware Policies

The basic components of any authorization policy are the subjects, actions, targets and the context. A sample policy would read: It is permitted for subject(s) S to perform action(s) A in the target(s) T in the context C. CIM ontology defines the classes depicted in Figure 4 to represent the management concepts that are related to an authorization policy. Privilege is the base class for all types of activities, which are granted or denied to a subject by a target. Authorized-Privilege is the specific subclass for the authorization activity.

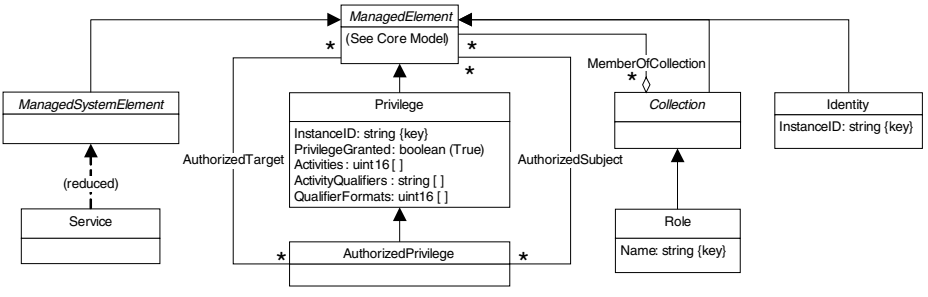


Fig. 4. UML diagram of CIM User-Authorization classes

Whether an individual Privilege is granted or denied is defined using the PrivilegeGranted boolean. The association of subjects to AuhorizedPrivileges is accomplished explicitly via the association AuthorizedSubject. The entities that are protected (targets) can be similarly defined via the association AuthorizedTarget. The Role object class is used to represent a position or set of responsibilities within an organization, organizational unit or system administration scope. It is filled by a person or persons represented by Identity class or non-human entities represented by ManagedSystemElement subclasses that may be explicitly or implicitly members of this collection subclass.

The Service class is derived from ManagedSystemElement that provides the representation of a service. Since both Service and Role are specializations of MagamentElement, they may be either the target or the subject of Privilege. For our purposes, we use Role and Identity as subject and Service as target.

In this context, the policy administrator may decide basic authorization policies, for example: “The service counter permits to query to the students”, or more complex ones, for example “If the service counter permits to query to the students, then the service counter permits to query to the teachers”. Figure 5 shows the logic representation of this policy that can be mapped easily to SWRL.

An authorization policy must be applied in an administrative domain described by the SDL language aforementioned where the main grid services will be described. In this particular example, the administrative domain is composed by a role that represent the set of students, a role that represent the set of teachers, and a privilege to query the counter. Figure 6 shows the description of the administrative domain.

```
CIM_AuthorizedTarget(authtarget?) ∧ CIM_AuthorizedSubject(authsubject1?) ∧
CIM_Service(counter?) ∧ CIM_AuthorizedPrivilege(privilege?) ∧
Name(counter,"Counter") ∧ CATPrivilege(authtarget,privilege) ∧
TargetElement(authtarget,counter) ∧ CASPrivilege(authsubject1,privilege) ∧
PrivilegedElement(authsubject1,#Students)
→
CIM_AuthorizedSubject(authsubject2) ∧ CASPrivilege(authsubject2,privilege) ∧
PrivilegedElement(authsubject2,#Teachers)
```

Fig. 5. Example of a security policy

```
<rdf:RDF>
        <CIM_Service rdf:about="#counter">
                <Name>Counter</Name>
        </CIM_Service>
        <CIM_AuthorizedTarget rdf:about="#authtarget1">
                <CATPrivilege rdf:resource="#authzquery" />
                <TargetElement rdf:resource="#counter" />
        </CIM_AuthorizedTarget>
        <CIM_AuthorizedPrivilege rdf:about="#authzquery">
                <InstanceID>positif:authzquery</InstanceID>
                <PrivilegeGranted>true</PrivilegeGranted>
                <Activities>Query</Activities>
        </CIM_AuthorizedPrivilege>
        <CIM_AuthorizedSubject rdf:about="#authsubject1">
                <CASPrivilege rdf:resource="#authzquery" />
                <PrivilegedElement rdf:resource="#Students" />
        </CIM_AuthorizedSubject>
        <CIM_Role rdf:about="#Students">
                <Name>Students</Name>
        </CIM_Role>
        <CIM_Role rdf:about="#Teachers">
                <Name>Teachers</Name>
        </CIM_Role>
</rdf:RDF>
```

Fig. 6. Representation in OWL of the administrative domain

3.3 Checking and Transforming Area

The SWRL-encoded SPL rule (representing the policies governing the behaviour of the grid services) together with the OWL-encoded administrative domain (representing the set of services and related information –e.g., roles– being managed) must be loaded into a rule reasoner in order to make things work. The reasoner will allow us to check the validity of the policy specification by testing it with the reference ontology of the CIM model and its instances. One rule reasoner we can be used for this purpose is the Jena reasoner as authors have been doing during this research. Figure 7 shows the Jena representation of this policy in our grid scenario.

```
#exampleRule:
( ?authtarget  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
          http://www.positif.com/cim#CIM_AuthorizedTarget )
( ?authsubject1  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
          http://www.positif.com/cim#CIM_AuthorizedSubject )
( ?counter  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
          http://www.positif.com/cim#CIM_Service )
( ?privilege  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
          http://www.positif.com/cim#CIM_AuthorizedPrivilege )
( ?counter http://www.positif.com/cim#Name 'Counter' )
( ?authtarget http://www.positif.com/cim#CATPrivilege ?privilege )
( ?authtarget http://www.positif.com/cim#TargetElement ?printer )
( ?authsubject1 http://www.positif.com/cim#CASPrivilege ?privilege )
( ?authsubject1 http://www.positif.com/cim#PrivilegedElement
          http://www.positif.com/cim#Students )
->
( ?authsubject2  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
          http://www.positif.com/cim#CIM_AuthorizedSubject )
( ?authsubject2 http://www.positif.com/cim#CASPrivilege ?privilege )
( ?authsubject2 http://www.positif.com/cim#PrivilegedElement
          http://www.positif.com/cim#Teachers )
```

Fig. 7. Jena representation of an authorization policy

The rule reasoner performs inference about SWRL rules deriving new individuals (i.e., new instances of CIM class or associations). For our example, the rule reasoner infers new data that grants the privilege to query the counter grid service to the teachers. Another important functionality of the rule-based reasoner is that it eases the detection and resolution of conflicts. A conflict occurs when two policy rules cannot be met simultaneously, e.g., one allows the user to start the service and another prohibits the same user from starting this service.

3.4 Management Area

POSITIF framework offers a guided, simple and graphic tool called ORE (Ontology Rule Editor) [10] in the management area. This tool uses grid-related SPL and SDL semantic languages described before. ORE offers several functionalities:

- *Loads system description*, either through OWL files or with the URI that identifies the ontology. Once the ontology is loaded, ORE shows diverse information about it (version, imported ontologies, etc.) and generates a hierarchical tree, which represents the classes and instances contained in the ontology model.
- *Edits policy rules* in a guided and easy way thanks to the Wizard that ORE incorporates with. Both left-hand-side (LHS) and right-hand-side (RHS) tuple-elements of any rule are defined in the same way, the Wizard guiding the user across three steps, choosing the subject, predicate and object of the LHS or RHS tuple-element. The Wizard makes semantical checking over the rules created, and warns the user if they are not correct.

- *Check the defined rules.* Using the security checker that incorporates a Jena reasoner. ORE sends the rules to the checker, the checker evaluates them, and the results are returned to ORE and shown to the administrator. In this case, if the checker is allowed to, the facts or actions inferred could be asserted or performed.

Figure 8 shows a snapshot of the ORE editor. It is basically compound by a navigator through the ontology used (in the left bottom part of this figure) and two lists (in the upper part of the same window) which reflect the antecedent and consequent elements added, in the form of RDF triplets. As such, three phases are used to create each triplet: one for the subject, one for the property attached to the subject and one for the object.

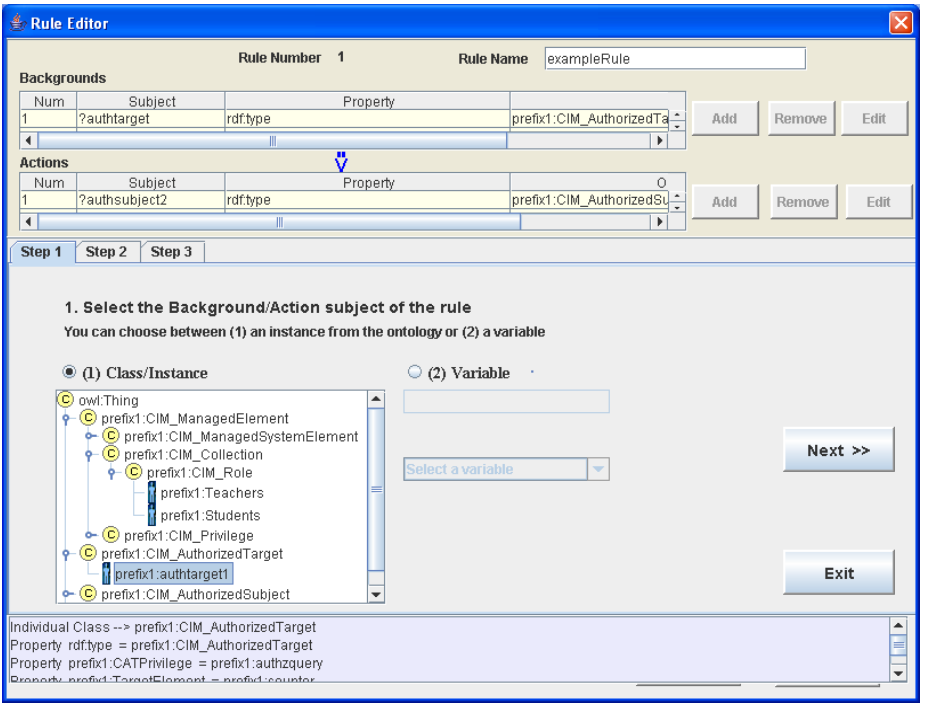


Fig. 8. ORE editor – loading system description

In our current prototype, GT4 policies are added to POSITIF through ORE editor. When a client requests a GT4 service, the POSITIF plug-in will check if the requested action is authorized on the basis of current policies by querying the POSITIF framework. This query is based on the reasoning capabilities offered as part of SPL.

4 Conclusions and Future Work

One of the main limitations in current GT4 infrastructures is the lack of frameworks enabling dynamic and automatic management of resources and services offered across a virtual organization (VO).

This paper has presented a semantic-aware framework enabling the dynamic management of security services in GT4 infrastructures. The defined framework also represents one step towards the automatic management of security services, considering not only authorization services, and providing also additional reasoning mechanisms to deal with issues such as detection and resolution of conflicts existing between different grid management rules.

As statement of direction, this research is now working to incorporate part of the reasoning mechanisms described in this paper to provide dynamic conflict resolution as part of the proactive monitoring components existing in the framework. Some other components of the framework (as the security module area) are being designed and implemented to provide security alarms in grid infrastructures.

Acknowledgments

This work has been partially funded by EU POSITIF (Policy-based Security Tools and Framework, IST-002-002314) IST project.

References

1. Common Information Model (CIM) Standards, DMTF, http://www.dmtf.org/standards/cim (2006)
2. Freeman, T., Ananthakrishnan, R.: Authorization processing for Globus Toolkit Java Web services, developerWorks, IBM (2005)
3. García Clemente, F.J., Martínez Pérez, G., Botía Blaya, J.A., Gómez Skarmeta A.F.: Description of Policies Enriched by Semantics for Security Management. Book Chapter, Book on Web Semantics and Ontology, Idea Group Inc. (2006)
4. García Clemente, F. J., Martinez Pérez, G., Botía Blaya, J. A., Gomez Skarmeta, A. F.: On the Application of the Semantic Web Rule Language in the Definition of Policies for System Security Management. Workshop on Agents, Web Services and Ontologies Merging (AWeSOMe), OnTheMove Workshops, Larnaca, Cyprus (2005)
5. Globus Toolkit Official Documentation, http://www.globus.org/toolkit/docs/ (2006)
6. GT Security (GSI), http://www.globus.org/toolkit/security/ (2006)
7. Harmer, T., Stell, A., McBride, D.: UK Engineering Task Force Globus Toolkit Version 4 Middleware Evaluation. Version 1.0. http://www.nesc.ac.uk/technical_papers/UKeS-2005-03.pdf (2005)
8. Jena – A Semantic Web Framework for Java, http://jena.sourceforge.net/ (2006)
9. Martínez Perez, G., García Clemente, F.J., Botía Blaya, J.A., Gómez Skarmeta, A.F.: Enabling Conflict Detection using Ontology and Rule-Based Reasoning in the Specification of Security Policies, 13th Workshop of HP OpenView University Association, Côte d'Azur, France (2006)
10. ORE (Ontology Rule Editor), http://sourceforge.net/projects/ore (2006)
11. PERMIS – Privilege and Role Management Infrastructure Standards Validation, http://sec.isi.salford.ac.uk/permis/ (2006)
12. POSITIF (Policy-based Security Tools and Framework) EU IST Project, http://www.positif.org/ (2006)

13. Uszok, A., Bradshaw, J., Jeffers, R.: KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services. In Proceedings of the Second International Conference on Trust Management, Springer-Verlag (2004)
14. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. W3C Recommendation. W3C (2004)
15. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, The Rule Markup Initiative, http://www.ruleml.org/swrl/ (2006)

A Fine-Grained and X.509-Based Access Control System for Globus*

Hristo Koshutanski[1], Fabio Martinelli[2], Paolo Mori[2],
Luca Borz[1], and Anna Vaccarelli[2]

[1] CREATE-NET
Via Solteri 38, Trento 38100, Italy
{hristo.koshutanski, luca.borz}@create-net.org
[2] Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Via Moruzzi 1, Pisa 56124, Italy
{fabio.martinelli, paolo.mori, anna.vaccarelli}@iit.cnr.it

Abstract. The rapid advancement of technologies such as Grid computing, peer-to-peer networking, Web Services to name a few, offer for companies and organizations an open and decentralized environment for dynamic resource sharing and integration. Globus toolkit emerged as the main resource sharing tool used in the Grid community.

Access control and access rights management become one of the main bottleneck when using Globus because in such an environment there are potentially unbounded number of users and resource providers without a priori established trust relationships. Thus, Grid computational resources could be executed by unknown applications running on behalf of distrusted users and therefore the integrity of those resources must be guaranteed.

To address this problem, the paper proposes an access control system that enhances the Globus toolkit with a number of features: (i) fine-grained behavioral control; (ii) application-level management of user's credentials for access control; (iii) full-fledged integration with X.509 certificate standard; (iv) access control feedback when users do not have enough permissions.

1 Introduction

Grid computing is a technology to support resource sharing among a very large dynamic and geographically dispersed set of entities. Entities participating in a Grid environment are organized in Virtual Organizations (VOs) and could be companies, universities, research institutes and other organizations. Grid resources could be computational resources, storage resources, software repositories and so on. Each entity exploits a Grid toolkit to set up a Grid service, i.e.

* This work is partially funded under the IST program of the EU Commission by the 2003-S116-00018 PAT-MOSTRO project, the STREP-project "ONE" (INFSO-IST-034744), the NoE-project "OPAALS" (INFSO-IST-034824), the STREP-project "S3MS" and the STREP-project "GRIDTrust".

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1336–1350, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the environment to share its resource on the Grid. Nowadays the most used Grid toolkit is Globus [1,2].

Security, in particular access control issue, is one of the main concerns in Grid mainly because users participating in such an environment are potentially unknown with almost no a priori established trust relationships, which belong to distinct administrative domains. Computational resources and other Grid services are usually shared and used by applications running on behalf of unknown (not trusted) Grid users.

To improve the security of computational services, this paper describes the implementation of an enhanced access control system for the Globus toolkit that results from the integration of two security tools: Gmon and *i*Access. The key features of the new system are the enhanced fine-grained monitoring control on applications' behavior, policy-based access control decisions, integration with X.509 digital certificate standard for users' access rights management and access control feedback on application execution failure.

The adoption of X.509 standard [3] allows for cross-domain certification and identification. The X.509 framework is widely used standard nowadays by many vendors, companies and service providers.

The access control feedback is when the Grid user is denied to execute an application because of not enough access rights. The feedback allows the user to submit a proper set of credentials next time when he exploits the same Grid service.

The paper outline is the following. Section 2 provides a description of the main access control approaches used with the Globus toolkit. Section 3 shows the system underlying logical model. Next, Section 4 presents the the system architecture and its functional description. Section 5 emphasizes on the implementation details of the components that compose the system. Section 6 shows our preliminary system evaluation and Section 7 concludes the paper.

2 Related Work

As mentioned, security is a challenging issue in Grid because of the dynamic, collaborative and distributed nature of the environment. Particularly, establishing trust relationship among partners in Grid, as well as, controlling access to Grid's resources has emerged as a key security requirement. If an adequate security support is not adopted applications could perform dangerous and malicious actions on Grid resources. Since Globus [4] is a widely used Grid toolkit, most of the approaches that have been proposed to improve the Grid security are related to it.

Community Authorization Service (CAS) [5,6] has been proposed by the Globus team in order to improve the Globus authorization system. CAS is a service that stores a database of VO policies, which determine what each Grid user is allowed to do on the VO resources as a VO member. This service issues user's proxy certificates that embed CAS policy assertions. The Grid user that wants to use a Grid resource contacts the CAS service to obtain a proper credential in order to execute an action on the resource. The credentials (certificates)

returned by a CAS server are to be presented to the local Grid manager hosting the desired service. This approach requires CAS-enabled services, i.e. services that are able to understand and enforce the policies included in the credentials released by a CAS server.

A more advanced effort for enforcing and administrating authorization policies across heterogeneous systems is the OASIS eXtensible Access Control Markup Language (XACML) framework [7]. The main actor here is the Policy Decision Point (PDP) responsible for retrieving the relevant policies with respect to the clients request, evaluating them and rendering an authorization decision. The work also considers the combination of different policies from various partners using some policy combining algorithms and taking an authorization decision on the base of evaluating them. As such XACML is a good candidate for an XML-based representation of access policies across Grid domains.

The other key approach of OASIS consortium is the Secure Assertion Markup Language (SAML) standard [8]. The main objective of the approach is to offer a standard way for exchanging authentication and authorization information between trust domains. The basic data objects of SAML are assertions. Assertions contain information that determines whether users can be authenticated or authorized to use resources. The SAML framework also defines a protocol for requesting assertions and responding to them, which makes it suitable when modeling interactive communications between entities in a distributed environment such as Grid.

Keahey and Welch [9,10] propose an approach that integrates an existing authorization system, called Akenti [11], in the Grid environment. Similarly, Stell, Sinnot and Watt [12] integrate a role based access control infrastructure, called PERMIS [13], with the Globus toolkit to provide fine-grained access control.

The main functionality behind PERMIS and Akenti is that the information needed for an access decision, such as identity, authorization and attributes is stored and conveyed in certificates, which are widely dispersed over the Internet (e.g., LDAP directories, Web servers etc.). The authorization engine has to gather and verify all user's related certificates and to evaluate them against the access policy in order to take a decision.

One of the advantages of PERMIS infrastructure with respect to other Grid access control systems is its integration with X.509 certificate framework. X.509 identity and attribute certificates are used to attest and convey users' identities and attributes. When PERMIS access decision function (ADF) gathers all user's certificates it checks them against time validity period and trusted certificate authorities. Once ADF validates and verifies the certificates it takes an access decision according to the local (resource specific) access policy and the user's access rights. The access control policy is also loaded and extracted from X.509 attribute certificate.

Since X.509 [3] is de facto the main and widely used digital certificate standard and since it was also adopted within the Globus toolkit so it is necessary for an access control system to be fully integrated with the standard.

The access control system proposed in this paper has the following characteristics over the existing approaches:

- Fine-grained application monitoring – applications executed on the computational service are not considered as atomic entities but all the interactions with local resources are monitored according to a behavioral policy;
- History-based application monitoring – permissions to execute actions depend also from the actions that have been previously executed by the application, i.e. the application behavior is taken into account;
- X.509 access rights management – permissions to execute actions depend also from the credentials that a Grid user has submitted to Globus. Full-fledged integration with X.509 Identity and Attribute Certificates;
- Fine-grained user's access rights management – users can control the scope of their credentials. We provide general or application specific granularity of credential management;
- Access control feedback – digitally signed SAML assertion is returned back to the user in case he needs more access rights.

3 The System Underlying Model

Each Grid service provider protects its computational resources by means of logical policies. We have defined three logical policies: Behavioral Policy, Access Policy and Disclosure Policy. These policies and the logical models for enforcing them are described in more details in [14]. For clarity of the presentation we will briefly recall them here.

The behavioral policy describes the allowed behavior of an application in terms of sequences of actions that the application is allowed to perform during its execution. For each action, the behavioral policy can also define a set of predicates that must be satisfied in order to authorize the execution of the action itself. The behavioral policy is evaluated and enforced by the Gmon module, as explained in the next section.

As an example, a behavioral policy could define the admitted behavior of an application when using client sockets. In this case, the admitted sequence of actions could be: "open the socket", "connect to the server", "send data" or "receive data" and "close the socket". The policy could define the set of servers the application can connect to. To this aim, the predicate paired with the action "connect to the server" includes a check on the server name. The policy could also state that, when connected to a server included in a given set, the application is allowed to receive data but is not allowed to send data. In this case, a predicate paired with the action "'send data" checks whether the server currently connected belongs to this set of hosts. The behavioral policy also defines when access decisions need to be taken by pairing access requests to some of the actions in the sequences. With reference to the previous example, the behavioral policy could state that only trusted users can send data on sockets because the local resource stores some critical data. In this case, the predicate paired with the action "send data" includes an access request "send data" that must be evaluated by

the access policy. In this way, the access request is evaluated when the application tries to execute the action "send data". When an access decision is needed the evaluation is performed according to the access and disclosure control policies.

The access control policy specifies what requirements a user (an application running on behalf of a user) has to satisfy in order to perform some actions on local Grid resources. Rather, the disclosure control policy specifies which credentials among those occurring in the access policy can be disclosed at any time so that if needed can be demanded from a client. The two logical policies are used by *i*Access module for taking access decisions. The intuition behind *i*Access logical module is the following. Whenever *i*Access is requested for an access decision, first it checks whether the user has enough access rights to perform the requested operation according to the access policy. If yes then *i*Access simply returns grant. If no then *i*Access steps in the interactive part of the computation where it first computes from the disclosure policy the set of disclosable credentials. Then among the disclosable credentials *i*Access computes a set of *missing credentials* that added to the user's set of active credentials grant the requested operation according to the access policy. If such a set is found then *i*Access returns those missing credentials otherwise returns deny. We refer the reader to [15] for more details on the model.

With reference to the previous example, the access policy could include the following three rules. A rule stating that the "send data" action requires the Grid user to present the role of "trusted user". The second rule could state that the role "admin" dominates the role "trusted user". The third rule could state that the role "trusted user" dominates the role "grid user".

While the disclosure policy could say that the need for a "admin" and "trusted user" certificate is disclosed only to clients that have presented a certificate attesting them as a "grid user".

Now, if a user presents a certificate for a "trusted user" (or "admin") the action "send data" will be granted (authorized) by the *i*Access module. Instead, if the user presents a "grid user" certificate then the action "send data" will be denied and a feedback to the user will be returned asking for an additional credential of "trusted user" or "admin". Here we note that the choice of either "trusted user" or "admin" depends on the minimality criteria used by *i*Access algorithm and refer the reader to [15] for details.

4 System Architecture and Functionality

4.1 The Architecture

Figure 1 shows the overall system architecture. The main components of the architecture are `Gmon`, `User's Profile` and *i*`Access`. Each of them represents a software module that has one or more interfaces to communicate with the other modules or with Globus.

The Gmon module is the access control enforcement point and the behavioral policy evaluation point of the architecture. As so it is the component integrated and connected directly with the Globus toolkit. In the standard Globus toolkit

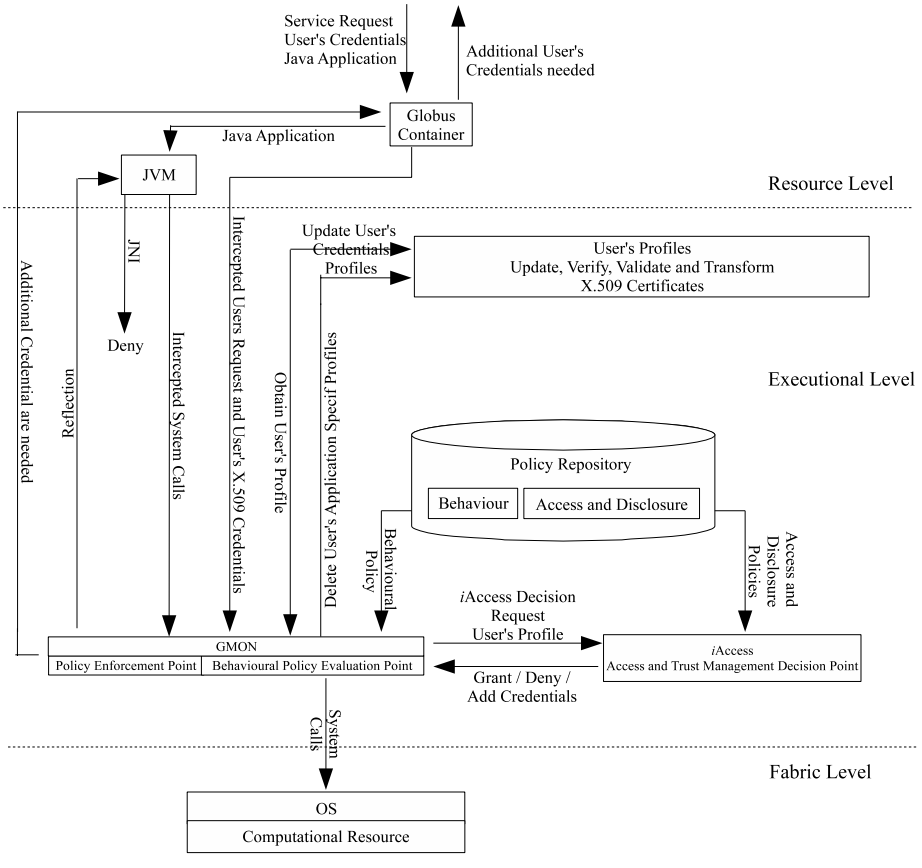


Fig. 1. System Architecture

when a job request is received first the Grid user is authenticated by his proxy certificate and then the job request is passed to Managed Job Factory Service (MJFS) implemented by the Grid Resource Allocation and Management (GRAM) component. For each job request MJFS creates a distinct Managed Job Service (MJS) that manages the application execution on the local resource [4]. In our implementation we focus our attention on Java applications because the portability of Java is an interesting feature for the Grid environment mainly because Grid is a collection of heterogeneous resources. The integration of our system with Globus requires that the MJFS interacts with Gmon to pass the job request together with the additional user's credentials and the MJS launches the Java application using a customized Java Virtual Machine. The resource requirements defined in the job request such as the main memory size or the CPU time are enforced by Gmon along with the behavioral policy while the user's credentials are passed to the User's Profile module. The customized JVM that runs Grid applications is monitored by Gmon. In particular, the JVM was modified in a way such that during a Java application execution whenever JVM

invokes a system call (interacting with the underlying operating system) Gmon intercepts the call and suspends the execution. The execution is resumed only if the outcome of all the controls performed by our system is positive.

Gmon is also the behavioral policy evaluation point in the system. As such, each time the JVM is suspended Gmon evaluates the behavioral policy in order to determine whether the current system call is allowed or not. The evaluation takes also into account the application history, i.e. the system calls previously invoked by the application.

The positive evaluation of a system call (allowing it for execution) is only if it is included in (at least) one of the sequences of actions that represent the behavioral policy and if all the system calls that precede the current one in this sequence have been already executed by the application. Moreover, the behavioral policy can also associate to a system call some predicates that have to be verified to allow the system call. These predicates include conditions on the system call parameters and results, as well as, conditions on the overall system status. These predicates can also include a request for an access decision that Gmon forwards to the *i*Access module by passing it also the set of credentials submitted by the user. A detailed description of Gmon is given in [16].

Gmon also communicates with the User's Profile module. Whenever a user submits a set of credentials together with the job request, Gmon invokes update user's profile interface presenting the user's submitted credentials and as a result of the execution Gmon receives the local user's profile (actually a path to its location) suitable for *i*Access execution. When Gmon obtains the local profile location it starts monitoring and enforcing the application behavior.

The User's Profile module provides a full-fledged integration with X.509 standard. The module is responsible for managing and to storing credentials submitted by a Grid user. It has two interfaces: `Update` User's Profiles and `Delete` User's Profiles. The former performs validation and verification of user's presented certificates (X.509 format) and generates a user's profile suitable for the underlying logical model of the *i*Access module. The latter interface deletes user's profiles as following. User's job specific profiles are deleted by Gmon when an application finishes its execution while user's general profile is deleted by Gmon when the Grid general policy specifies.

As an outcome of the Update User's Profile interface it is returned the location where the user's profile (the one suitable for the logical model) is locally stored.

*i*Access is the access control decision point of the architecture. Once invoked, *i*Access takes an access decision according to the access and disclosure policies. The *i*Access module replies to Gmon with the result of the evaluation: grant, deny or a set of additional credentials. The set of additional credentials is in the form of SAML assertion digitally signed by *i*Access.

Whenever Gmon receives the *i*Access response, it enforces the decision according to the behavioral policy. In case of missing credentials Gmon stops the application and returns the need of missing credentials back to the user. Gmon also deletes the *user's job specific profile* when an application terminates. It

does so by invoking the proper interface of the User's Profile component and specifying as input the application session id.

4.2 Functionality

Following are the key functionalities our system offers with respect to the already existing approaches.

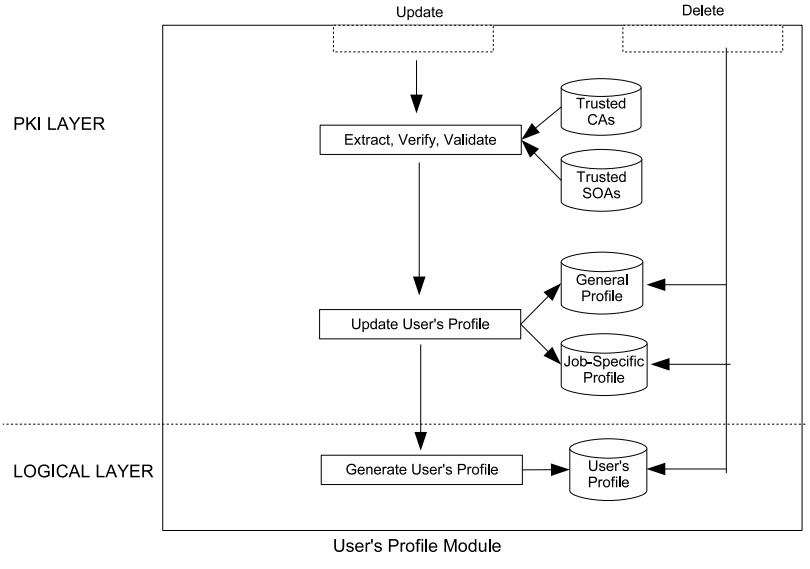


Fig. 2. User's Profile Module Architecture

- *User general profile vs multiple job specific profiles.* User general profile allows sharing of credentials among all applications running on behalf of the user. Job specific profiles allow for independent credential management, i.e. only visible to a specific application session and not shared among other user's application sessions.
  We point out that user's general profile is kept much longer than the job specific one as it is used across multiple job related executions. For each user the system keeps one general user profile and multiple user's job specific profiles.
- *Digitally signed access decision feedback* in the form of digitally signed SAML assertion [8]. *i*Access lists all missing credentials as SAML attributes wrapped into a SAML assertion.
- *Full integration with X.509 standard.* As part of *i*Access project[1] this modules integrates X.509 certificate framework [3]. Figure 2 shows the architecture of the User's Profile module. The module has two interfaces: `Update`

[1] www.interactiveaccess.org

and `Delete`. `Update` interface takes as input the user's set of X.509 certificates and the user ID. The user ID is mandatory for the interface execution. In our case, the user ID is extracted from the user's (proxy) certificate that has been submitted to the Globus system. Rather, `Delete` interface takes as input either a session ID related to a particular application execution or a user ID related to a particular user. In the former case, the interface deletes the user job specific session info together with the user's profile located at the logical layer. In the latter case the interface deletes the user general profile located at the PKI layer together with (if exists) the user's job specific profile and the respective one at the logical layer.

The certificate management is divided in five steps:

1. Extract and parse X.509 certificates,
2. Validate X.509 certificates against expiration time and data corruption,
3. Verify X.509 certificates against trusted Certificate Authorities (CAs) if identity certificate, or trusted Source of Authorities (SOAs) if attribute certificate,
4. Update User's profile according to certificates' scope,
5. Transform all user's certificates to predicates suitable for the *i*Access logical model.

The part Update takes the X.509 certificates and saves them in session (XML) files. If a session info already exists then it is updated with the newly presented certificates. The Update functionality creates a separate session for "job" marked certificates and an application specific session for each "user" marked certificates. The functionality presented above are contained in the PKI Session Layer.

The role of Generate User's Profile module is to transform and prepare the information contained in X.509 certificates in a from suitable for the *i*Access logical model, located on the Logical Layer.

5 System Implementation

Section 4 shows that our framework integrates three software components: Globus, Gmon and *i*Access. This section describes the implementation details that concern the integration of those components.

5.1 Globus

The Globus Toolkit v4 (GT4) provides a set of Grid infrastructure services that implement interfaces for managing computational, storage and other resources [4]. The Globus Container runs services provided by local resources including the standard Globus services and the custom services developed by Grid resource providers. The Grid Resource Allocation and Management (GRAM) is the Globus service that is in charge of execution management of applications submitted by remote Grid users.

The integration of our framework with the Globus toolkit was easy due to the high modularity of the Globus architecture. The Globus toolkit was modified basically for two tasks:

– defining and managing the `java` job type;
– transferring submitted credentials by Grid users to the Grid services and managing them to support the access control decisions.

Grid users issues a job request for a service in order to submit an application to Grid. The job request is an XML document that conforms to the Resource Specification Language (RSL) to describe the job features through two type of information:

– the resource requirements such as the main memory size or the CPU time
– the job configuration such as the file transfers to be executed to get the application code and its data, the name of the executable file, the input parameters, the file for the input and the output of the application, and so on.

The standard Globus job description schema defines the following job types: mpi, single, multiple and condor. We defined the `java` job type. To this aim, we modified the job description schema by adding the `java` job type in the Job Type Enumeration field. To manage the `java` job type, instead, we modified the Job Manager Perl script included in the Managed Job Service (MJS) that submits the application executable code to the scheduler of the local resource. The modified script, in the case of `java` jobs, submits the application specified in the executable field of the job request to our customized JVM. In this way, when a Grid user wants to execute a Java application he specifies `java` as a job type in its job request. Then on the computational service side Globus automatically invokes the security enhanced JVM in order to execute the submitted application.

The credentials submitted by a remote Grid user are also included in the job request by exploiting the extension field. In Globus, the job request is examined by the Managed Job Factory Service (MJFS) of the GRAM component to extract the requirements to instantiate a proper MJS for managing the request. We modified the MJFS to export the job request to Gmon. In this case the MJFS exports the job request on a local file that will be loaded by Gmon. At this stage Gmon is not started yet. The jobDescriptionType and RSLHelper classes of the Globus package provide simple methods to convert the job description in a text string to be saved in a file.

The standard error stream is used in order to return to Grid users error messages when an application has been stopped because of a security policy violation. It is exactly the case when the credentials submitted by a user are not enough to allow user's application to execute an action. In this case the error message specifies the credentials that the user additionally needs to provide. The standard Globus mechanism to transfer files, i.e. GridFTP, is exploited to send to the remote Grid user the log file with the error description.

5.2 Gmon

The successful Gmon integration within the framework requires Gmon to interact with the User's Profile component and with *i*Access. Gmon interacts with Globus in order to get the service request submitted by the remote Grid user. This

interaction is simply implemented through an XML file that is generated by Globus as described in the previous section. Gmon simply loads it during the initialization phase. Once it is loaded, Gmon extracts from the job request the resource requirements along with the submitted credentials. Gmon also imports from Globus the user ID extracted from the proxy certificate that has been submitted by the user.

Next, Gmon invokes the User's Profile module as an external library function in order to update the user profile by passing the submitted credentials (in X.509 format) and the user ID.

Since the behavioral policy defines the predicates that Gmon has to evaluate for each action, access requests are simply represented as new predicates in the policy and the evaluation of those predicates consists of the invocation of the *i*Access component with input the action request and the link to the user's local profile. The *i*Access (as well as User's Profile component) are invoked by Gmon as an external library functions.

The interactions between Gmon and *i*Access are the most critical part of the architecture in term of performance. Since *i*Access module is Java based and Gmon is C based, we have to provide a way to invoke *i*Access without loading the JVM every time. Since *i*Access is an internal and trusted application it is executed by exploiting a standard JVM. We use the Java Native Interface (JNI) library [17] that allows to invoke Java methods from inside a C code. As a part of the functionality of this library we have a special command that explicitly loads the JVM needed for the execution of a Java class and also commands to obtain links to a Java class and its methods. A separate command is provided by JNI for releasing the JVM. In this case, at initialization time, Gmon loads the JVM and creates the links to the *i*Access methods and keeps them in memory for all the subsequent invocation of *i*Access. When the application terminates Gmon release the JVM before it terminates.

5.3 User's Profiles

The Update User's Profile interface has as input the path of an XML file that contains the certificates presented by the user and the user ID extracted from the user X.509 proxy (identity) certificate.

A certificate sent to the system has an additional property scope indicating *user-* or *job-specific* lifetime. Following is an example of user input of identity and attribute certificates.

```
<UserInput>
    <X509V3Cert type="id" scope="user">
            MIIBtjCCAwIBAgICAN4wDQYJKoZIhvcNAQEFBQAwTTELMAkGA1UEBtTAPB
            XcKJyiqoOkY68DcwPVahmd ....
     </X509V3Cert>
     <X509V3Cert type="attr" scope="job">
            MIIBtjCCAR8CAQEwbKFqpGgwZjELMAkGA1UEBhMCSVQxDDAKBgNVByQzE
            VFJFTlRPMRIwEAYDVQQDEwlM ....
     </X509V3Cert>
</UserInput>
```

Identity Certificates are implemented with the standard library of Sun JDK while the Attribute certificates are implemented with the Bouncy Castle[2] security libraries.

Each identity certificate has a subject name and an issuer name while, in contrast, each attribute certificate has a holder name, attribute value and issuer name. Those are the relevant fields for the `Generate User's Profile` function. The role of this function is first to match identity certificates with attribute certificates in order to extract appropriate information regarding the user's access rights. The matching checks whether the serial number of an identity certificate is within the field of the holder name in an attribute certificate, as recommended by ITU-T for use of X.509 specification.

Once the matching is finished then the `Generate User's Profile` function generates appropriate predicates according to the matching outcome. The outcome of the generation is a set of logical predicates describing user's identity and attributes.

5.4 *i*Access

*i*Access is a Java-based access decision engine that implements the interactive access control model [15]. DLV system is used as an underlying logical engine for logical operations. Particularly, for the integration with Gmon *i*Access engine has been optimized only to the level of taking access decision without managing user's credentials in contrast to the model in [15].

6 System Preliminary Evaluation

Our preliminary results we performed to evaluate the impact on the adoption of our framework on the execution time of Java applications. This overhead strongly depends on the specific Java application, on the enforced policy and on the number of certificates submitted by the user.

If the application performs only few system calls or if the security policy is very simple the impact of the security checks on the overall execution time is negligible with respect to the application execution itself.

The first set of experiments evaluates the execution time of two Java applications adopting a standard JVM, a JVM enhanced with Gmon and a JVM enhanced with Gmon and *i*Access. We exploited the Jikes RVM Java Virtual Machine developed by the IBM T.J. Watson Research Center [18], running on Linux (FC4 with 2.6.13 kernel) operating system. The applications we chose are two benchmarks from the Ashes Suite Collection benchmarks[3], that are suitable for our tests because they perform a large number of system calls. In our experiments we adopted a simple policy similar to the one shown in [14]. The performances are shown in Figure 3. Javazoom is an mp3 to wav converter. It performs about 1500 system call in about 5 seconds. Most of these system calls

[2] `http://www.bouncycastle.org`

[3] http://www.sable.mcgill.ca/ashes

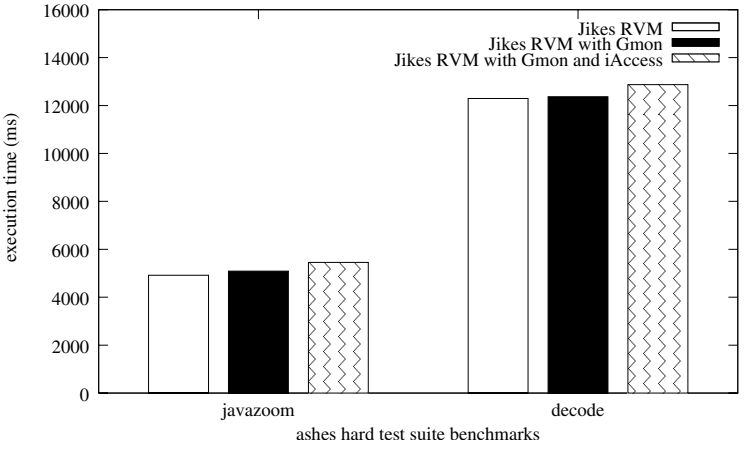


Fig. 3. Java benchmarks execution times

are issued to read the mp3 file and to write the wav one. Our system controls whether javazoom has the credentials to open the mp3 file, to create the wav one and whether access to these files is compatible with the admitted behavior described by the behavioral policy. In this case, the overhead introduced by Gmon only is about 4%, while the overhead by Gmon and *i*Access is about 10% of the execution time of the application.

Decode is an algorithm for decoding encrypted messages that uses the Shamir's Secret Sharing scheme. It executes about 570 system calls in about 12 seconds. In this case, the overhead introduced by Gmon is about 1% while the overhead by Gmon and *i*Access is about 5% of the application execution time.

The second set of experiments concerns the evaluation of the performance of the `User's Profile` module. It has been done on Linux (FC4 with 2.6.13 kernel) operating system as well. The table below shows the several sessions we have performed. For each row we run the Update User's Profile prototype with a different number of certificates.

Number of Certs	5	10	20	50
Update User's Profile (execution time ms)	452	589	727	1884

All the times are expressed in milliseconds. The Update process has different phases (refer to Figure 2). The time table shows that having 50 certificates (which is quite a few for a user to have) the time is less than two seconds. It indicates that the time response of the `User's Profile` module is acceptable whit respect to the overall system performance.

7 Conclusions and Future Work

We have presented an access control system that enhances the access control mechanism in the Globus toolkit. The key system functionalities with respect

to other access control systems are: (i) fine-grained behavioral control combined with a credential-based access control mechanism; (ii) integration with X.509 certificate standard and application-level granularity management of user's credentials; (iii) access control feedback when users do not have enough permissions to use Grid resources.

Even if the adoption of different policies, different benchmarks or different sets of user's certificates could result in very different performances, we believe that the results from the preliminary evaluation confirm that it is possible to define a fine-grained security policy that integrates behavioral and trust management aspects and that the overhead to enforce is affordable.

Future work is on improving the performance of the system and performing experimental assessments when applying it on real Grid scenarios.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. International Journal of Supercomputer Applications **15**(3) (2001) 200–222
2. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid: An open grid service architecture for distributed system integration. Globus Project (2002) http://www.globus.org/research/papers/ogsa.pdf.
3. X.509: The directory: Public-key and attribute certificate frameworks (2001) ITU-T Recommendation X.509:2000(E) | ISO/IEC 9594-8:2001(E).
4. Foster, I.: Globus toolkit version 4: Software for service-oriented systems. In: Proceedings of IFIP International Conference on Network and Parallel Computing. Volume 3779., Springer-Verlag, LNCS (2005) 2–13
5. Foster, I., Kesselman, C., Pearlman, L., Tuecke, S., Welch, V.: A community authorization service for group collaboration. In: Proceedings of the 3rd IEEE Int. Workshop on Policies for Distributed Systems and Networks (POLICY 02). (2002) 50–59
6. Pearlman, L., Kesselman, C., Welch, V., Foster, I., Tuecke, S.: The community authorization service: Status and future. Proceedings of Computing in High Energy and Nuclear Physics (CHEP 03): ECONF **C0303241** (2003) TUBT003
7. XACML: eXtensible Access Control Markup Language (XACML) (2005) www.oasis-open.org/committees/xacml.
8. SAML: Security Assertion Markup Language (SAML) (2005) www.oasis-open.org/committees/security.
9. Keahey, K., Welch, V.: Fine-grain authorization for resource management in the grid environment. In: GRID '02: Proceedings of the Third International Workshop on Grid Computing - LNCS. Volume 2536. (2002) 199–206
10. Thompson, M., Essiari, A., Keahey, K., Welch, V., Lang, S., Liu, B.: Fine-grained authorization for job and resource management using akenti and the globus toolkit. In: Proceedings of Computing in High Energy and Nuclear Physics (CHEP03). (2003)
11. Thompson, M., Johnston, W., Mudumbai, S., Hoo, G., Jackson, K., Essiari, A.: Certificate-based access control for widely distributed resources. In: Proceedings of Eighth USENIX Security Symposium (Security'99). (1999) 215–228

12. Stell, A.J., Sinnott, R.O., Watt, J.P.: Comparison of advanced authorisation infrastructures for grid computing. In: Proceedings of High Performance Computing System and Applications 2005, HPCS. (2005) 195–201
13. Chadwick, D.W., Otenko, A.: The PERMIS X.509 role-based privilege management infrastructure. In: Seventh ACM Symposium on Access Control Models and Technologies, ACM Press (2002) 135–140
14. Koshutanski, H., Martinelli, F., Mori, P., Vaccarelli, A.: Fine-grained and history-based access control with trust management for autonomic grid services. In: Proceedings of the 2nd International Conference on Autonomic and Autonomous Systems (ICAS'06), IEEE Computer Society (2006)
15. Koshutanski, H., Massacci, F.: Interactive access control for Web Services. In: Proceedings of the 19th IFIP Information Security Conference (SEC 2004), Toulouse, France, Kluwer Press (2004) 151–166
16. Martinelli, F., Mori, P., Vaccarelli, A.: Towards continuous usage control on grid computational services. In: Proceedings of Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS-ICNS 2005), IEEE Computer Society. (2005) 82
17. Liang, S.: Java(TM) Native Interface: Programmer's Guide and Specification. Addison-Wesley (1999)
18. Alpern, B., Attanasio, C., Barton, J., et al.: The jalapeño virtual machine. IBM System Journal **39**(1) (2000) 211–221

Dynamic Reconfiguration of Scientific Components Using Aspect Oriented Programming: A Case Study

Manuel Díaz, Sergio Romero, Bartolomé Rubio,
Enrique Soler, and José M. Troya

Department of Languages and Computer Science, University of Málaga, 29071 Spain
{mdr, sromero, tolo, esc, troya}@lcc.uma.es

Abstract. This paper is a case study on the use of a high-level, aspect-oriented programming technology for the modelling of the communication and interaction scheme that affects the set of components of a parallel scientific application. Such an application uses domain decomposition methods and static grid adaptation techniques in order to obtain the numerical solution of a reaction-diffusion problem modelled as a system of two time-dependent, non-linearly coupled partial differential equations. Besides achieving the usual advantages in terms of modularity and reusability, we aim to improve the efficiency by means of dynamic changes of aspects at runtime. The application design considers two types of components as first-order entities: Scientific Components (SCs) for the computational tasks and Communication Aspect Components (CACs) for dynamic management of the communication among SCs. The experiments show the suitability of the proposal as well as the performance benefits obtained by readjusting the communication aspect.

1 Introduction

Recently, significant efforts are being made in order to incorporate component-oriented programming [5] to develop new parallel and distributed programming environments (PEs) in the high-performance computing area. In this sense, ASSIST [11] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. SBASCO [1] is a PE oriented to the efficient development of parallel and distributed numerical applications. A large effort is currently being devoted to defining a standard component architecture for high-performance computing in the context of the Common Component Architecture (CCA) Forum [10].

On the other hand, aspect-oriented programming (AOP) [6] enables developers to capture the crosscutting structure that is spread over the different components of a system. This allows crosscutting concerns to be programmed in a modular way, and to achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and has a greater potential for reuse. A well-modularised crosscutting concern is called an aspect.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1351–1360, 2006.
© Springer-Verlag Berlin Heidelberg 2006

This paper discusses the development of a parallel scientific application using a high-level programming technology that integrates software components and aspects. The solution is based on the SBASCO extension described in [2], which considers the encapsulation of several functionalities affecting the set of scientific components into separate aspects which are also modelled as components. More specifically, this work is focused on dynamic management of the application communication scheme and processor layout.

The reaction-diffusion problem considered here is described as a system of two time-dependent, non-linearly coupled partial differential equations (PDEs). Domain decomposition methods [9] are used to achieve the solution in a two-dimensional rectangular domain of long-aspect ratio, i.e. $L_x > L_y$. These methods divide the original geometry into separate regions so that a parallelization strategy can be applied to tackle large-scale and realistic problems.

The problem solution is characterized by the presence of a travelling wave front in the x-direction. Due to the steepness and the curvature of the front, a large number of grid points are needed to accurately calculate the numerical solution of the system of equations. Alternatives to equally spaced grids that can reduce the computational cost are those based on grid-adaptive procedures [7]. We use a static grid adaption technique that concentrates the grid points in an adaptive static fashion on regions where they are needed.

Two different levels of parallelism are combined in the application. On the one hand, the solution in the different subdomains is computed in parallel by assigning one task (component) per subdomain. On the other hand, scientific components use data-parallel red-black Gauss-Seidel relaxation to solve the large systems of linear equations so that data-parallel components are considered.

The high-level application design presented here allows the developer to be released from the tedious and error-prone parallelism exploitation details, such as the creation, synchronization and communication of data-parallel tasks, which are automatically managed by the runtime support. The communication scheme is encapsulated into a Communication Aspect Component (CAC) to which the different Scientific Components (SCs) are connected. During the execution, the CAC can detect that a SC is running slower than the others, which decreases the application performance. In response, the CAC can dynamically carry out a re-mapping of the processors being assigned to the SCs and establish a new efficient communication scheme for component interaction.

This paper is structured as follows. The next section outlines the fundamentals of the framework the solution is based on. Section 3 presents the physical problem taken into consideration. Section 4 describes the design of the application and its implementation. Experimental results are shown in section 5. Finally, some conclusions are outlined.

2 Aspect-Oriented Framework Overview

The aspect-oriented framework [2] used in this work represents an evolution of SBASCO in the sense that this is extended in an attempt to apply the AOP

paradigm to the high-performance computing scene. The application domain is the parallel and distributed solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains, and other kinds of problems that take advantage of integrating task and data parallelism and have a communication pattern based on (sub)array interchange. Usually, their execution times are long.

The approach provides the basis for separating specific aspects that appear in the above-mentioned application domain, such as communication, synchronization, distribution, convergence criteria, numerical method, boundary conditions, etc, which are managed as first-order entities.

There are two types of components: Scientific Components (SCs), which are in charge of computational tasks, and Aspect Components (ACs) used to encapsulate crosscutting concerns. Both types of components are combined in order to construct "aspectized" scientific applications.

SCs interact with each other following a "data flow" style by means of a typical `getData`/`putData` scheme based on the SBASCO approach, whereas the interaction among SCs and ACs follows a "procedural" style based on asynchronous method calls. The language used to specify the interfaces of both SCs and ACs has a syntax similar to that used in CCM, enriched by constructors and data types characterizing the programming model of SBASCO.

In SBASCO, applications are static in the sense that configuration parameters such as the number of processors assigned to each component are fixed at the composition level. However, the evolution of the system during execution may require a dynamic readjustment of these parameters in order to preserve the efficiency. As a consequence, the communication scheme among the different components, which depends on this information, should also be modified.

In our approach, the communication scheme is an example of a crosscutting concern which is encapsulated into a Communication Aspect Component (CAC). Such a scheme is described using a software skeleton which specifies data distribution, processor layout and, possibly, the number of replicas of each SC. A set of predefined skeletons is inherited from the SBASCO programming model.

The strategy or actions followed to dynamically reconfigure an application as well as the conditions under which such actions are applied can be easily programmed as part of the CAC.

3 Problem Formulation and Discretization

Physical phenomena involving heat and mass transfer, combustion, etc. are characterized by reaction-diffusion equations with non-linear source terms. Here, we consider the following set of two time-dependent, nonlinearly coupled PDEs:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + S(U), \tag{1}$$

where

$$U = (u, v)^T, \qquad S = (-uv, uv - \lambda v)^T, \tag{2}$$

u and v represent the concentration of a reactant and the temperature, respectively, $u = 1$ and $v = 0$ on the external boundaries, t is time, x and y denote Cartesian coordinates, λ is a constant (in this paper, $\lambda = 0.5$), and the superscript T denotes transpose. Eq. (1) has been previously studied in [8], where a comparison of several numerical techniques for tackling domain decomposition problems in irregular domains is presented.

Eq. (1) was discretized by means of an implicit, linearized, θ-method where the non-linear term $S_{i,j}^{n+1}$ was approximated by means of its Taylor polynomial of first degree around (t^n, x_i, y_j) to obtain the following system of linear algebraic equations:

$$\frac{\Delta U_{i,j}}{k} = \frac{1}{\Delta x^2}\left[\theta\delta_x^2 \Delta U_{i,j} + \delta_x^2 U_{i,j}^n\right] + \frac{1}{\Delta y^2}\left[\theta\delta_y^2 \Delta U_{i,j} + \delta_y^2 U_{i,j}^n\right] + S_{i,j}^n + \theta J_{i,j}^n \Delta U_{i,j}, \tag{3}$$

where

$$\Delta U_{i,j} = U_{i,j}^{n+1} - U_{i,j}^n, \quad S_{i,j}^n = S(U_{i,j}^n), \quad J_{i,j}^n = \frac{\partial S}{\partial U}(t^n, x_i, y_j)$$
$$\delta_x^2 U_{i,j} = U_{i+1,j} - 2U_{i,j} + U_{i-1,j}, \quad \delta_y^2 U_{i,j} = U_{i,j+1} - 2U_{i,j} + U_{i,j-1}, \tag{4}$$

i and j denote x_i and y_j, respectively, t^n denotes the nth time level, k is the time step, Δx and Δy represent the grid spacing in the x- and y-directions, respectively, and $0 < \theta \leq 1$ is the implicitness parameter. In this paper, $\theta = 0.5$, i.e., second-order accurate finite difference methods are employed.

4 Application Design and Implementation

This section covers a series of topics such as the algorithm description and its parallelization, the application design and the programming of the components.

4.1 Algorithm Description and Parallelization Strategy

In order to apply domain decomposition, the geometry of the problem is divided into several overlapping subdomains. In the example shown in Fig. 1, each subdomain $\Omega_k, 1 \leq k \leq 4$, is the rectangular area delimited by the cartesian points `(dkl,y0)` and `(dkr,y1)`. The part of the boundary of Ω_i that is interior to Ω_j is denoted by $\Gamma_{i,j}$. In the Dirichlet method considered here, the current solution in one subdomain is used to update the boundary data in the adjacent subdomains. Then, the interior points are recalculated by solving the system of linear equations, Eq. (3). This iterative procedure is repeated until convergence.

In addition, we are interested in obtaining a more accurate numerical result in those areas where the solution presents steep gradients and high curvature. A simple static grid refinement procedure is used to increase the number of grid points where the travelling wave front is located. As the propagation of the front is mainly carried out in the x-direction, the number of points in that direction

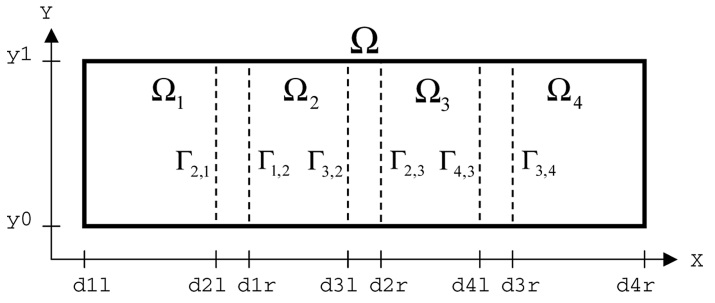


Fig. 1. Original two-dimensional domain divided into overlapping regions

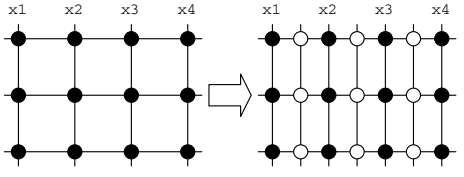


Fig. 2. Double-sized grid used to improve the accuracy of the solution

is duplicated as well as the grid spacing parameter, Δx, is reduced by half. The unknown points of the refined grid, which are denoted by white dots in Fig. 2, are calculated by means of cubic spline interpolation using the current solution available in the black dots. This procedure is applied when $||[\partial v/\partial x]||$ exceeds a predefined constant. From that instant, the double-sized grid is used in the calculation of each time step until the previous criterion fails, which means that the wave front does not significantly affect the subdomain.

The operations carried out in each subdomain are summarized as follows:

```
 1.  Set initial conditions; is_duplicated = FALSE
 2.  For time_step = 1..MAX_TIME_STEPS Do
 3.    If (is_duplicated = FALSE) AND (slope > eps) Then
 4.      Interpolate; use double-sized grid; is_duplicated = TRUE
 5.    If (is_duplicated = TRUE) AND (slope < eps) Then
 6.      Use simple grid; is_duplicated = FALSE
 7.    Repeat
 8.      Update boundaries
 9.      Solve system of linear equations
10.   Until convergence on ALL subdomains
```

The above algorithm is parallelized by associating one task per subdomain. These tasks are encapsulated into SCs comprising the application so that, we have different instances of components running in parallel to solve the problem. In addition, components internally exploit parallelism as they use data-parallel red-black Gauss-Seidel relaxation to solve the system of linear equations. As a

```
Interface ScConfigure {
  void go(in Domain2D dom, in CommScheme scheme, in unsigned procs,
          in unsigned iters);
  void continue(in unsigned iters);
  void changeCommScheme(in CommScheme scheme);
  void changeNumProcs(in unsigned procs);
};
Interface CacProvided {
  void finished(in Sc sc);
};

component Sc {
  provides ScConfigure conf;
  uses CacProvided cac;
};
component Solve1 : Sc{};
component Solve2 : Sc{};
...

component Cac {
  skeleton ReactionDiffusion {
    Domain2D dom1 = {d1l,y0,d1r,y1},
             dom2 = {d2l,y0,d2r,y1},
             ...
    Multiblock {
      Solve1(dom1:(BLOCK,*)) ON PROCS(2);
      Solve2(dom2:(BLOCK,*)) ON PROCS(2);
      ...
    WithBorders
      dom1(d1r,y0,d1r,y1) <- dom2(_);
      dom2(d2l,y0,d2l,y1) <- dom1(_);
      ...
    };
  };
  provides CacProvided cac;
  uses ScConfigure conf;
};
```

Fig. 3. Definition of SCs and CAC for the reaction-diffusion application

consequence, this application design means the efficient integration of both data and task parallelism.

4.2 Scientific and Communication Aspect Components

The software components considered in the application development are shown in Fig. 3. The definition of the CAC, besides specifying the interfaces provided and used, establishes the `Multiblock` skeleton that describes the interaction scheme among the SCs. Such a skeleton is focussed on the solution of multi-block and domain decomposition-based problems. The predefined type `Domain2D` is used to capture the Cartesian points that establish the region of a domain [3].

On the other hand, the type `CommScheme` will contain the required information, which is declared in the skeleton, in order to carry out an efficient inter-component communication: data distribution, domain interactions and mapping of processors. The data distribution types are HPF-like. For example, the expression `dom1:(BLOCK,*)` declares a distribution by rows for the domain `dom1`.

As stated before, the problem is solved iteratively, so that for each iteration a communication of the internal boundaries, which are defined among the domains, is needed. For example, the expression `dom1(d1r,y0,d1r,y1)<-dom2(_)` indicates that the zone of `dom1` delimited by points `(d1r,y0)` and `(d1r,y1)` will

```
void Cac::configRule() {
  int i;
  Sc *comp, *worstComp;

  worstComp = appComponents.worst;
  if (appComponents.availableProcs > 0)
    worstComp->increaseProcs(1);
  else {
    for (i=0; i < appComponents.numComps; i++)
      if (appComponents.fromBestToWorst[i]->procs > 1) {
        comp = appComponents.fromBestToWorst[i];
        break;
      }
    if ((i < appComponents.numComps) &&
        (comp->execTime < worstComp->execTime * alpha)) {
      comp->decreaseProcs(1);
      worstComp->increaseProcs(1);
    }
  }
}
```

Fig. 4. Implementation of a re-configuration strategy in the CAC

be updated by the values belonging to the zone of `dom2` delimited by the same points. SCs communicate their internal boundaries by means of the predefined data flow primitives `getData` and `putData` which, respectively, obtain data from and send data to a SC.

In addition, the relationship between domains and the processor layout where the SCs are going to be executed is also indicated, e.g. `ON PROCS(2)`. Each SC is solved by a disjoint set of processors.

4.3 Application Programming and Execution

The development of both scientific and aspect components is carried out following an object-oriented programming style based on C++ and MPI.

The programming of the CAC includes the implementation of two methods in the class that represents the component in order to allow easy programming of the dynamic reconfiguration procedure.

The `history` method uses the predefined object `schedule` to establish a set of time stamps that represent the points at which dynamic changes in the application will (possibly) be applied. For example a call to `schedule.cycle(100)` means each time a hundred time steps are computed by the SCs, the CAC decides if a re-configuration of the application is needed. Time stamps can also be individually set up by calling `schedule.setTimeStamp( time_stamp)`.

The reconfiguration operation is coded in the `configRule` method. The programmer uses the `appComponents` object to query the different components and perform actions on them. Figure 4 proposes a very simple but effective implementation of this method. It must be noted that much more complicated strategies based on the evaluation of a large amount of parameters could be applied.

The programming of the SCs consists of writing the scientific code. The initial conditions of the problem are established in the `initialize` method. The code of the `task` method is executed each time new input data are available.

Most of the functions are called from here: the selection between the simple and double-sized grid, the interpolation procedure, the method to solve the system of equations, etc. The data-parallel Gauss-Seidel technique is implemented using MPI.

When the number of processors of a SC is dynamically changed, the system guarantees that the current state of the computation is maintained by means of a correct data re-distribution. This is automatically carried out for the domain, which is the only data declared at the composition level. However, other internal variables may also be considered part of the component state. So, the SC provides the methods `preMapping` and `postMapping` which are executed, respectively, immediately before and after the re-mapping of processors. The programmer can implement these operations to communicate his/her variables as needed.

The sequence of application execution begins when the CAC calls `go` on the SCs, so that the domain, initial communication scheme and number of processors are established. The number of iterations used corresponds to the first time stamp. The `go` operation calls `initialize` and `task`. When a SC finishes its computational task, it notifies the CAC calling `finished`. Then, the `configRule` method evaluates the reconfiguration of the application. For example, let us suppose that the number of processors of the SC `Solve1` needs to be incremented by one. In that case, the CAC calls `changeNumProcs` on the relevant SC, which executes the redistribution code. Additionally, some other SCs must be informed about the change in the communication scheme. More specifically, the CAC calls `changeCommScheme` on `Solve2` as it is the only component that communicates with `Solve1` (subdomains `dom1` and `dom2` are connected by internal boundaries, see Fig 3). The remaining components can preserve their communication scheme. Finally, the CAC calls `continue` on the SCs so that the computation goes on.

The solution is constructed in the context of the framework introduced in section 2. The creation and reconfiguration of SCs are based on the process management functions defined in MPI-2 [4]. The standard also provides the mechanisms for communicating stand-alone parallel applications (SCs in our context).

5 Experimental Results

The application has been used to solve Eq. (1) in a rectangular domain using four overlapping subdomains as described in Fig. 1.

The solution is characterized by a reaction front that propagates across the domain. Once the front reaches the upper and lower borders, the propagation is carried out in the x-direction, as shown in Fig. 5.

Table 1 summarizes the execution time, in seconds, for the different experiments carried out. Two problem instances have been considered, based on the use of four subdomains of sizes 100x100 and 100x200, respectively, with an overlapping length of 20 points. Processors are equally distributed among the SCs.

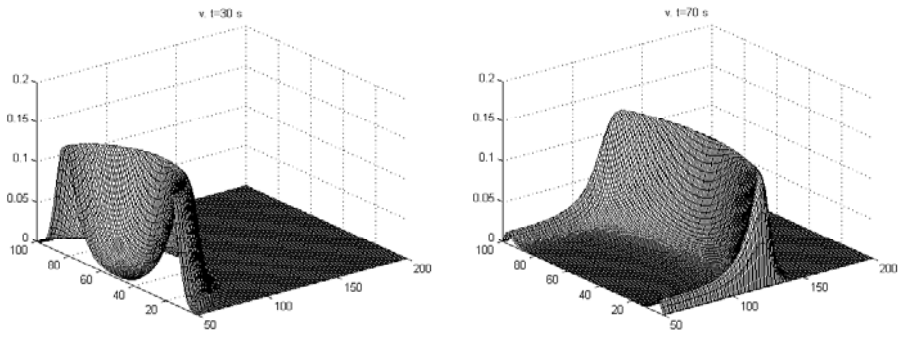


Fig. 5. Spatial distribution of temperature at $t = 30$ (left), and 70 seconds (right)

Table 1. Execution time (in seconds) for the reaction-diffusion problem. In brackets, percentage improvement obtained by using dynamic reconfiguration of components.

		Number of processors					
Domain	Type	4	6	8	10	12	14
100x340	static	1236.4	1045.2	719.7	641.1	505.5	479.3
	dynamic	-	655.2	532.9	473.4	415.7	379.9
			(37.31%)	(25.95%)	(26.15%)	(17.76%)	(20.73%)
100x740	static	4656.8	3645.6	2665.5	2268.6	1884.1	1767.83
	dynamic	-	2264.7	1938.6	1559.0	1358.6	1236.05
			(37.87%)	(27.27%)	(31.27%)	(27.88%)	(30.08%)

In cases where the number of processors is not a multiple of 4, the two SCs associated with the domains Ω_2 and Ω_3 (see Fig. 1) obtain one additional processor each. Two types of component configurations are evaluated: static, using a fixed processor mapping, and dynamic, carrying out processor re-assignments at runtime based on the strategy described in Fig. 4. Experiments were performed using a 1Gb/s Myrinet network of Pentium 4, 2.66GHz, 1GB RAM Linux workstations running LAM/MPI as message passing interface implementation.

Results show significant improvements when using dynamic reconfiguration of components. In our application, the computational cost of a scientific component evolves as it is mainly influenced by the grid refinement technique used. The synchronization of tasks imposed by the boundaries updating means that the overall performance of the application decreases when any of the components are running slower than the others. In such conditions, any of the results obtained by means of a static processor mapping will incur a reduction in performance as some of the processors will be idle during some periods of time. The redistribution of processors at runtime overcomes this shortcoming.

6 Conclusions

The implementation of efficient communications in parallel scientific applications needs to consider a variety of factors such as the interaction scheme among tasks, the distribution of data, the mapping of processors and so on.

In our approach, all the parameters that influence the interaction among Scientific Components (SCs) are encapsulated into a special type of component, the so-called Communication Aspect Component (CAC). In addition, this component can perform dynamic changes in the application, e.g. readjustments in the processor mapping, thereby establishing new efficient communication schemes.

The modelling of communications using aspect-oriented programming (AOP) allows the SCs to focus only on the computational tasks, so that significant improvements in terms of modularity and code reusability are obtained. Moreover, the performance of the application increases as the dynamic changes carried out by the CAC lead to a better utilization of the computational resources.

References

1. Díaz, M., Rubio, B., Soler, E., Troya, J.M., SBASCO: Skeleton-Based Scientific Components, in "Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2004)", pp. 318–324, IEEE Computer Society, A Coruña, Spain, 2004.
2. Díaz, M., Romero, S., Rubio, B., Soler, E., Troya, J.M., An Aspect-Oriented Framework for Scientific Component Development, in "Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2005)", pp. 290–296, IEEE Computer Society, Lugano, Switzerland, 2005.
3. Díaz, M., Rubio, B., Soler, E., Troya, J.M., A Border-based Coordination Language for Integrating Task and Data Parallelism, *Journal of Parallel and Distributed Computing*, **62**, 4, pp. 715–740, 2002.
4. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M., MPI: The Complete Reference, volume 2–The MPI-2 Extensions. MIT Press, 1998.
5. Heineman, G.T., Council, W.T., Component-Based Software Engineering: Putting the Pieces Together. Addision Wesley, 2001.
6. Kiczales, G., et. al., Aspect-Oriented Programming, in "Proceedings of the Europe Conference on Object-Oriented Programming (ECOOP 1997)", LNCS 1241, pp. 220–242, Springer, Jyvskyl, Finland, 1997.
7. Lang, J., High-Resolution Self-Adaptive Methods for Turbulent Chemically Reacting Flows, *Chemical Engineering Science*, **51**, pp. 1055–1070, 1996.
8. Ramos, J.I., Soler, E., Domain Decomposition Techniques for Reaction-Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners, *Applied Mathematics and Computation*, **118**, 2-3 (2001), pp. 189–221.
9. Smith, B., Bjørstad, P., Gropp, W., Domain Decomposition. Parallel Multilevel Methods for Elliptic P.D.E.'e. Cambridge University Press, 1996.
10. Common Component Architecture Forum, home page http://www.cca-forum.org.
11. Vanneschi, M., The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications, *Parallel Computing*, **28**, 12, pp. 1709–1732, 2002.

MGS: An API for Developing Mobile Grid Services

Sze-Wing Wong and Kam-Wing Ng

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, New Territories, Hong Kong SAR
{swwong, kwng}@cse.cuhk.edu.hk

Abstract. This paper presents an application programming interface called MGS which supports service development in the Mobile Grid Service middleware framework. Mobile Grid Services, the extension of the original static Grid services, are characterized by the ability of moving from nodes to nodes during execution. By combining an existing mobile agent system (JADE) and a generic grid system toolkit (Globus), the Mobile Grid Services framework is constructed. The API consisting of the AgentManager class, Task Agent template and configurable Monitor Agent is used to provide both an easy and flexible environment for Mobile Grid Services development.

1 Introduction

Grid computing has been drawing a lot of attentions from both academia and industry in recent years. A Grid [1] is a set of resources distributed over wide-area networks that can support large-scale distributed applications. The Grid problem [2] is defined as the coordinated resource sharing and problem solving in dynamic, multi-institutional, virtual organizations. Grid computing enables users and systems to dynamically share resources and balance the loading on resources.

Regarding to current grid technologies, service providers usually provide stationary services. They are fixed on the Grid node machines providing the services and cannot be moved to other nodes even if these other nodes have lots of idle resources.

To overcome the deficiency of static Grid Services, the concept of Mobile Grid Services is proposed as an extension. Mobile Grid Services are able to leave their hosts and migrate to other Grid nodes containing more idle resources. We can coordinate both services and resources by this kind of service mobility and maximize the resource usages of grids. Since the execution of a Mobile Grid Service can be migrated to other nodes, the overload problem can be relieved even if a large amount of service requests are present at the same time. This can also improve the practicability and flexibility of Grid Services.

This paper is structured as follows: The next section illustrates the overview of our proposed Mobile Grid Services framework. To ease the development of Mobile Grid Services, the MGS API is implemented. In Section 3, we will discuss the issues of designing the API. Section 4 presents the details of the MGS API. Section 5 shows some work by others related to Mobile Grid Services. Finally, Section 6 summarizes our conclusion and presents future work on our framework.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1361 – 1375, 2006.
© Springer-Verlag Berlin Heidelberg 2006

2 Mobile Grid Services Framework [12]

We are working on a project aiming to develop a middleware framework for Mobile Grid Services. The framework is constructed by joining the Java Agent Development Framework (JADE) [7] and the Globus Toolkit [3].

In our framework, a JADE mobile agent (supporting weak mobility) is used to encapsulate the actual working unit of the application task. JADE containers are setup throughout the grid to provide platforms for mobile agents to move on. The communication between these JADE agents is accomplished by exchanging messages in the format of the Agent Communication Language (ACL) defined by the FIPA specification [16].

For each application, a Globus grid service will create the corresponding mobile agents and act as a relay between users and the mobile agents. In this way, Mobile Grid Services are realized as a type of grid services which distribute the actual working tasks to mobile agents. The advantage of this design is that the existence of the relay part makes the Mobile Grid Services conform to the Globus grid services architecture. At the same time, the mobile agent part is able to exploit useful resources in other grid nodes.

2.1 Advantages of Mobile Grid Services

Comparing with the standard grid services, Mobile Grid Services are characterized by the service mobility. The mobile agent part of the services has the ability to bring task execution to other hosts. It can help to balance the load on all grid nodes. Imagine that a service requires a lot of resources for its execution. If the service is implemented as a standard grid service, it will be blocked when the hosting machine runs out of resources. However, if the service is implemented as a Mobile Grid Service, it can move the execution to another host with plenty of resources. Moreover, the overload problem can be relieved due to the service migration. Even a large amount of service requests are present at the same time, the execution load can be distributed to other hosts and overloading on the hosting machine is avoided.

Besides the advantages above, Mobile Grid Services can also travel throughout the Grid intelligently according to resource needs with the help of the mobility. This improves the flexibility of Grid Services. If a standard grid service needs to access massive data in other nodes, continuous connections between Grid nodes are essential. However, this is less critical for Mobile Grid Services where they can move to the nodes storing the data and get data from the Grid nodes locally, carry out the execution in those nodes and finally bring the results back to their original hosts. This may also reduce the network traffic of the data transmission.

2.2 Components of Mobile Grid Services

For any Grid Service, the service interface, the service deployment descriptor and the service implementation are essential elements that define all the details of the service. In GT4, the service interface and the service deployment descriptor are described in Web Service Description Language (WSDL) [8] and Web Service Deployment Descriptor (WSDD) respectively. Service implementation is a Java class or a group of classes expressing the service task logic.

Since Mobile Grid Services are realized as Globus grid services with JADE mobile agent support, the WSDL and WSDD definitions (describing the service interface and the service deployment descriptor) of Mobile Grid Services are just similar to those normal Globus grid services (without mobility). However, the situation is different in the service implementation. Instead of using normal Java classes only, Mobile Grid Services implementation involves both Java classes and JADE mobile agents. Three main components in the implementation part of Mobile Grid Services are Agent Manager, Task Agent and Monitor Agent.

Agent Manager. It is not a JADE mobile agent but a normal Java class which is responsible for managing the agents used for its own service. It will be stationary on its original node. This feature avoids the redeployment of the service and keeps the original Globus architecture.

The main tasks of Agent Manager are to:

- Create the Monitor Agent and Task Agent
- Receive client requests (through normal grid service calls)
- Redirect client requests to the Task Agent and Monitor Agent through ACL messages
- Redirect results from the Task Agent to client
- Keep track of the Task Agent and Monitor Agent locations as well as their execution situation

Task Agent. It is a JADE mobile agent which is responsible for doing the actual service task of the Mobile Grid Service. If the auto-migrate function is enabled, each Task Agent will have one Monitor Agent working with it. Each service may have multiple Task Agents for different tasks.

The main tasks of Task Agent are to:

- Carry out the actual tasks of the application services
- Perform suitable actions after receiving commands in the form of ACL messages from Agent Manager and Monitor Agent

Monitor Agent. It is a JADE mobile agent which is responsible for monitoring the resource information of the grid nodes and make migration decisions. It may not be created if the service providers do not want their Task Agents to migrate without their order (i.e. the auto-migrate function is disabled). Service providers or clients can change the parameter settings in order to adjust the migration decision policy.

The main tasks of a Monitor Agent are to:

- Receive grid nodes resource information from the Resource Information Service
- Use its own logic to analyze the resource information
- Make the migration decision including “move or not” and “where to move”
- Send an ACL message to the corresponding Task Agent and order it to move on suitable situations
- Move along with the corresponding Task Agent

2.3 Resource Information Service

Resource information (configuration and usage) of grid nodes is essential in our framework for supporting automatic run-time service migration. In our framework, the information is provided by the Resource Information Service.

The raw resource data of the node machines including processor and memory information, CPU and memory usage and host data are collected from the Ganglia cluster monitoring system [9] via the Globus Index Service [10] regularly.

Besides Resource information, this service also gets all the Monitor Agents' details from the Agent Management System residing on the JADE main container. The collected information will not be redirected to Monitor Agents directly. The irrelevant information will be filtered out first. In our current implementation, we only focus on the CPU clock speed, last minute CPU usage, current available RAM size and available harddisk space.

In order to reduce the message size used for resource data transmission, the resource information will be further processed to calculate three values (CPU, RAM, HD) before transferring to the Monitor Agents.

Before presenting the calculation of the CPU, RAM and HD values, three maximum variables used in the Resource Information Service will be introduced first. These three variables are Max_CPU, Max_RAM and Max_HD which are the best possible/expected CPU ClockSpeed, RAM available and Harddisk space available in the whole grid respectively. These variables are configurable during the setup of the Resource Information Service and the grid managers can adjust suitable values for their own grid according to the hardware available and the expected service requirement. Table 1 shows the unit used in the three maximum variables and an example configuration in a grid:

Table 1. Example value of the three maximum variables in Resource Information Service

Variable Name	Example Value	Unit
Max_CPU	2800 (2.8 GHz)	Megahertz (MHz)
Max_RAM	1024 (1 GB)	Megabyte (MB)
Max_HD	1024 (1 GB)	Megabyte (MB)

Now, assume that Host A has a 3.0GHz CPU and the current CPU usage is 20%. Moverover, 499MB RAM is available now and there is 5000MB free space in the harddisk. In the Resource Information Service, the CPU, RAM and HD values of Host A can be calculated as follows:

CPU = (3000 * (1-20%)) / Max_CPU = 2400/ 2800 = 85.71%
RAM = 499 / Max_RAM = 499 / 1024 = 48.73%
HD = 5000 / Max_HD = 5000 / 1024 = 488% (chop to 100%)

In this case, the resource data of Host A (CPU=8571, RAM=4873, HD=10000) will be transferred to all Monitors Agents.

The calculated value must be between 0 and 10000. It represents the percentage of available CPU/RAM/HD compared with the best possible/expected resources.

After the calculation, the processed results will be sent to all Monitor Agents through ACL messages. The Monitor Agents for different Mobile Grid Services can then use the resource information to make their migration decisions.

2.4 Scenario of Mobile Grid Service Execution

Figure 1 shows the steps involved when a Mobile Grid Service executes in the framework. Assume that a Mobile Grid Service "Service A" is setup on Node A. Its Agent Manager listens to any user's requests. The steps are as follows:

1. User requests an operation in "Service A" (e.g. a complex computation).
2. Agent Manager of "Service A" receives the request and creates appropriate Task Agent and Monitor Agent (called X and Y respectively) on Node A.
3. Task Agent X starts its service task while Monitor Agent Y starts to receive any resource information from the Resource Information Service.
4. Resources in Node A are running out due to some reasons. Monitor Agent Y gets the information and decides to migrate to Node B which has plenty of resources. Then, it moves to Node B with Task Agent X.
5. Both agents arrive at Node B. Task Agent X resumes its work while Monitor Agent Y continues to receive information from the Resource Information Service.
6. Task Agent X completes the service task. Agent Manager gets the result of the operation and terminates the two agents.
7. The result of the operation is redirected to User from Agent Manager. In fact, User can ask for the execution situation of the service from the Agent Manager throughout the task execution.

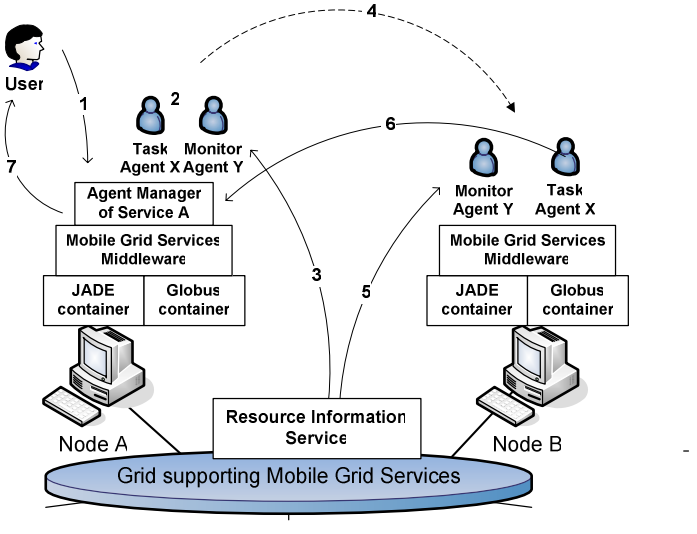


Fig. 1. Service migration

3 API Design

In this section, the issues which we have considered in the design of the MGS API will be presented.

In the framework illustrated above, each Mobile Grid Service is composed of the Agent Manager, Task Agent and Monitor Agent components. Obviously, the service implementation is more complex than the original one. The problem becomes more significant as mobile agent technology is employed. Mobile Grid Services will be discouraged if developers need to fully implement all these components.

In order to relieve this drawback, the MGS API is built with the aim of supporting easy Mobile Grid Services development. It is achieved by providing a collection of methods for Mobile Grid Services development. Complex operations for agent management and communication are hidden from service developers by wrapping them into some simple methods.

On the other hand, the API should be flexible enough for developers to implement their services without too many constraints. Though different services may have different requirements on their design, developers should be able to use the API to implement them. Therefore, maintaining high flexibility is another important issue for the API design. Many parts of the API are particularly designed for overcoming this issue. For example, the decision logic of the Monitor Agent can be configured with different parameters in order to utilize suitable migration strategies. Developers can even make use of self-defined logic by modifying the Monitor Agent in simple steps.

4 API Implementation

4.1 Overview

Figure 2 shows the architecture of the MGS API. The API is written in the Java programming language [11]. It is built on libraries of the Java Agent Development

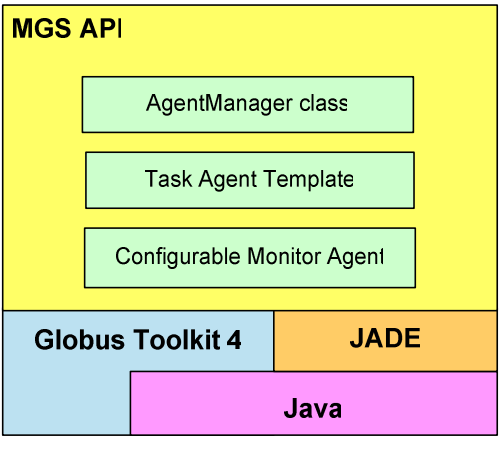


Fig. 2. Architecture of the MGS API

Framework (JADE) [7] for the implementation of mobile agent management and communication. At the same time, it will use some of the facilities in the Globus Toolkit 4 [3].

This API provides a collection of libraries for supporting the development of Mobile Grid Services. As mentioned in the framework overview, Mobile Grid Services are realized as special grid services which distribute the actual working tasks to mobile agents. The main components of each Mobile Grid Service including Agent Manager, Monitor Agent and Task Agent can be easily implemented with the help of the MGS API. Since the security support of the framework in the MGS API is still under construction, it will not be included in the following section.

Generally speaking, the API can be divided into three parts: AgentManager class, configurable Monitor Agent and Task Agent Template. AgentManager class takes the role of the Agent Manager component and provides methods for developers to manage the created agents. Configurable Monitor Agent is the default implementation of the Monitor Agent component and provides configurable options to meet various requirements. For the Task Agent component, developers must implement their own versions according to their requirements. Task Agent Template is provided in the API to help developers to do this job in a simpler way. Detailed descriptions of them are given in the followings.

4.2 Agent Manager Class

In a Mobile Grid Service, the Agent Manager component (non-agent part) basically acts as a relay between users and the mobile agents (i.e. Task Agents and Monitor Agents). This part of the Grid Service implementation can be simplified and partially accomplished by employing the *AgentManager* class in the MGS API.

In the Grid Service implementation, an *AgentManager* object should be created to enable the service to possess the essential functions of Agent Manager. These functions include managing agents used in the service, preparing the execution environment connected to the JADE main container and setting up communication channels for those agents.

During the startup of the *AgentManager* object, a JADE remote container (connecting to the JADE main container) is setup by using the JADE in-process interface. This interface allows external Java application to use JADE as a kind of library. That means it allows the launching of the JADE Runtime as well as the creation of agents within the application itself. The JADE container in the *AgentManager* object provides an executing environment for all newly created agents through the Agent Manager. Besides the container, an Agent Manager Agent is also created during the setup of *AgentManager* object. This agent is used to help the *AgentManager* object to send ACL messages to and receive messages from other agents in the JADE platform. It is stationary because it always resides at the same machine as the static Agent Manager and thus it is not necessary to move.

To preserve the autonomy of agents, the in-process interface of JADE is designed such that the application cannot obtain a direct reference to the agents and cannot perform method calls on the agents. That means the *AgentManager* object is unable to pass its commands or queries to the Task Agents through direct method calls on the Agent Manager Agent. Therefore, a special mechanism is taken to carry out the message exchange between Agent Manager and Agent Manager Agent.

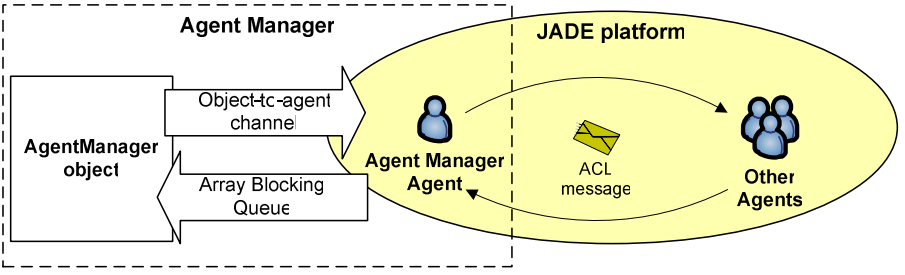


Fig. 3. Communication between AgentManager object and Agent Manager Agent

Figure 3 shows the communication between *AgentManager* object and Agent Manager Agent. For *AgentManager* object sending a message to Agent Manager Agent, the object-to-agent channel provided by JADE is used. This channel allows applications to pass objects to the agents they hold. In our implementation, Agent Manager Agent will enable this object-to-agent communication channel and continuously receive any objects from the *AgentManager* object. Once *AgentManager* object need to send an ACL message, it will pass the message as well as the reference of an ArrayBlockingQueue object to its Agent Manager Agent. The agent will then send the message to the target agent and wait for a reply. After receiving the reply, the message is needed to flow in the opposite direction. The Agent Manager Agent will put the reply message to the received ArrayBlockingQueue object. The *AgentManager* object will be notified that the queue is non-empty and get the reply ACL message. By this mechanism, *Agent Manager* can carry out the message exchange with Agent Manager Agent successfully and further communicate with all agents in the JADE platform.

On the other hand, the *AgentManager* class also provides methods to help service developers to implement agent management and interaction between users and services in their Mobile Grid Services. The main methods provided are:

void createAgent(String agentName, String agentClass, Object[] agentArgs)

This method is used to create a new Task Agent in the Mobile Grid Service. Since different tasks should be implemented in different Task Agents, this method receives an argument *agentClass* in order to allow developers to specify their required Task Agent being instantiated in their services. No Monitor Agent will be created by this method. It is designed for those services which do not require automatic service migration initiated by the Monitor Agent. The mobility of the Task Agent is preserved such that its execution still can move to other hosts if the route is preset.

void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, String monitorClass, Object[] monitorArgs)

Unlike the *createAgent* method above, this method will create the specified Task Agent as well as a Monitor Agent in the service. The argument *monitorClass* allows the developer to specify the Class used to instantiate the Monitor Agent. Another new argument *MonitorArgs* is used to pass arguments to the Monitor Agent.

void createAgentWithMonitor(String agentName, String agentClass, Object[] agentArgs, MonitorSetting setting)

This method is similar to the one above but the Class used to instantiate the Monitor Agent is fixed. It will use the default Monitor Agent provided in the MGS API compulsorily such that developers have no need to concern about the Monitor Agent class. The argument *MonitorSetting* is used to configure the default Monitor Agent. Details of the default Monitor Agent are shown in section 4.4.

ACLMessage createACLMessage(String agentName, int performative, String content)

It is used to create an ACL message for the message exchange between the Agent Manager and mobile agents. The arguments are used to specify the receiver, the content and the FIPA performative of the message (e.g. REQUEST, INFORM).

String sendACL(ACLMessage query)

It is used to send a command or a query to mobile agents through an ACL message from the Agent Manger. This method is important for implementing the interactions between service callers and Task Agents. After being created by the *createACLMessage* method, the ACL message can be sent to the target agent through this method and the reply will be returned. In the meantime, the ACL message is passed from the *AgentManager* object to the Agent Manager Agent while the reply message is forwarded in opposite direction (as shown in Figure 3). The actual sending and receiving procedures are hidden in this method. Service developers can concentrate on the design of ACL message exchange and the handling of the reply (i.e. return of *sendACL()* method).

MGSResult getCurrentResult(String taskName)

This method is used to get the current result of a Task Agent from the ResultTable maintained in the *AgentManager* object. The ResultTable is used in a special mechanism for the Task Agent sending back the execution result to Agent Manager. Since the execution time of a Task Agent may be very long, it is unreasonable if returning from the sendACL() method is the only way to get the result. As a result, Task Agent should have a mean to send the result back to Agent Manager anytime. To achieve it, a ResultTable is added to the *AgentManager* object. This table stores a number of *MGSResult* objects with task name as their keys. *MGSResult* is composed of a task result (in form of String object) and a flag "isFinal" indicating if the result is finalized. After executing to a certain stage, Task Agent can send the temporary result to its corresponding Agent Manager Agent. The Agent Manager Agent will then receive the result and update the data in the ResultTable. When the users want to get the current result of a task and invoke this method, the target *MGSResult* object obtained from the table will be returned.

MGSResult getFinalResult(String taskName)

This method is used to get the final result of a Task Agent from the ResultTable. It will call the getCurrentResult() method repeatedly until the obtained result is finalized. The status of the result can be identified by checking the "isFinal" flag of the returned MGSResult object. Finally, the final result of the task will be returned.

AgentSituation getAgentSituation(String agentName)

This method returns the current situation of a specific agent. The result is returned as an *AgentSituation* object containing information about agent state, position, etc. In fact, a special ACL message is sent to the agent to ask for the current situation.

void moveAgentTo(String agentName, String containerName, String containerAddress)

This method directly forces the agent named with the specified *agentName* to move to a specific host. The command is sent to the agent platform instead of the agent itself.

4.3 Task Agent Template

The Task Agent component is the most variable part in the Mobile Grid Services. As it is responsible for the actual service task, the implementation of the Task Agent is heavily dependent on the task properties of the Mobile Grid Services. It is impossible to have a single universal Task Agent which can fulfill all requirements of various types of service tasks. Therefore, the implementation of Task Agent must rely on the developers themselves. To help developers building application-specific Task Agents for their own Mobile Grid Services, a template is provided. The Task Agent Template consists of several incomplete classes which have already implemented most of the basic and essential elements in a Task Agent. To complete the Task Agent implementation, developers need to follow the guidelines in the template and complete all the classes such that they can carry out their application-specific tasks.

As stated in the implementation procedure of JADE agents, the developer who wants to implement an agent-specific task should define one or more Behaviour subclasses, instantiate them and add the Behaviour objects to the agent job list. As a result, the template is composed of three essential classes for the Task Agent:

- *TaskAgent.java*

It is the main body of the Task Agent. It extends the *Agent* class in the JADE library to inherit the ability to accomplish essential interactions with the agent platform (e.g. registration) and communication (through ACL message exchange) with other agents. It is also responsible for the initialization of the agent. On the other hand, two methods - *notifyResult()* and *notifyFinalResult()* are provided to update the ResultTable in the Agent Manager. They should be invoked when we want to notify the temporary or final result to the Agent Manager (usually after Task Agent finishes its task to certain stage). After that, a special message will be sent to the corresponding Agent Manager Agent and the ResultTable will be updated by the new result.

- *TaskServeIncomingMessagesBehaviour.java*

This Behaviour class is responsible to receive any incoming ACL messages and keeps reacting with all relevant messages. For instance, it will start the migration process and move to the target host marked in the message when it receives a “move” message from the Monitor Agent. Hence, all the message handlings should be defined in this class. To simplify the implementation, all the handling of pre-defined command messages such as “move” is already implemented in the template.

- *TaskBehaviour.java*

It is the class dealing with the actual service task. Developers should implement the program logic of the service tasks in the *action()* method in this class. The *action()* method will be invoked when the Task Agent starts. After *action()* return, the *done()* method will be called. The return value of this *done()* method will determine the life of the Behaviour class. If the value is true, the behaviour will end. Otherwise, the *action()* method will be invoked again. Therefore, the stopping criteria of the behaviour should be implemented in the *done()* method.

To develop a new Task Agent, a developer should implement each service task into the indicated location of the *TaskBehaviour* class. Different tasks should be implemented in different *TaskBehaviour* classes. Then, these behaviour classes should be added to the *TaskAgent* by following the guideline in the template. Although, the handling of pre-defined command messages are already implemented in the template, service developers still are responsible to implement the reaction of receiving their self-defined command messages from the Agent Manager or other message exchange with other agents.

4.4 Configurable Monitor Agent

A default Monitor Agent is provided in the API. It is used to accomplish the implementation of the Monitor Agent component in a Mobile Grid Service. The presence of this configurable Monitor Agent allows service developers to ignore the related implementation and thus simplifies the development process. It is composed of the following two classes:

- *MonitorAgent.java*

It is the main body of the Monitor Agent which stores resource information for migration decisions. A set of hash tables are created in the agent to store those resource data of each host including the host name, CPU value, RAM value and HD value calculated in the Resource Information service. It contains a *decide()* method which is responsible to analyze the resource information and make migration decisions. It will decide whether migration is necessary and where it should move to. The *decide()* method does not take in any argument. It uses the hosts and their information stored in the hash tables for migration decisions. It will return the chosen host name (if decide to move) or null (if not decide to move).

- *MonitorServeIncomingMessagesBehaviour.java*

It is the only behaviour added to the *MonitorAgent* class. It is responsible to handle the ACL messages received from the Resource Information Service. It will update the resource information stored in *MonitorAgent* and then call the *decide()* method in *MonitorAgent* at appropriate times. If the *decide()* method return a host name, it will send "move" message to the corresponding Task Agent and then move to the decided host together.

To make the default Monitor Agent useful under different situations (different developers' requirements), it is designed to be configurable. This goal is achieved by the *MonitorSetting* object. The constructor of *MonitorAgent* class will take in a *MonitorSetting* object as argument. The *MonitorSetting* object contains some configurable variables for controlling the execution mode (normal mode, debug mode and GUI

mode) of the Monitor Agent. In debug mode, the Monitor Agent will show detailed information about any incoming and outgoing messages as well as migration decisions to the screen for debug purposes. In GUI mode, a graphical user interface of the Monitor Agent will be used for displaying all the Resource information. In normal mode (same as non-debug and non-GUI mode), the Monitor Agent will keep silent (unless error occurs) to reduce unnecessary overhead and hide the existence of the Monitor Agent from the service/resource provider.

Besides the execution mode, the migration decision logic in the Monitor Agent (in the *decide()* method) can be configured by the variables in the *MonitorSetting* object. The service developers can set the minimum requirement on currently available CPU, memory and harddisk in the current host. Moreover, the importance weight on CPU, memory and harddisk used in the decision logic can be configured. If the current host meets the minimum requirements, the *decide()* method will return null. Otherwise, the *decide()* method will choose and return the best host according to the score calculated by following formula:

$$\text{Score} = (\text{CPU_weight} * \text{CPU}) + (\text{RAM_weight} * \text{RAM}) + (\text{HD_weight} * \text{HD})$$

However, the provided configurable options undoubtedly cannot fulfill all developers' requirements. For example, developers may require that certain hosts have extra bonus score in the migration decision. For any special requirements (like the above example) on the migration decision logic, service developers can also develop their own Monitor Agent by extending the provided one. The only work required is to override the *decide()* method. In this way, they can produce their own Monitor Agents which are best-fit to their requirements

4.5 Example Application

The code below shows an example of Mobile Grid Service developed on top of the MGS API. In the constructor of this Grid Service "MyApplication", an *AgentManager* object is created with the JADE main container's name and port as well as its application name in the whole grid. The *createAgentwithMonitor()* method of *AgentManager* object is then used to create a Task Agent named "task1" using the Java class in path "myPath.MyTaskAgent". At the same time, a Monitor Agent working with "task1" is created using the default Monitor Agent in path "mgs.MonitorAgent.MonitorAgent". The Monitor Agent is configured by a *MonitorSetting* object without changing any default setting.

Two methods are implemented for the client to interact with the service "MyApplication". In method1, we create and send "request_A" ACL message to the Task Agent "task1". After that, a reply will be received as the return of *sendACL()* method. Further handling on the reply can be done according to the service requirement. In method2, the *getAgentSituation()* method is called to get the current situation of the Task Agent "task1". An *AgentSituation* object will be received and the details will be printed out.

This example demonstrates the general techniques of implementing Mobile Grid Services using MGS API. The basic steps include the creation of *AgentManager* object, using it to create Task Agents and Monitor Agents, and use other provided methods to implement any interactions between users and agents.

Example of a Mobile Grid Service developed using MGS API

```
Public class MyApplication(){
  AgentManager agentManager;
  public MyApplication() {
    ...
    agentManger = new AentManager("mainHost", 1099,
             "MyApplication");
    MonitorSetting setting = new MonitorSetting();
    agentManager.createAgentWithMonitor("task1",
             "myPath.MyTaskAgent", null,
             "msg.MonitorAgent.MonitorAgent",setting);
  }
  public method1() {     //request_A to Task agent
    ACLMessage msg = agentMan-
             ager.createACLMessage("task1", ACLMes-
             sage.REQUEST, "request_A");
    String reply = agentManager.sendACL(msg);
    ... // handling of the reply
  }
  public method2() {    //get Task agent situation
    AgentSituation situation =
    agentManager.getAgentSituation("task1");
    System.out.println(situation.toString());
  }
}
```

5 Related Work

There exist a number of grid development toolkits; however they support static services only. Globus [3] is the most popular one among them. It offers good resources for grid application development. It provides software services and libraries for resource monitoring, resource discovery, resource management, security, file management, and fault detection. Nevertheless, it does not support service mobility. Up to now, none of the existing grid development toolkits support Mobile Grid Services.

In [13] and [14], the authors present the formal definition of Grid Mobile Service (GMS) and some critical factors in GMS implementation. GMS is a concept similar to the Mobile Grid Services presented in the previous sections. Both of them aim to enhance the static Grid Services with mobility by making use of Mobile Agent technology. However, the implementation details of the GMS project are limited and its approach tends to extend and modify the OGSA standards, the WSDL and the SOAP protocols to achieve the merging of mobile agents into the Grid Service. Comparatively, our design tends to maintain the Globus Grid service architecture (follows the OGSA standards) and make the framework more conformable to the existing and future Globus-based software.

Poggi [15] presents another approach to realize a development framework for agent grid system. This new approach is to extend agent-based middleware to support grid features instead of adding mobile agent support to grid middleware. In the paper, the authors present an extension of the JADE framework by introducing two new types of agent supporting intelligent movement of tasks as well as intelligent task composition

and generation. This provides the basis for the realization of flexible agent-based grid system. The project is still at an early stage and no framework is built yet.

Though mobility study in Grid Services is still a relatively young research area, a similar idea about migrating services has already been explored in the research area of Mobile Agent technology for years. Some projects use mobile agents to provide, share and access distributed resources in computational grids [4, 5, 6]. These mobile agent researches use different approaches to realize the computing job migration which is comparable to the service migration of Mobile Grid Services. However, all of them use their own developed grid infrastructures. Those infrastructures are simple and provide less functionality than those provided by well developed grid toolkits such as Globus [3].

6 Conclusions and Future Work

In this paper, we have briefly described the Mobile Grid Service which is an extension of the original stationary Grid Service. Moreover, a middleware framework of Mobile Grid Services constructed by combining the Java Agent Development Framework (JADE) [7] and the Globus Toolkit [3] is introduced. To support easy service development in the proposed framework, the MGS API is designed and implemented. The design issues as well as the implementation of the API are presented.

In the future, security mechanisms including authentication, authorization, message signature and encryption as well as the protection of mobile agents against malicious hosts will be added to the MGS API. It will provide extra facilities for developers to develop secure Mobile Grid Services. On the other hand, the presence of mobile agents in the Grid Services undoubtedly introduces overheads. The runtime and migration overheads of mobile agents may be significant. We will carry out analysis and simulation to find out the influence of those overheads on the performance of the Mobile Grid Services.

Acknowledgments. The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project no. CUHK4220/04E).

References

1. Ian Foster, Carl Kesselman, and Steve Tuecke, “The Anatomy of the Grid, Enabling Scalable Virtual Organizations”, International Journal of Supercomputer Applications, 2001.
2. Ian Foster and C. Kesselman, editors, “The Grid2: Blueprint for a New Computing Infrastructure”, Morgan Kaufmann, 2004.
3. Globus toolkit, http://www.globus.org/
4. M. Fukuda, Y. Tanaka, N. Suzuki, Lubomir F. Bic and S. Kobayashi, “A mobile-agent-based PC grid”, In proceedings of Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS'03), 2003, Seattle, Washington.
5. Rodrigo M. Barbosa and Alfredo Goldman, “MobiGrid - Framework for Mobile Agents on Computer Grid Environments”, MATA 2004, LNCS 3284, pp. 147-157, 2004.

6. Niranjan Suri, Paul T. Groth, Jeffrey M. Bradshaw, "While You're Away: A System for Load-Balancing and Resource Sharing Based on Mobile Agents", In proceedings of the 1st International Symposium on Cluster Computing and the Grid, p.470, May 15-18, 2001.
7. Java Agent Development framework (JADE), http://jade.tilab.com/
8. Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana, "Web Services description language (WSDL) version 1.2", http://www.w3.org/TR/2003/WD-wsdl12-20030303/
9. Ganglia monitoring system, http://ganglia.sourceforge.net
10. Globus Index Service, http://www.globus.org/toolkit/docs/4.0/info/index/
11. Java, http://java.sun.com
12. Sze-Wing Wong and Kam-Wing Ng, "A Middleware Framework for Secure Mobile Grid Services", in International Workshop on Agent based Grid Computing at 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2006), May 16-19, 2006
13. Shang-Fen Guo, Wei Zhang, Dan Ma and Wen-Li Zhang, "Grid Mobile Service: Using Mobile Software Agents in Grid Mobile Service", In proceedings of the Third International Conference on Machine Learning and Cybernetics, Shanghai, 2004.
14. Wei Zhang, Jun Zhang, Dan Ma, Benli Wang, Yuntao Chen. "Key Technique Research on Grid Mobile Service", In proceedings of the 2nd International Conference on Information Technology for Application (ICITA 2004).
15. Agostino Poggi, Michele Tomaiuolo and Paola Turci, "Extending JADE for Agent Grid Application", In proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'04).
16. The Foundation for Intelligent Physical Agents (FIPA), http://www.fipa.org

Using Classification Techniques to Improve Replica Selection in Data Grid*

Hai Jin, Jin Huang, Xia Xie, and Qin Zhang

Cluster and Grid Computing Lab
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

Abstract. Data grid is developed to facilitate sharing data and resources located in different parts of the world. The major barrier to support fast data access in a data grid is the high latency of wide area networks and the Internet. Data replication is adopted to improve data access performance. When different sites hold replicas, there are significant benefits while selecting the best replica. In this paper, we propose a new replica selection strategy based on classification techniques. In this strategy the replica selection problem is regarded as a classification problem. The data transfer history is utilized to help predicting the best site holding the replica. The adoption of the switch mechanism of replica selection model avoids a waste of time for inaccurate classification results. In this paper, we study and simulate KNN and SVM methods for different file access patterns and compare results with the traditional replica catalog model. The results show that our replica selection model outperforms the traditional one for certain file access requests.

1 Introduction

A data grid connects a collection of hundreds of geographically distributed computers and storage resources located in different parts of the world to facilitate sharing data and resources [1, 2]. The size of the data need to be accessed in the data grid is in the order of terabytes today and is soon expected to reach petabytes. Ensuring efficient access to such huge and widely distributed data is a challenge to network and grid designers. The major barrier supporting fast data access in a data grid is the high latency of wide area networks and the Internet, which impacts scalability and fault tolerance of the total grid system. Data replication is adopted to improve data access performance. The idea of replication is to store copies of data in different locations so that data can be easily recovered if one copy at one location is lost. Moreover, if data can be kept closer to user via replication, data access performance can be improved dramatically. Replication is a solution to many grid-based applications such as climate data analysis and the *Grid Physics Network* [3] which requires responsive navigation and manipulation of large-scale datasets.

Replica management service is responsible for managing complete and partial copies of data sets. It is an important issue for a number of applications. Its functions

* This paper is supported by National Science Foundation of China under grant No.90412010 and ChinaGrid project from Ministry of Education of China.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1376 – 1387, 2006.
© Springer-Verlag Berlin Heidelberg 2006

include creating new copies of a complete or partial data set, registering these new copies in a replica catalog, allowing users and applications to query the catalog to find all existing copies of a particular file or collection of files, and selecting the best replica for access based on storage and network performance predictions. The replica selection problem can be divided into two sub-problems: discovering the physical locations of a file given a logical file name, and selecting the best replica from a set based on some selection criteria.

To select the best one when different sites hold replicas, an optimization technique [4] is described to consider both disk throughput and network latencies while selecting the best replica. The data transfer history can help predicting the best site holding replica. *K-Nearest Neighbors* (KNN) rule is used as one such predictive technique. In this technique, when a new request arrives for the best replica, it looks at all previous data to find a subset of previous file requests that are similar to it and uses them to predict the best site holding the replica. For this optimization technique, there are the following aspects that should be further taken into account.

KNN method only considers the obvious classification factors, such as file index and access time, and a finite number of history data. It cannot reflect complicated factors such as more cases of previous data access for a file and its neighbors. That is, its decision method is very simple.

KNN classifier is instance-based or lazy learner in which it stores all of the training data and does not build a classifier until new data needs to be classified. Lazy learner can incur expensive computational costs when the number of potential neighbors to be compared with given new data is great.

In KNN, the resource node always contacts the site found by the classifier and requests the file, regardless of the accuracy of classification result. The switch from classification method to traditional one is not considered when the classification result is far from ideal. So, the system performance will decrease for the inaccurate file access. In order to solve these problems, we use the other classification technique to improve the accuracy of classification and reduce the costs of execution time. Meanwhile, the process of replica selection is improved by using the switch mechanism and the performance loss caused by the inaccurate classification is decreased. In this paper, we adopt *Support Vector Machines* (SVM) method to select the best replica and compare its execution performance with KNN and traditional methods. As introducing more decision factors and efficient classification method, the new replica selection strategy shows better performance.

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 discusses two classification methods used in replica selection. Section 4 describes the improved replica selection model. Section 5 presents the simulation configuration and an evaluation and comparison of our technique with the tradition replica catalog model. In Section 6, we conclude and give directions for future work.

2 Related Work

Replication is an essential requirement for distributed environments. In this field, there are several research issues that have been discussed.

Feng et al. [5] adopt a new grid data transferring tool rFTP to increase performance further by exploiting replica-level parallelism in grids. rFTP improves the data

transfer rate and reliability on grids by utilizing multiple replica sources concurrently. By using multiple replicas together, rFTP avoids replica selection and dramatically increases performance of the aggregate transfer rates of all replicas.

Kavitha and Foster [6] discuss various replication strategies. All these replication strategies are tested on hierarchical grid architecture. The replication algorithms proposed are based on the assumption that popular files of one site will be popular at another site. The client will count the number of hops for each site that holds replicas. According to the model, the best replica will be the one with the least number of hops from the requesting client.

To determine which replica can be accessed most efficiently in a data grid, many factors should be considered including physical characteristics of the resources and the load behavior on the CPUs, networks, and storage devices. Vazhkudai et al. [7] develop a predictive framework to resolve the file replica selection problem, which combines the integrated instrumentation that collects information about the end-to-end performance of previous transfers, the predictors to estimate future transfer times, and a data delivery infrastructure providing users with access to both the raw data and predictions.

As users requests vary continuously, the system needs a dynamic replication strategy that adapts to users' dynamic behavior. To address such issues, Rahman et al. [8] present and evaluate the performance of six dynamic replication strategies for two different access patterns. Their replication strategies are mainly based on utility and risk. Before placing a replica at a site, system calculates an expected utility and risk index for each site by considering current network load and user requests. A replication site is then chosen by optimizing expected utility or risk indexes.

Other works on replica management reported include: Chervenak et al. [9] provide an architectural framework and define the role of each component. Allcock et al. [1] develop a replica management service using the building blocks of Globus Toolkit. The Replica Management infrastructure includes Replica Catalog and Replica Management Services for managing multiple copies of shared data sets. Zhao et al. [10] develop a grid replica selection service based on *Open Grid Service Architecture* (OGSA) which facilitates the discovery and incorporation of the service by other grid components. Cai et al. [11] propose a fault tolerant and scalable *Peer-to-Peer Replica Location Service* (P-RLS) applicable to a data grid environment.

3 Classification Techniques in Replica Selection

In a data grid, the users register files to a replica catalog using a logical filename or collection. The replica catalog provides mappings from logical names of files to the storage system location of one or more replicas of these objects. In the two-stage process of selecting a replica, the traditional model may incur expensive time costs. First, the requesting site must wait while all requests in the replica location queue are processed for the centralized replica location service. This may cause substantial delays if there are a lot of pending requests. Second, when many sites hold the replica, the requesting site must find the best one by probing the network link and the current status of disk storage. So, we analyze the file transfer history to obtain the best storage site holding a replica, and the time can be saved. The history sometimes provides some useful information such as the access frequency of a file, the access pattern between different file requests. If the requested file or its nearby file has been

accessed recently, it may be accessed from the same site again with a significant probability while considering spatial and temporal locality of data access. Therefore, the resource node can directly request the file from the previous requested storage site rather than getting information from the replica location service every time.

In our discussion, the replica selection is regarded as finding the replica that has the best transfer performance. From data mining point of view, this may be considered as a classification problem. The classification problem is stated as: Given the training data $D=\{t_1, t_2,\ldots, t_n\}$ of data tuple and a set of classes $C=\{c_1, c_2,\ldots, c_m\}$, the classification problem is to define a mapping f: $D\rightarrow C$ where each t_i is assigned to one class. A class, c_j, contains precisely those tuples mapped to it; that is, $c_j=\{t_i \mid f(t_i)=c_j,\ 1\le i\le n,\ t_i\in D\}$. So, the classification is viewed as a mapping from the data tuples to the set of classes. In practice, the problem usually is implemented in a two-step process: (a) Learning: training data are analyzed by a classification algorithm. The learned model or classifier is represented in the form of classification rules. (b) Classification: the model can be used to classify future data tuples or objects for which the class label is not known.

For the convenience of comparison, we detail two classification methods including KNN [4] and SVM, and apply them to the replica selection. As the different features of these methods, they have the different accuracy and complexity of implementations. The execution performance of these methods will be compared with one of the traditional replica catalog method.

3.1 K-Nearest Neighbors Method

One common classification scheme based on the use of distance measurement is *K-Nearest Neighbors* [12, 13]. KNN technique assumes that the entire training set includes not only the data in the set but also the desired classification for each item. In effect, the training data become the model. When a classification is to be made for a new item, its distance to each item in the training set must be determined. Only the K closest items in the training set are considered further. The new item is then placed in the class that contains the most items from this set of K closest items. The basic steps of the algorithm include: computing the distances between the new item and all previous items, already classified into clusters; sorting the distances in increasing order and selecting K items with the smallest distance values; and applying the voting principle. A new item will be classified to the largest cluster out of K selected items.

The key to KNN method is the definition of distance between the items and the selection of K value. In our replica selection model, each resource node maintains a history of transfer when a file has been requested and transported to that node. The history includes the timestamp when the file transfer ends, the file index of the logical file and the storage site from which the file has been transferred. We use these attributes in history log to define a 3-tuple:

$$<timestamp, file\ index, storage\ site>$$

where *storage site* is regarded as the class label. Let T_1 and T_2 be two tuples $<t_1, f_1, s_1>$ and $<t_2, f_2, s_2>$. The distance between T_1 and T_2 is presented as follows:

$$d(T_1,T_2)=\sqrt{w_1(t_1-t_2)^2+w_2(f_1-f_2)^2} \tag{1}$$

where w_i represents the weight assigned to each attribute. Because this distance metric considers the spatial and temporal factors of file access, timestamp and file index distance measures will definitely improve the classification rate.

3.2 Support Vector Machines Method

Support Vector Machine is a novel learning machine based on the Structural Risk Minimization principle from computational learning theory [14]. SVMs are excellent examples of supervised learning that tries to maximize the generalization by maximizing the margin and also supports nonlinear separation using advanced kernels, by which SVMs try to avoid overfitting and underfitting. The margin in SVMs denotes the distance from the boundary to the closest data in the feature space. A simple description of the SVM algorithm is provided here, for more details please refer to [15]. The underlying theme of the class of supervised learning methods is to learn from observations. There is an input space, denoted by X, an output space, denoted by Y, and a training set $S=((x_1, y_1), (x_2, y_2),\ldots, (x_l, y_l)) \subseteq (X\times Y)^l$, l is the size of the training set. The overall assumption for learning is the existence of a hidden function $Y=f(X)$, and the task of classification is to construct a heuristic function $h(X)$, such that $h\rightarrow f$ on the prediction of Y. The nature of the output space Y decides the learning type. SVM belongs to the type of maximal margin classifier, in which the problem of computing a margin maximized boundary function is specified by the following quadratic programming problem:

$$\min_{\alpha} \ \frac{1}{2}\sum_{i=1}^{l}\sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j K(x_i, x_j) - \sum_{j=1}^{l} \alpha_j$$

$$s.t. \ \sum_{i=1}^{l} y_i \alpha_i = 0 \qquad (0\le\alpha_i\le C,\ i=1,2,\ldots,l) \tag{2}$$

where α is a vector of l variables, in which each component α_i corresponds to a training data (x_i, y_i). C is the soft margin parameter controlling the influence of the outliers in training data. The algorithm based on support vector classification is described as follows:

(1) Let a training set $S=((x_1, y_1), (x_2, y_2),\ldots, (x_l, y_l)) \subseteq (X\times Y)^l$, where $x_i\in X = R^n$, $y_i\in Y = \{1, -1\}$, $i=1, 2,\ldots, l$, l is the size of the training set;
(2) Select a proper kernel function $K(x, x')$ and parameter C to construct and solve quadratic programming problem described above, and its optimal solution is $\alpha^*=(\alpha_1^*, \alpha_2^*, \ldots, \alpha_l^*)^T$;
(3) Select a positive component of α^* $(0<\alpha_j^*<C)$, and calculate $b^* = y_j - \sum_{i=1}^{l} y_i \alpha_i^* K(x_i, x_j)$;
(4) Construct the decision function $f(x) = \mathrm{sgn}(\sum_{i=1}^{l} \alpha_i^* y_i K(x, x_i) + b^*)$. If $f(x)=1$, new item is classified to positive class; otherwise to negative class.

For a multiple class classification problem in our study, we adopt one class against remaining classes method and construct a series of binary classification machines to classify. Based on the similar consideration as KNN method, we utilize the same 3-tuple as pattern of training set. It is reasonable to expect that the distance between these vectors provides a measure of similarity between corresponding tuples. Therefore a kernel can then be created using a Gaussian based on 2-norm:

$$K(x,x') = \exp(-\frac{\left\|x-x'\right\|_2^2}{2\sigma^2}) \tag{3}$$

Here, the definition of a measure of similarity is specified. Let the vectors x and x' be denoted by tuples <t, f, s> and <t', f', s'>, the distance measure, 2-norm, between x and x' is presented as follow:

$$d(x,x') = \begin{cases} \sqrt{w_1(t-t')^2 + w_2(f-f')^2}, & |t-t'| < d_t \\ \sqrt{w_1 d_t^2 + w_2(f-f')^2}, & |t-t'| \geq d_t \end{cases} \tag{4}$$

where w_i represents the weight assigned to each attribute, and d_t is a threshold specified that limits the difference between timestamp components of two vectors. The reason for this definition is to avoid too big distance measure caused by timestamp component. Considering the facts of replica selection, whether a file is again accessed successfully in the certain site cannot be determined markedly by an interval of time between two accesses. In other words, the time factor is useful rather than decisive.

The significant difference between KNN and SVM methods is that KNN regards explicitly the contiguity of file index and timestamp as the classification evidence, and the association form between input attributes and classification outcome has been fixed; whereas SVM method has the capability of learning. It can not only construct the association form between input attributes and classification outcome, but also adjust one automatically by learning more updating training data. Besides, SVM method constructs a generalization model before receiving new tuples to classify. Therefore, SVM method makes the classification model better suitable for some mutable cases.

4 Replica Selection Model

In our model, the replica selection is done from file transfer history stored on each resource node rather than consulting the replica catalog. But for any file request, the resource node always contacts the site found by the classifier and requests the file, regardless of the accuracy of classification result. It is very possible that the file cannot be found in storage site specified by an inaccurate classification result. Then, the traditional replica selection method is still needed. The resource node contacts replica catalog to get replica list information and selects the best one for file access. It is quite obvious that an inaccurate classification will lead to more time costs. So, the switch mechanism from classification model to traditional model should be considered. Here we define the discrepancy measure between the files as the difference between their indexes. The relationship between a file and a storage site is

determined by the minimum discrepancy value between the requested file and each file that has been accessed in that storage site. If the minimum is less than or equal to the predefined switch threshold d_f, it shows this file and this site are related and can utilize classification result to guide file access. Otherwise we apply directly the traditional replica catalog model rather than contact the site found by the classifier.

The execution flow of a resource node is illustrated in Fig. 1. First, if the node has sufficient history and has built the classifier, it finds the best site using the classifier. After obtaining the classification result, the node selects all the transfer logs about this site and calculates the discrepancy values between the requested file and each one that has been transferred in the site. When the minimal discrepancy value is less than or equal to the predefined threshold, the node contacts the storage site and requests the file; otherwise it contacts the replica catalog directly and selects the best file replica. If the site found by the classifier does not hold the requested file because the file has been deleted or the classification result is inaccurate, the node contacts the replica catalog to find other available replicas for that file. When file transfer ends, this transfer information is recorded and becomes the updating training data.

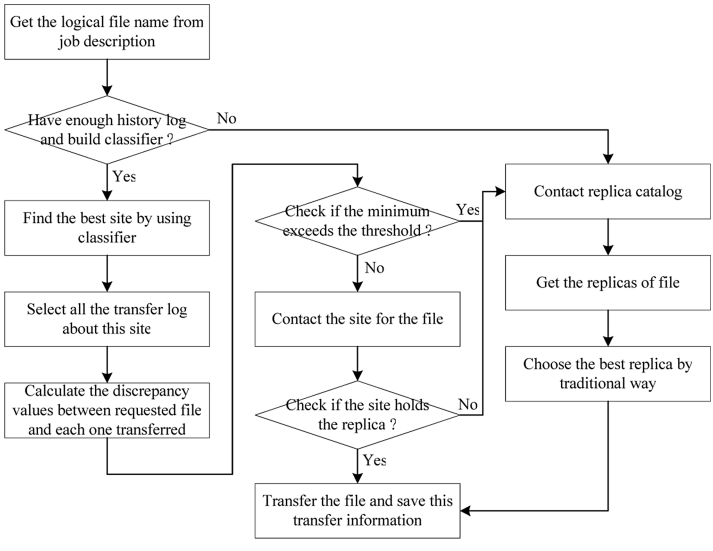


Fig. 1. Execution flow of a resource node

5 Performance Evaluation

To evaluate our approach we use a data grid simulation infrastructure called GridSim.datagrid, as a latest extension to GridSim [16, 17], written in Java. It is a data grid simulator extended with the ability to handle the replication of data to several sites, query for location of the replicated data, access to the replicated data, and make complex queries about data attributes. With these features, it is allowed to do integrated studies of on demand replication strategies with job scheduling to available resources.

5.1 Simulation Configuration

GridSim is designed by assuming that a grid consists of a number of entities including resources, users, brokers, information service and replica catalog, network components and so on. Each resource can have one or more different computing and storage elements, and enables users to run their jobs as well as to gain access to available data sets. A user can be modeled as an application performing on behalf of the user. A broker acts as a meta-scheduler that controls job scheduling to different resource entities. An information and replica catalog service is an entity that provides resource registration, indexing, discovery services and files and their replicas managements. The network components allow GridSim entities to be connected using links and routers, with different packet scheduling policies for realistic experiments.

During execution of user jobs, a job will typically request a set of logical file names for data access. The order in which the files are requested is determined by the access pattern. In this simulation, we consider two typical access patterns: unitary random and random [18]. The unitary random pattern means that the set of logical file names is ordered and successive file requests are exactly one element away from the previous file request but the direction is random, and random pattern describes that files are selected randomly from a set with a flat probability distribution. When the direction of file requests is fixed, unitary random access pattern will become the simple sequential access pattern. These file access patterns are shown in Fig. 2.

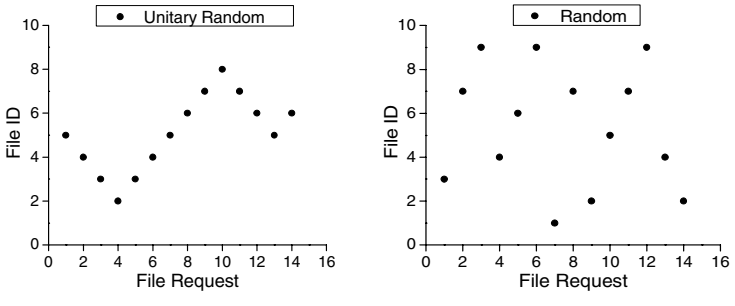


Fig. 2. Various file access patterns

In this experiment, when a resource node accesses a file, it creates a replica for that file. When the local disk space is insufficient, LRU rule is used for file replacement. That is, each resource node stores a replica of the file. If the resource node has sufficient space, the file will be stored there; otherwise the resource node makes space for the newest file by deleting the least recently used file. In the traditional model, the file access is done as follows: for each file required of the job, the resource node consults with the replica catalog to find all the replicas of this file, and then calculates

the time to access each replica. The resource node selects replica node that has the minimal access cost. In the classification model, the replica selection is done from of file transfer history stored on each node rather than consulting the replica catalog. When classification result exceeds the predefined threshold, the replica selection pattern will switch to the traditional model. Moreover, a resource node also gets help from replica catalog when the file is not found in the requested site due to the inexistence of file.

In this experiment, 22 resource nodes are configured. In order to analyze the job execution, we submit lots of jobs to six nodes, *A* to *F*. These jobs all request a set of files for data access. The simulation experiments are run for 500 jobs of various types. The numbers of the files submitted by the jobs to nodes A~F request all exceed 200. We choose the parameter w_1=1, w_2=5, and K=5 in KNN method, and σ=1.0, C=10, d_l=20 and switch threshold d_f=50 in SVM method.

5.2 Simulation Results

Fig. 3 shows the performance of two classification models and the traditional model with respect to total job time. For the unitary random access pattern, the classification models outperform the traditional one. A job will request files one after another and the node that runs that job creates replicas of all files needed to run the job. So, a file and its neighbors are held in a resource node. When a job requests the files with unitary random access pattern, the classification models will find a recent transfer history of neighbors of a requested file. In this sense, when there are enough file accesses with unitary random pattern in the system, applying the classification model can always improve the performance of file access. However, we do not see much improvement in the random file access pattern. This is because for this pattern files are accessed without any relation of previous requests. For further study, in two classification models, SVM is of the better performance. The training process of SVM can be executed in idle time, and only classification process is done when a new tuple comes. When the number of training data is large, SVM also works well. As KNN method does not build a classifier ahead of time, its cost is expensive for computation of a new tuple. So, there is a critical number for training data in KNN method. The data access performance will deteriorate when the number of training data is in excess of the critical number. In the random access pattern, two classification models do not gain good performance because many useless file accesses lead to a waster of time due to a lot of inaccurate classification results.

Fig. 4 shows the comparison of total job time between two classification models with and without switch function. For two access patterns, the classification methods show that the model with switch function outperforms the model without one. This is because the adoption of control mechanism for relationship between a file and a storage site avoids wasting access time. Moreover, it may not be the best one even though the classification result is valid. So, through the switch from classification model to traditional one, the resource node can obtain the best file transfer which provides the updating training data for future classifications.

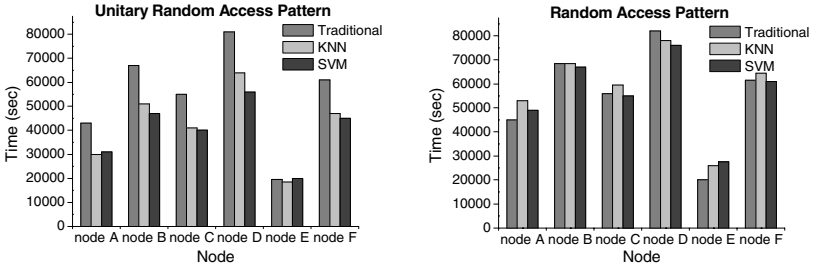


Fig. 3. Comparison of total job time for different models

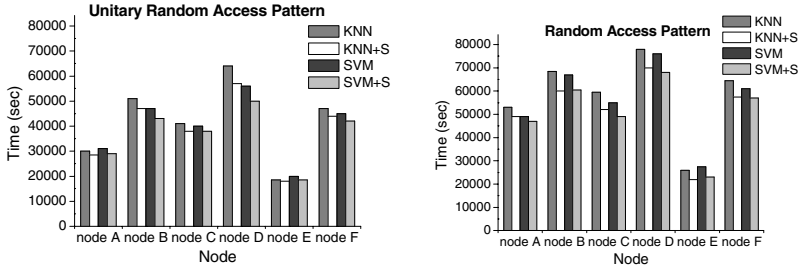


Fig. 4. Comparison of total job time for models with and without switch function

Table 1 presents the classification accuracy of two classification methods and the times of switching to the traditional model for the different file access patterns.

Table 1. Classification and switch rate for different file access patterns

Method	*Node*	*Unitary random*			*Random*		
		Classification		Traditional	Classification		Traditional
		Valid	Invalid		Valid	Invalid	
KNN	A	252	20	21	146	57	90
	B	366	35	19	174	79	167
	C	298	17	29	120	67	157
	D	430	46	37	189	86	238
	E	168	23	18	87	28	94
	F	349	11	21	123	103	155
SVM	A	269	13	11	180	39	74
	B	381	27	12	205	112	103
	C	305	15	24	218	46	80
	D	464	29	20	242	119	152
	E	183	18	8	128	15	66
	F	363	9	9	270	35	76

If the relation measure of classification result does not exceed the threshold, file access is regarded as classification method; otherwise it is regarded as traditional

method. In the classification method, if the site found by the classifier holds the requested file, the classification result is regarded as valid; otherwise invalid.

As a whole, the unitary random access pattern has more valid classifications and less model switches than random access pattern. From the different classification methods, SVM has the better performance. In the unitary random access pattern, the rates of valid classification for KNN and SVM methods are individually 86% and 91%. But in the random access pattern, they rapidly deteriorate to 39% and 59%. It is obvious that the number of switching to the traditional model is increasing remarkably. For file access without any relation between the files, the time cost caused by the inaccurate classifications will be saved by utilizing the traditional model directly.

6 Conclusion

In this paper, we present a model that utilizes the classification techniques, including KNN and SVM methods, to select the best replica for file access. To improve the accuracy of replica selection, we have considered the efficient classification factors and method. The SVM method based on learning can not only construct the association form between input attributes and classification outcome, but also adjust one automatically by learning more updating training data. The adoption of the switch mechanism of replica selection model avoids a waste of time for inaccurate classification results. We study different file access patterns on the performance of the classification model and compare our results with traditional model. The classification model shows a significant performance improvement for unitary random file access pattern, and SVM method is better than KNN.

In our future work, more decision attributes will be considered in SVM method. More related classification factors can reflect the system state as completely as possible and improve the accuracy of replica selection. Furthermore, we expect to use the adaptive parameters in the classification methods and adjust the classifier dynamically to get the best classification results.

References

1. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S.: Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In: *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and Technologies* (2001) 13–28
2. Stockinger, H., Samar, A., Holtman, K., Allcock, B., Foster, I., and Tierney, B.: File and Object Replication in Data Grids. In: *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing* (2001) 76–86
3. The GriPhyN Project, http://www.griphyn.org/
4. Rahman, R., Barker, K., and Alhajj, R.: Replica Selection in Grid Environment: A Data-Mining Approach. In: *Proceedings of the 2005 ACM Symposium on Applied Computing* (2005) 695–700
5. Feng, J. and Humphrey, M.: Eliminating Replica Selection - Using Multiple Replicas to Accelerate Data Transfer on Grids. In: *Proceedings of the Tenth International Conference on Parallel and Distributed Systems* (2004) 359–366

6. Kavitha, R. and Foster, I.: Design and Evaluation of Dynamic Replication Strategies for High Performance Data Grids. In: *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics* (2001) 106–118
7. Vazhkudai, S., Schopf, J., and Foster, I.: Predicting the Performance of Wide Area Data Transfers. In: *Proceedings of the 16th International Parallel and Distributed Processing Symposium* (2002) 34–43
8. Rahman, R., Barker, K., and Alhajj, R.: Replica Placement in Data Grid: Considering Utility and Risk. In: *Proceedings of the International Conference on Information Technology: Coding and Computing* (2005) 354–359
9. Chervenak, A., Deelman, E., Foster, I., Guy, L., Hoschek, W., Iamnitchi, A., Kesselman, C., Kunszt, P., Ripeanu, M., Schwartzkopf, B., Stockinger, H., Stockinger, K., and Tierney, B.: Giggle: A Framework for Constructing Scalable Replica Location Services. In: *Proceedings of the ACM/IEEE conference on Supercomputing* (2002) 1–17
10. Zhao, Y. and Hu, Y.: Gress - A Grid Replica Selection Service. In: *Proceedings of ISCA 16th International Conference on Parallel and Distributed Computing Systems* (2003)
11. Cai, M., Chervenak, A., and Frank, M.: A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table. In: *Proceedings of the ACM/IEEE Conference on Supercomputing* (2004) 56–56
12. Aha, D., Kibler, D., and Albert, M.: Instance-Based Learning Algorithms. *Machine Learning* (1991) 6(1): 37–66
13. Han, J. and Kamber, M.: *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc. (2001)
14. Cristianini, N. and Shawe-Taylor, J.: *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, Cambridge, England (2000)
15. Muller, K., Mika, S., Ratsch, G., Tsuda, K., and Scholkopf, B.: An Introduction to Kernel-based Learning Algorithms. *IEEE Transactions on Neural Networks* (2001) 12(2): 181–201
16. Sulistio, A., Poduvaly, G., Buyya, R., and Tham, C.: Constructing a Grid Simulation with Differentiated Network Service Using GridSim. In: *Proceedings of the 6th International Conference on Internet Computing* (2005) 437–444
17. The GridSim Project, http://www.gridbus.org/gridsim/
18. Bell, W., Cameron, D., Capozza, L., Millar, A., Stockinger, K., and Zini, F.: OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. *Int. J. of High Performance Computing Applications* (2003) 17(4): 403–416

Searching Moving Objects in a Spatio-temporal Distributed Database Servers System

Mauricio Marín[1], Andrea Rodríguez[2], Tonio Fincke[3], and Carlos Román[2]

[1] Center for Quaternary Studies, CEQUA, Universidad de Magallanes, Chile
mauricio.marin@umag.cl
[2] Universidad de Concepción, Chile
{andrea, caroman}@udec.cl
[3] University of Münster, Germany
fincke@uni-muenster.de

Abstract. Querying about the time-varying locations of moving objects is particularly cumbersome in environments composed of a very large number of distributed spatio-temporal database servers. In particular, searching for a specific object can require to visit each server. In this paper we propose a strategy to avoid such an exhaustive search that is based on the use of a centralized index, called *meta-index*, which is the entry point for spatio-temporal search queries. This index allows a software agent to determine a search plan for visiting the most likely servers to contain the target object. An important issue for large and dynamic distributed servers systems is to keep the meta-index as up-to-date as possible with the real system. This paper defines and compares two different strategies for maintaining properly updated the meta-index: *crawling*, where the centralized system that keeps the index controls itself the updating process, and *harvesting*, where each distributed database server autonomously transfers data directly into the central index system. Both strategies were implemented and compared by using discrete-event simulators with demanding synthetic spatio-temporal data. The results show that crawling has better performance.

1 Introduction

Systems with dynamic data often need to collect and store data in multiple disperse servers, while handling the whole or partial data access from a centralized location. Centralized access strategies maintain copies or summary of data at a central location using *replicas* [8] [7] [5] or *indexes* [4].

This paper presents a search strategy for moving objects located in a large number of distributed database servers and explores different strategies for updating centralized data. The aim of this work is to process queries such as *where was an object X within a time interval* $[t_i, t_j]$*?* To answer this query, this work proposes to maintain centralized data of the type of objects (e.g., car, truck, train, bus, and so on) that were located within predefined space partitions during time.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1388–1401, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The centralized data are organized in a meta-index that contains the *number of objects per type* in each server at different time instants. Thus, the meta-index does not store the actual location of individual objects over space, which could be impractical for applications with high dynamism and very large number of moving objects. Handling centralized data about aggregated data of moving objects has also been addressed in the context of spatio-temporal data warehouses [12]. These studies state that in very large data sets, one is often interested in knowing information about groups of objects instead of particular object trajectories.

The goal of the centralized *meta-index* is to allow the fast determination of the server that contains the object in the near past to the temporal interval of a query. The search starts from this server onwards by setting a software agent to jump from one server to the another until the agent gets all locations of the target moving object during the time interval of interest. To find the starting server, data in the meta-index is used to rank servers based on their likelihood of containing desired data. This ranking of the servers depends on the particular query, because, for a given type of object at a certain instant of time, certain servers can be more important (better ranking) than others as starting points for object searches.

The proposed type of indexing is useful for cases with a large number of objects and long time period of the data collection where it is impractical to maintain information about each object behavior. Nevertheless, the strategies proposed in this paper can be easily adapted to smaller systems in which one can maintain information about every individual stored in a centralized server. Our results show that, for these systems, it is possible to achieve good performance, even for cases in which the central server maintains only partial information about trajectories followed by the objects across the distributed spatio-temporal database servers. Certainly, a combination of both approaches is feasible; for example, for certain types of inviduals we can resort to maintaining detailed information of trajectories, whereas others are treated in bulk in accordance with their specific type classification.

This paper concentrates on defining and evaluating strategies for maintaining up-to-date a centralized meta-index of data about moving objects stored in multiple servers. We explore two classical approaches for maintaining centralized data: *harvesting*, by which the sources update the copies, and *crawling*, by which the central data repository accesses the sources. These types of strategies can be found in the context of the WWW. Most search engines use a *centralized crawler-indexer architecture*, which is associated with a *crawling* strategy for data collection. Particularly interesting for this work are strategies for refreshment [1] and crawling ordering [2] in the WWW, which have been associated with the relevance of web pages. Such relevance have been determined by using change frequency, incoming or outcoming relevant links, and information content. Unlike solutions for the WWW domain, our work presents a new time-varying rank for crawling or harvesting ordering, which is defined in terms of the variability and amount of information in each server.

To the best of our knowledge little research has been done so far on the kind of systems and restrictions we have in mind in this paper. When it comes to keeping track of moving objects, an important number of studies focuses on tracking the trajectories of an object, which means that the system knows the actual location of an object at distinct times [11] [10]. Such type of tasks can also benefit from the results of this work, since finding the starting location of an object within a time interval $[t_1, t_2]$ corresponds to the type of query addressed in this work.

Within the context of handling distributed data of moving objects, Mouza and Rigaux [6] propose a web architecture for scalable moving object servers. They come to the conclusion that it is helpful to divide the space into grids, so that each location of the space is contained in exactly one grid. Their work examines how to answer queries, in which an object enters, crosses or leaves a query window. A similar work that uses partitions of the space was done by Xia and Prabhakar [14]. They propose an index at two different levels: a low level for movements of objects and a high level that organizes the grid of partitions. Both studies based on partitions are focused on window queries and not on queries about locations of particular objects.

The main contributions of this work can be summarized as follows:

- It proposes a search strategy for querying moving objects in a distributed environment.
- It defines two approaches for maintaining the centralized meta-index: *crawling* (i.e., a centralized control of updating) and *harvesting* (i.e., an asynchronous distributed control of updating). This implies the definition of proper rules for triggering communication and performing the ranking of database servers to be contacted first.
- Unlike previous work related to data replication, this work defines strategies that consider only the local state of servers rather than the state of all other servers. The idea is to minimize the exchange of data between servers and between servers and the centralized system.

The organization of this paper is as follows. Section 2 presents the overall system and the search strategy. Strategies for collecting and updating a centralized meta-index are presented in Section 3. Experimental results for comparisons of the harvesting and crawling strategies are shown in Section 4. Conclusions and future research directions are in Section 5.

2 Overall System and Search Strategy

We assume systems composed of several distributed servers that collect information about moving objects. These servers record object movements that occur or have occurred. We aim at a very large number of servers and objects. Each server controls the objects' locations and events (i.e., change of location) within a specific space partition or region. For simplification, the distribution of servers avoids overlapping regions such that the information about an object's location at a specific time instance is stored in only one server.

In this environment, objects move in or move out of regions handled by servers. Unlike previous studies that have addressed queries concerning spatio-temporal windows, such as *what objects were/are in a spatial window at a particular time interval?*, this work aims at providing a method for answering queries of the type *where was a specific object at a given time interval?* This is a particularly challenging query for moving objects with no additional information about previous objects' locations. In a basic system, such query is processed by sending queries to all servers or agents, with overloading consequences for the system. We propose to answer this query with a by using a meta-index that reduces the search space.

For a large-enough distributed database servers system, wherein each server contains large collections of objects of many different types, it is clear that using such a meta-index can reduce the searching time of particular objects by avoiding blind searches that start, for example, at randomly selected servers.

The basic algorithm we use to conduct searches is as follows,

// Query= where was object o of type c at time t ?.

1. Rank servers according to priority values given by the number of objects of type c at time t in every server. Greater numbers imply higher priorities.
2. Visit every server in ranking order asking for the later event of object o found at time less than or equal to t. Let s be the first server found to have an instance of o.
3. Start the search at server s following the path left by o until reaching the server in which the first move-out event of object o is greater than t. Report solution for Query.

This algorithm can be easily modified to go towards the past or present depending on wether the event nearest to the query time t is a move-out or a move-in respectively.

In table 1 we show results for this strategy and query (we use data from the GSTD generator described in section 4 of this paper). These are results for 10K and 50K moving objects traveling around 100 and 484 nodes (database servers). Movements follow a Gaussian or uniform distribution, whereas nodes (servers) are uniformly distributes over the geographic space. Therefore, the distribution of objects in each server is not always uniform.

In table 1, the column Seq and Prio show results for the average number of visited servers for the blind (Seq) and proposed (Prio) search strategies, respectively. The values are the average obtained by performing searches of randomly chosen objects. The column P/S (i.e. Prio/Seq) shows the effectiveness of the proposed searching strategy. [The results in columns Traj and T/P are explained by the end of this section.]

Certainly the efficiency is crucially dependent on the proper ranking of servers. For simplicity we use the number of objects of a given type but, the particularities of real-life systems should expose much more sophisticated ways of defining priorities for servers. This effect can be seen in table 1, looking at the differences

Table 1. Results for searches on a many moving object system

Gauss Distribution							Uniform Distribution						
Obs	Nds	Seq	Prio	Traj	P/S	T/S	Obs	Nds	Seq	Prio	Traj	P/S	T/S
10000	100	39.418	5.104	2.588	0.13	0.07	10000	100	46.498	26.358	2.494	0.57	0.05
10000	484	175.476	16.254	4.926	0.09	0.03	10000	484	216.734	107.834	4.788	0.50	0.02
50000	100	44.762	4.546	2.452	0.10	0.05	50000	100	46.354	25.822	2.318	0.56	0.05
50000	484	199.360	16.110	4.470	0.08	0.02	50000	484	217.594	107.726	4.412	0.50	0.02

P/S between the two spatial distributions (uniform and Gauss) for the initial deployment of moving objects. In the case of the uniform distribution, the number of objects per node tends to be similar in every node.

Using the same benchmark system, we performed additional experiments for the case in which both the communication bandwith and central server capacity allow one to maintain information about the database servers visited by each object; we call this information *object trajectories*. In particular, we set our simulator to maintain into the central server upto 20% of the object trajectories, provided that for each object at least the initial event is available in the meta-index. As shown in column Traj of table 1, the results are clearly more efficient and stable than in the case of the previous searching algorithm based on type of objects. Column T/S (i.e. Traj/Prio) shows the comparative performance with respect to blind searching (Seq).

The algorithm for searching based on object trajectories is simpler than the one based on object types. Processing a query for object i involves finding the event for i which is closer to the time of the query and then going to the database server s where that event took place and execute the step 3 of the above algorithm based on object types. Certainly this object-trajectories based searching strategy is much faster than the one based on object types. However, a clear information transfer cost tradeoff arises when we significantly scale up the overall system. In this case for practical systems we should expect the number of different object types to increase much more slowly than the number of moving objects. As conclusion, while object trajectories is an efficient search strategy, searching based on object types keeps a good balance between the computational cost of finding a specific object' location and maintaining the central index.

3 Strategies for Collecting Information

We explore two strategies for collecting data about moving objects, which can be related to the *crawling* and *harvesting* strategies for data replication [8], respectively.

A crucial assumption about the distributed spatio-temporal database server systems we deal with in this paper is that they are not supposed to be able to communicate/cooperate between them, namely they are not organized in a specific communication topology and no concept of neighboring servers exists.

That is, objects can move arbitrarily to any server during their lifetimes and can appear/ disappear in any of them at any time.

In the case of crawling, the central server knows addresses of database servers and in the case of harvesting every server knows the address of the central server. Satisfying this restriction ensures the development of strategies for very large and dynamic systems. For harvesting, the relevant problem to solve is how a particular server can autonomously decide based on local information when to establish a connection with the central server. A sufficiently large number of servers trying to establish a connection with the central server can quickly saturate the available bandwidth. A solution can be the use of clock-based synchronous strategies but it is well-known that this type of approaches are not scalable as they can unnecessarily force a no-so-relevant server to communicate with the central one; that is, they fail to put most relevant servers in a given period of time to make use of the bandwidth.

Both strategies are based on the idea of defining a *communication triggering rule*; that is, a rule that indicates the need for updating a server's information as a function of time (i..e, a time-varying rank for each server). The following definitions characterize the state of a server, which are used for determining the server's rank in an ordering policy for updating (priority).

Definition 1 (Average number of objects). *Given the different numbers of objects in a server i at consecutive time instants $(n(i,t_1)\dots n(i,t_n))$ and the time difference $\Delta(t_j) = t_j - t_{j-1}$, the average number of objects in the server i in the time interval $[t_1,t_n]$ $(AvgN(i,t_1,t_n))$ is given by*

$$AvgN(i,t_1,t_n) = \frac{\sum_{\forall t_j, 1<j\leq n} n(i,t_j) * \Delta(t_j)}{t_n - t_1} \tag{1}$$

As an example, consider Figure 1. This figure indicates that there are two time instants when the number of objects changes in a server i in the time interval [0,6]. The average number of objects in the time interval $[0,6]$ is given by

$$AvgN(i,0,6) = \frac{10*2 + 9*1 + 4*3}{6} = 6.8$$

Every time the system checks for the need of updating information of a server i, it defines the average number of objects in that server since the last check.

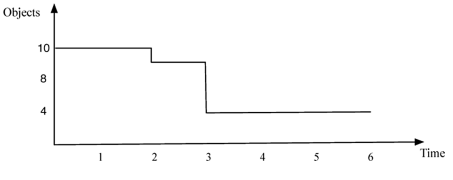


Fig. 1. Variation in number of objects in a server i

Considering consecutive checks of the system after the last update, one can determine the maximum average number of objects since the last update of the server. Because the system may check more than once before being able to update, there may be more than one average number of objects.

Definition 2 (Normalized variation). *Given the total number of objects in a server i at time instant t_1 ($n(i,t_1)$) and the maximum average number of objects in the server in the time interval $[t_1,t_2]$ ($MaxAvgN(i,t_1,t_2)$), the normalized variation of a server is given by*

$$\alpha(i,t_1,t_2) = \frac{MaxAvgN(i,t_1,t_2)}{n(i,t_1)} \tag{2}$$

Definition 3 (Weighted average of objects). *Given a total number of objects N, the number of objects in a server i at time t ($n(i,t)$), and the time difference between consecutive time instants $\Delta(t)$, the weighted average of objects in the server for a time interval $[t_1,t_2]$ is given by*

$$\beta(i,t_1,t_2) = \frac{\sum_{\forall t_j, t_1<t_j\leq t_2} n(i,t_j)\Delta(t_j)}{(t_2-t_1)\times N} = \frac{AvgN(i,t_1,t_2)}{N} \tag{3}$$

Definition 4 (Normalized rate of update requests). *Let $N_\alpha(i,t_1,t_2)$ be the number of times that the normalized variation of a server during a time interval $[t_1,t_2]$ is larger than a tolerance threshold τ (i.e, $\alpha(i,t_1,t) > \tau$, with $t \leq t_2$). The normalized number of rate requests is given by*

$$\delta(i,t_1,t_2) = \frac{N_\alpha(i,t_1,t_2)}{(t_2-t_1)} \tag{4}$$

Note that the highest values for $\beta(i,t_1,t_2)$ and $\delta(i,t_1,t_2)$ (i.e., 1.0) represent the case when the server concentrates the total set of objects and update requests at all time instances. We consider this case, the highest possible value associated with the importance of updating a server, i.e., the highest rank for data collection (or update).

We represent the state of servers as points in a 2D space, with one of the axis being the weighted average of objects $\beta(i,t_1,t_2)$ and the other axis being the normalized number of update requests $\delta(i,t_1,t_2)$. Using this space, the server rank for data collection is defined by the inverse of the distance between the state point of a server and the maximum reference point (1.0,1.0) (Equation 5, Figure 2).

$$rank(i,t_1,t_2) = 1 - \sqrt{\frac{(1.0-\beta(i,t_1,t_2))^2 + (1.0-\delta(i,t_1,t_2))^2}{2}} \tag{5}$$

The two strategies for updating the centralized meta-index are:

– **The crawling strategy** follows a similar approach to the *crawling* strategy of WWW search engines [3]. There exists a *scheduler* that makes use of a

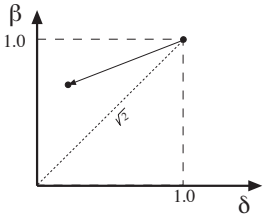


Fig. 2. Rank space

list of servers to assign the job of retrieving one server to each of a set of so-called *robots*. Normally, the scheduler fetches a server and passes it onto the next idle robot. Once a robot retrieves the data associated with its current server, it also extracts from it statistics data (i.e., number of update requests and current number of objects per type), and passes them to the scheduler. These data are used to re-calculate the ranking of servers in the scheduler's list.

– **The harvesting strategy** makes each database server to control the update of its information in the centralized system. There exists a *harvester* that receives requests from servers to update information. To trigger the request of an update to the meta-index, each server periodically calculates the α value defined by Equation 2. In the case that the relative variation of values is larger than a tolerance threshold τ, the database server sends an update request to the harvester. If the harvester is idle, it receives the information; otherwise, the server needs to re-try latter.

In a sense, crawling uses *global* information of the recent past to decide what servers should communicate with the central one, whereas harvesting uses all *local* information available upto the very present time.

For the searching strategy based on object types what is passed on to the central server during a communication action is a representation of the variation function for the number of object of each type observed in the period, whereas for the strategy based on object trajectories what is passed on is a list of events associated with every moving object.

Both cases allow the sending of partial information that affects noticeably the performance of the searching algorithm started at the central server side. For the object-types strategy, things are simpler as we only need to reconstruct an approximation of the curve shape. The object-trajectory strategy reduces the detail in the description of objects' trajectory in the system. The strategy we used in the experiments reported in section 2 of this paper is as follows. We reported the central server just *move_in* events per objects in the database whose last trajectory updated in the central server was before Δt units of time. The value Δt was set so that the total number of events kept in the central server was on average 20% of the global total number of events in the system.

4 Experimental Evaluations

The problem of gathering data of moving objects in a distributed system is a hard scheduling problem that can be modeled mathematically and simulated as we propose in this paper. Our approach to implementing the crawling and harvesting strategies is based on the use of object-oriented discrete-event simulation models for analyzing the behavior of data collection strategies.

To compare different strategies, we consider the time period when the meta-index is out of date. Consider the Figure 3 describing the changes in the number of objects, per type, in a given server, where at times t_0 and t_n we have data collections for a given node. Between $[t_0, t_n]$, there were 4 events that occurred (2 move_in and 2 move_out events). The time in which the information was out of date before the collection at time t_1 can be calculated as the sum of differences between the time of data collection and time of previous no collected events.

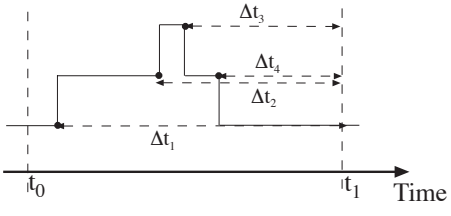


Fig. 3. Period of no updated data

Definition 5 (Divergence metrics). *Given $Ev(i,t)$ the set of events at node i and time instant t, $n(i,t)$ the number of objects in server i at time t, and N the total number of objects, the divergence metrics for the node i in a time interval $[t_1, t_2]$ ($D_n(t_1, t_2)$) is defined as the weighted sum of the temporal gap between events and updates, that is,*

$$D(i, t_1, t_2) = \sum_{\forall t, t_1 \leq t \leq t_2} |Ev(t, n)| * (t_2 - t)) * n(i, t_2)/N \tag{6}$$

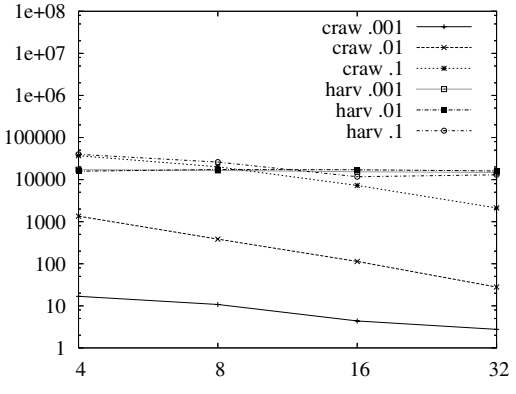
In addition to the divergence metrics, we analyze for harvesting strategy the time when the harvester was unable to attend a request for receiving data from a server, i.e., it was busy.

The experiments use two synthetic data about moving objects:

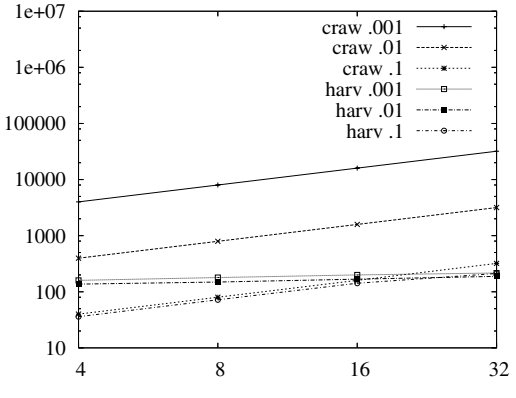
- **Data from the generator GSTD [13].** Sets with 10000 and 50000 points (objects), respectively, were created. These points are distributed uniformly or with a Gaussian distribution in a region at time instant 0.0. Subsequently, they are moved randomly during the next 1000 time instants. Using this data, we created a uniform distributions of servers (100 and 484 servers) that store the movements of objects.

– **A simulated environment of objects and servers.** A number of 100 and 500 servers were distributed uniformly or with a gaussian distribution. Each of these servers was initially set with a random number of objects to obtain 10000 to 50000 moving objects. The regions controlled by servers were defined by using a Delaunay Triangulation [9]. The movements of objects follow the shortest paths between two random servers in the neighboring graph of space partitions.

Results with the GSTD Data Set. In figures 4 and 6 we show results for the crawling and harvesting strategies. The tolerance *tau*, i.e., the threshold for the observed change of the normalized variation of a server (section 3), was kept at 0.15 in all experiments (similar values were observed for $tau < 0.3$). In the figures *Divergence* is a measure indicating how efficient is a given strategy to update the meta-index. Smaller values indicate better efficiency. We performed experiments for different network latencies, namely 0.001, 0.01 and 0.1. The curve

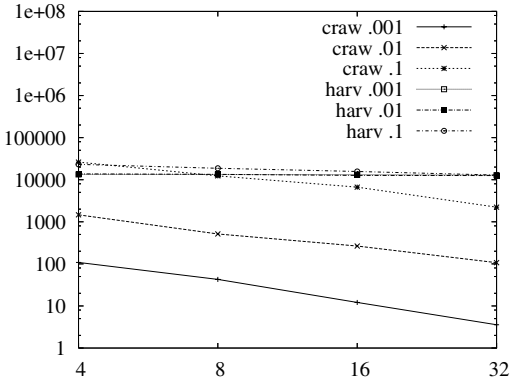


(a) Divergence

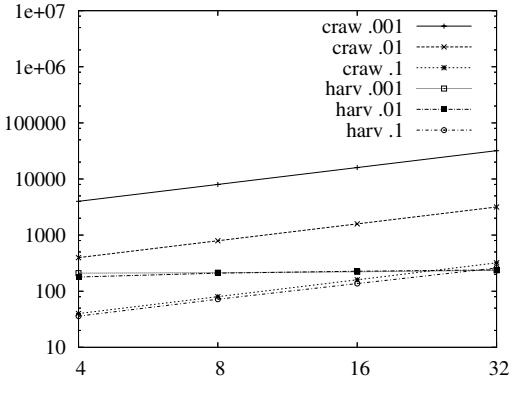


(b) Visits

Fig. 4. Uniform distribution of 10,000 objects



(a) Divergence



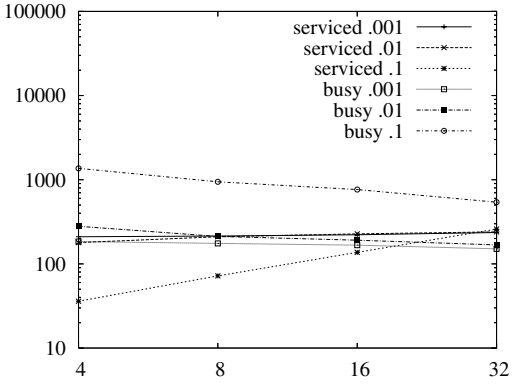
(b) Visits

Fig. 5. Gaussian distribution of 10,000 objects

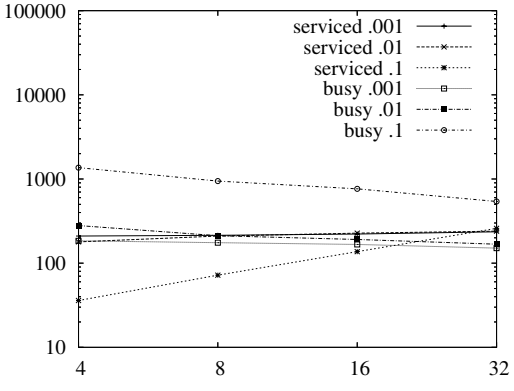
labels are suffixed with 001, 01 and 1 to identify the respective latencies. We also experimented with 4, 8, 16 and 32 bandwiths which means that the central server can hold 4, 8, 16 and 32 concurrent connexions respectively. These are the values for the x-axis in the figures. The measure *Visits* indicates the number of times a given entity collected data from the system in order to update the meta-index.

The results in figures 4 and 5 clearly indicate that crawling is more efficient than harvesting in terms of keeping better updated the meta-index. This stands over the different experiments. This is because the crawling strategy is able to visit more frequently the database servers. Similar results were obtained for 50,000 objects.

In figure 6, we show that the harvesting strategy is not able to achieve this because its lack of global synchronization leads to many failed attempts to contact the central server (in those cases the server is busy attending other database servers requests). Only when network latencies are very high, the crawling strategy tends to be similar to the harvesting strategy.



(a) 10,000 objects



(b) 50,000 objects

Fig. 6. Gaussian distribution for the harvesting strategy. Curves show the number of attempts to contact the central server in order to update its meta-index.

Table 2. Results with a random uniform distribution

Delay	Connexions	Divergence Crawling	Divergence Harvesting	Visits Crawling	Visits Harvesting	Busy Harvesting
0.1	4	38961	2189827	4000	191	925
0.1	32	3574	2278456	32000	217	395
0.5	4	196699	2083029	797	172	3215
0.5	32	22316	2088935	6369	309	901
1.0	4	393697	1962487	397	148	5994
1.0	32	44385	2118937	3169	359	1591

Results with the Simulated Servers and Moving Objects. The results for these experiments are shown in tables 1 and 2. Similar conclusions to the above experiments hold in this case as well.

Table 3. Results with a random clustering distribution

Delay	Connexions	Divergence Crawling	Divergence Harvesting	Visits Crawling	Visits Harvesting	Busy Harvesting
0.1	4	40706	2360123	4000	185	895
0.1	32	4043	2383329	32000	233	415
0.5	4	207966	2198299	797	170	3325
0.5	32	23870	2288753	6369	315	921
1.0	4	416594	2043532	397	162	6126
1.0	32	46907	2282859	3169	350	1571

5 Conclusions

We have presented a strategy to maintain up-to-date a meta-index which allows a centralized server to conduct searches for particular objects in a spatio-temporal moving objects system. We have evaluated two strategies (crawling and harvesting) to collect information from the distributed database servers that maintain track of the moving objects located at a particular area.

We used demanding synthetic work-loads to evaluate the performance of the crawling and harvesting strategies under exactly the same conditions and scenarios. The results show that crawling is more efficient than harvesting as it keeps the meta-index closer to the real system. This is particularly evident for large number of objects. This advantage stands even in conditions of very high latency in communication of information from the database servers. The reason is that crawling is able to exploit in a more efficient manner the available bandwith for communication.

Acknowledgment. This work has been funded by CONICYT, Chile, under grant FONDECYT 1050944. Mauricio Marín's research work was also funded by UMAG PR-F1-01-IC-04. Tonio Fincke's work was funded by ALFA Program EduGI.LA II.

References

1. A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, August 2001.
2. R. Baeza-Yates, C. Castillo, M. Marín, and A. Rodríguez. Crawling a country: Better strategies than breadth-first for web page ordering. In *WWW05, Industrial Track*. ACM Press, 2005.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley/ACM Press, 1999.
4. Edith Cohen and Haim Kaplan. Refreshment policies for web content caches. *Comput. Networks*, 38(6):795–808, 2002.
5. L. Do, O. Ram, and P. Drew. The need for distributed asynchronous transactions. *ACM SIGMOD Record*, 28(2):534–535, 1999.

6. C. du Mouza and P. Rigaux. Web architectures for scalable moving object servers. In *Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 17 – 22, New York, NY, 2002. ACM Press.
7. Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *ICDE*, pages 767–778. IEEE Computer Society, 2005.
8. Christopher Olston and Jennifer Widom. Efficient monitoring and querying of distributed, dynamic data via approximate replication. *IEEE Data Eng. Bull.*, 28(1):11–18, 2005.
9. J. O'Rourke. *Computational Geometry.* Cambridge University Press, 1993.
10. Kostas Patroumpas and Timos K. Sellis. Managing trajectories of moving objects as data streams. In Jörg Sander and Mario A. Nascimento, editors, *STDBM*, pages 41–48, 2004.
11. C. Shahabi, M. Kolahdouzan, S. Thakkar, J. Ambite, and C. Knoblock. Efficiently querying moving objects with pre-defined paths in a distributed environment. In *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, pages 34 – 40, New York, NY, 2001. ACM Press.
12. Yufei Tao and Dimitris Papadias. Historical spatio-temporal aggregation. *ACM Transactions on Information Systems*, 23(1):61–102, January 2005.
13. Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In Ralf Hartmut Güting, Dimitris Papadias, and Frederick H. Lochovsky, editors, *SSD*, volume 1651 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 1999.
14. Y. Xia and S. Prabhakar. Efficient CNG indexing in location aware services. In *23rd International Conference on Distributed Computing Systems Workshops*, page 414. IEEE Press, 2003.

A Generic Deployment Framework for Grid Computing and Distributed Applications

Areski Flissi[1] and Philippe Merle[2]

[1] LIFL / CNRS
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq, Lille, France
areski.flissi@lifl.fr
[2] INRIA Futurs / LIFL
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq, Lille, France
philippe.merle@inria.fr

Abstract. Deployment of distributed applications on large systems, and especially on grid infrastructures, becomes a more and more complex task. Grid users spend a lot of time to prepare, install and configure middleware and application binaries on nodes, and eventually start their applications. The problem is that the deployment process is composed of many heterogeneous tasks that have to be orchestrated in a specific correct order. As a consequence, the automatization of the deployment process is currently very difficult to reach. To address this problem, we propose in this paper a generic deployment framework allowing to automatize the execution of heterogeneous tasks composing the whole deployment process. Our approach is based on a reification as software components of all required deployment mechanisms or existing tools. Grid users only have to describe the configuration to deploy in a simple natural language instead of programming or scripting how the deployment process is executed. As a toy example, this framework is used to deploy CORBA component-based applications and OpenCCM middleware on one thousand nodes of the French Grid5000 infrastructure.

Keywords: Grid computing, middleware, distributed applications, deployment, software components.

1 Introduction

Thanks to the availability of some grid infrastructures as Grid5000 [1] –the French grid infrastructure for research– software deployment tends to become a major research topic for software engineering and distributed systems communities. Deployment can be defined as a set of tasks to realize such as software installation/uninstallation on remote nodes and activation/deactivation of software instances. These deployment tasks have to be orchestrated in a defined order. For instance, software activation cannot be done until it has been downloaded, installed and configured. Many works try to address, at different levels, the problem of software deployment on large distributed systems. On one hand, in the field of middleware, most of component-based platforms offer deployment facilities allowing

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1402 – 1411, 2006.
© Springer-Verlag Berlin Heidelberg 2006

to deploy component-based applications. For example, the OSGi technology [2], which is dedicated to services executed on gateways, provides mechanisms for remotely deploying and managing OSGi bundles representing end-user services. JOnAS [3] or OpenCCM [4] platforms also provide the same kind of features to respectively deploy J2EE and CCM applications. But, these platforms do not address the problem of the middleware deployment itself. Concerning grids, ProActive middleware [5] addresses the deployment of servers and applications but built on top of ProActive only.

On the other hand, many *adhoc* solutions for deploying software, as file transfer protocols to download and install software on remote nodes (*e.g.*, FTP, HTTP, scp, rsync) or remote access protocols to launch processes (*e.g.*, SSH, telnet, rlogin), exist and are often used. In addition, grid infrastructures themselves provide sometimes end-users tools to deploy applications. For instance, Grid5000 provides the OAR tool [6] for reservation of nodes and Kadeploy [7] for deployment of customized operating system images.

Thus, software deployment on large infrastructures (grids are composed of thousands nodes) becomes a more and more complex task: Grid users spend a lot of time (from hours to days!) to prepare, configure nodes, install middleware and software and finally start their experiences. In support of this view, let's just take an example about deployment of CCM applications on the Grid5000 infrastructure. At first, users must connect to the frontal node (using *ssh*) of one cluster of the grid, check for available nodes, submit a reservation request using OAR, and get the list of allocated nodes. Then, the CCM application servers that will host applicative components have to be installed on each node, plus all required libraries and external stuffs (*e.g.*, a *Java Runtime Environment* or a CORBA runtime). To achieve that, users will probably use classical techniques to download, unzip and install binaries on remote nodes or can use Kadeploy to deploy a specific image previously built. Unfortunately, this second step has to be repeated for each experience because Grid5000 is a shared infrastructure, which means nodes are reinitialized by next users. After that, component servers have to be started on each node, but also useful middleware services. However, some dependencies exist between these processes. For example, the CORBA Naming Service must be started on a node before all CCM component servers. At this stage, the execution support is finally ready for the deployment of the application. This is true here (but not always) as the OpenCCM platform provides a distributed deployment infrastructure named DCI [8] that manages the "high-level" deployment tasks, *–i.e.*, installation and instantiation of component binaries on nodes, binding of components and finally activation of component instances.

Our contribution in this paper is about a generic deployment framework dedicated to large systems as grid infrastructures. Our approach consists in reifying existing mechanisms and tools involved in the deployment process as Fractal components [9]. These components are composed together which allows to assemble heterogeneous mechanisms for an automatized execution. Software to deploy, whatever the granularity is, and physical infrastructure are also reified as components. Bindings between components represent dependencies. Finally, the assembly of components represents the configuration of the system to deploy.

The reminder of the paper is organized as follows. Section 2 describes the key principles of our framework and the generic deployment components. Section 3

explains how to write software personalities for deployment of specific software and gives, as an illustration, a concrete example about the deployment of OpenCCM middleware on Grid5000. In Section 4, related works are discussed, and finally, Section 5 concludes this paper and gives some future works.

2 A Generic Deployment Framework

2.1 Overview of the Approach

Our approach is based on a generic deployment framework allowing to build, using personalities, the appropriate tool to automatically orchestrate the deployment process of software, regardless of the technology or the granularity (from middleware platforms, application servers to applications, services, or even objects), as illustrated in Figure 1.

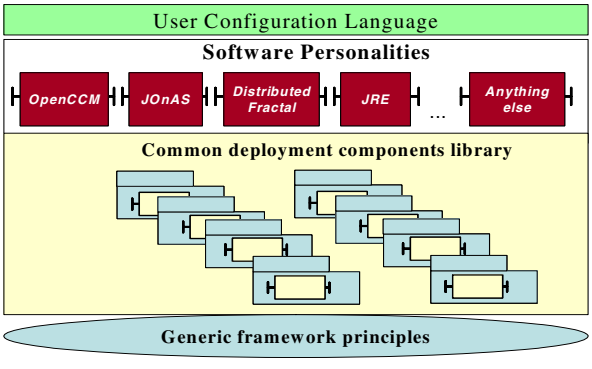


Fig. 1. Overview of the generic deployment framework

The deployment framework first relies on some key principles, as "everything is reified as components", provides a common library of deployment components, and finally personalities written once by developers for specific technology or software. The main advantage for users of such a tool is to allow to just describe configurations –*i.e.*, nodes of the targeted infrastructure and services[1] to deploy on them– instead of programming the way deployment has to be executed and without taking care of the orchestration of deployment tasks. As a consequence, automatization of heterogeneous activities becomes possible thanks to our framework.

2.2 Everything Reified as Components

First of all, the main requirement of our work was not to re-invent the wheel but to rely on existing mechanisms. In other terms, our goal is to provide, through a generic framework, a simple way to reuse and especially compose existing heterogeneous tools for automatic deployment of software on very large systems as grids. The first

[1] The word "*services*" is used in this paper as a generic term to reference any kind of "software": components, objects, scripts, programs, middleware services, application servers, processes, etc.

key principle of our approach is that everything is reified as software components [10], or more precisely Fractal components. "Everything" here includes all the elements that are manipulated or involved in the deployment process:

- Physical hosts, such as Grid5000 nodes
- Remote access protocols, such as *SSH*, *Telnet*, etc.
- File transfer protocols, such as *FTP*, *HTTP*, *scp*, *rsync*
- Shells, such as *SH*, *CSH* or *Windows Command Shell*
- Shell variables and commands
- Software and tools, such as *Java Runtime Environment*, *OAR* or *Kadeploy*
- Middleware services, such as a *CORBA Naming Service*
- Application servers, such as *OpenCCM*, etc.

The second principle is to compose these components in order to obtain an assemby (or composite) which symbolizes software to deploy. In other words, the execution of this composite corresponds to the execution of the deployment process. Bindings between components represent dependencies. For example, the *Shell* component reifies a "real" shell and provides, through a server interface, a method to execute shell commands. For that, it needs a remote access protocol to send the command to a remote host. So, *Shell* component is bound to a *Protocol* component, itself bound to other components for getting hostname, port and user access information.

Finally, the third principle of our approach, is that orchestration of components contained in the global composite is done by the framework, not by the user. To allow that, every component contained in the composite provides a server interface named `Deployment` that offers the generic methods `install()`, `start()`, `stop()` and `uninstall()`. Components are executed according to the state diagram logic (install if is uninstalled then start; stop if is started then uninstall) illustrated in Figure 2 and in an order that is determined by bindings between components. For instance, in the case of the JRE software, that is used by many other software as OpenCCM or JOnAS, we can draw different scenarios depending on the fact the JRE is already installed or not on the node. If not, the JRE has to be installed before, whereas if it is already installed, it has just to be started –*i.e.* set the `JAVA_HOME` shell variable and add *java* binary to the `PATH` environment variable using the *deployment components*.

This process is very similar to the orchestration framework of tasks we defined in [11]. The choice of Fractal is well suited here since it is a hierarchic component model supporting sharing of component instances and component introspection operations. The *specific action* mentionned in Figure 2 represents a particular

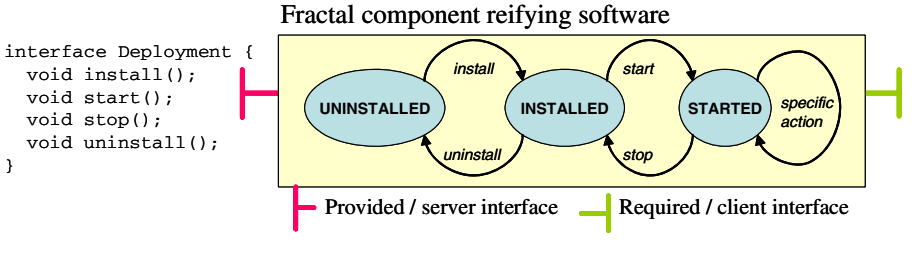


Fig. 2. Generic state diagram used for orchestration of components

"business" operation provided by another server interface of the component (*e.g.,* the `send()` method of the *Protocol* component) and used by another component (*e.g.,* the *Shell* component).

2.3 A Common Reusable Library of Deployment Components

We have developed a set of reusable deployment components, that, relying on our deployment framework and combined to specific components (software personalities) can be composed together by users to describe a configuration and automatically deploy it on the grid. These components provide common server interfaces that are used during the deployment process. These main interfaces are described in Table 1. Many implementations of these interfaces are possible. Thus, the implementation of the *Protocol* interface can be based on *SSH* or *Telnet* whereas the implementation of *Shell* can be *SH*, *CSH* or *Windows Command*. What should be emphasized again here is that our components do not re-implement things. For example, we developed one *Telnet* implementation of the *Protocol* component that uses a free open source external library (named *JTelnet*) and one implementation of the *SSH* component that uses external OpenSSH process.

Table 1. Main component interfaces of our generic deployment framework

Interface	Main offered methods and description
Hostname	`getInternetHostname()`: Return or (dynamically) compute an Internet hostname (or IP address).
Port	`getInternetPort()`: Return an Internet port for protocols (*e.g.,* 22 for *SSH*).
User	Provides methods for getting user access information (login, password, RSA private key, etc.).
Protocol	`send(`*`args`*`)`: Send commands to a remote host.
Shell	`execute(`*`args`*`)`: Execute shell commands.

On top of these fundamental deployment components, we wrote some other reusable components such as JRE. JRE can be defined as a *service* to "start" in user's configuration or can be used by other service components (*e.g.,* OpenCCM or JOnAS need a JRE). As explained in Section 2.2 (all *services* are reified as components that implement the *Deployment* interface), JRE implements install, start, stop and uninstall methods. For that, it is bound to fundamental deployment components (such as *Shell*, *Variable*, etc.) and others like a *File Transfer* component (for installation of the JRE archive from the SUN web site!). Figure 3 illustrates an example of assembly for this particular useful component.

Assemblies and complex composites are described once by developers via Fractal *Architecture Description Language* (ADL) files. This allows users to ignore bindings (dependencies) between components and provides a high level of abstraction of the deployment process: deployment mechanisms, dependencies between software or orchestration complexity are hidden for users and managed by the framework.

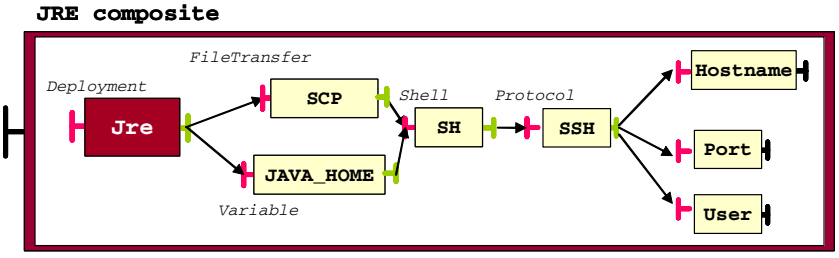


Fig. 3. An example of assembly for JRE component

3 Illustration: Automatic Deployment of OpenCCM on Grid5000

This section illustrates a concrete example of the use of our deployment framework to deploy the OpenCCM middleware on Grid5000.

3.1 Write Personalities Once, Work Everytime

For developers, writing personalities mainly consists in writing Fractal ADL files describing composites of components, and in some cases, specific components implementing specific business code. At first, in the case of OpenCCM, we wrote few composites and primitive components that reify OpenCCM entities: OpenCCM (as the software), NS (the CORBA Naming Service), DCI (*OpenCCM DCI Manager* that manages distributed deployment of CORBA components) or *DCI_NODE* (servers that host CORBA components). These composites are a composition of our generic deployment components and of components they depend on, as JRE. Secondly, in the purpose of deployment on Grid5000 infrastructure and according to the same principles used for OpenCCM components, we wrote components for OARGrid (reservation of nodes on many clusters of Grid5000) and Kadeploy (deployment of customized images) tools.

We have finally written a specific implementation of the *Hostname* interface allowing to provide a dynamic hostname used by the *Protocol* component. Indeed, hostnames of nodes allocated by OARGrid aren't known in advance. Moreover, as we will see in Section 3.2, this allows users to describe configurations of the physical infrastructure simply by "telling" that hostnames are dynamically computed (this point is very useful for Grid5000 as currently grid users' applications have to manage themselves a dynamic list of allocated nodes).

3.2 Write User Configurations to Deploy

Let's now see what users finally have to write in order to automatically initiate a complex heterogeneous deployment process. As we have seen in previous sections, the main goal is to allow users to just describe in a simple natural language their configurations (the services to deploy and the targeted nodes) without taking care of how things are done. An example of a simple configuration is given below.

```
MyDeployment = OpenCCM.Deployment {
  nodes = {
        hostname = DynamicHost(~/nodelist)
        apply FOR(i,0,500) {
           node-%{i} = Grid5000_NODE {
                    hostname = nodes/hostname
                    protocol = SSH
                    port = Port(22)
                    shell = SH
                    user = User(aflissi, ,~/.ssh/id_rsa.pub)
                    jre = Jre(/opt/java/jdk-1.5.0_05)
                    openccm = OpenCCM(/opt/OpenCCM-0.9,/opt/CORBA/JacORB-2.2)
   } } }
  services = {
        ns = OpenCCM.NameService {
                node = nodes/node-0 }
        dci = OpenCCM.DCIManager(MyDCI) {
                ns = services/ns
                node = nodes/node-0 }
        servers = ParallelRunner {
                apply FOR(i,1,500) {
                   server-%{i}= OpenCCM.DCI_NODE(NM_%{i}){
                           dci = services/dci
                           node = nodes/node-%{i}

   } } } } }
```

This piece of code uses a simple *Domain Specific Language* (DSL) that is composed of two parts: the first one (*nodes {...}*) describes configuration of the targeted nodes and the second one (*services {...}*) specifies services to start and where to start them. Here the user simply describes that: 1- he/she uses Grid5000 nodes with specific configurations for protocol (SSH), shell (SH), JRE, OpenCCM; hostnames are dynamically computed, and 2- he/she wants to start an OpenCCM server (DCI_NODE) on each node (*node-1* to *node-500*) and the *CORBA Naming Service* (NS) on *node-0*. Services are by default sequentially executed (*ns* first, then *dci*, then *servers*) but user can specify particular behavior using tag such as *ParallelRunner* for parallel deployment of servers. Other examples of *services* can be specified in the configuration: OARGrid, Kadeploy or anything else. If a reservation task is needed, the user simply has to add it in the configuration thanks to the following lines:

```
  nodes = {    oar_server = Grid5000_NODE {
                 hostname = StaticHost(oar.lille.grid5000.fr)
   } }
  services = { reservation =
                 OARGrid(gdx=300|azur=100|grillon=50|lille=50,~/nodelist) {
                 node = nodes/oar_server

   } }
```

From user's configuration, a Fractal ADL file is generated and interpreted by the Fractal ADL tool to instantiate the global composite that abstracts the system to deploy. The generated Fractal ADL file contains definitions of both generic deployment components and software personalities. The deployment framework, relying on existing infrastructure, is then in charge of the orchestration of tasks, the management of software dependencies, or the execution of specified policies –i.e., sequential or parallel deployment of services. Figure 4 illustrates the deployment of our example on the Grid5000 infrastructure. Components contained in the global composite are executed according to a specific order computed by our orchestration logic. In this example, OARGrid is started first since it has no dependencies (no

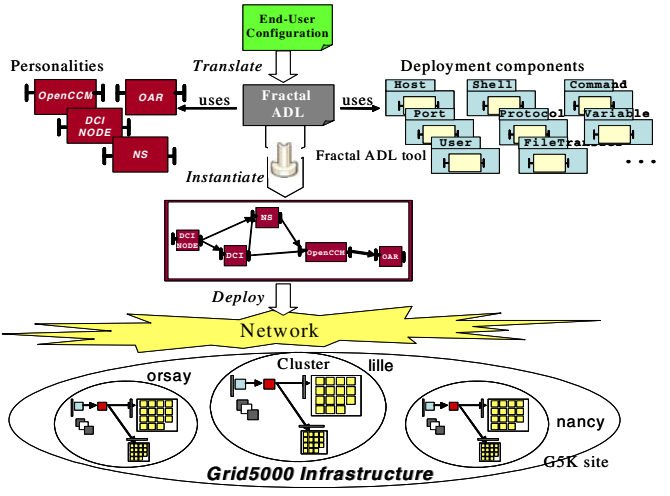


Fig. 4. Automatic deployment of the OpenCCM middleware on Grid5000

required interfaces): inside the OARGrid composite, fundamental deployment components (shell, protocol, hostname, port, user, etc.) are started to send the *oarsub* request *–i.e.*, submit a reservation– to a remote server of Grid5000 (here *oar.lille.grid5000.fr*). Then, the OpenCCM component is executed, which means: install OpenCCM middleware using a *File Transfer* component to download binary from OpenCCM web site, and send commands on each remote node to set and configure OpenCCCM shell variables. Next, the NS component and at least the *DCI_NODE* components (CCM application servers) are started.

3.3 Evaluation and Performances

Deployment of OpenCCM middleware up to one thousand nodes of Grid5000 have been successfully tested using our framework. For this case, effective deployment

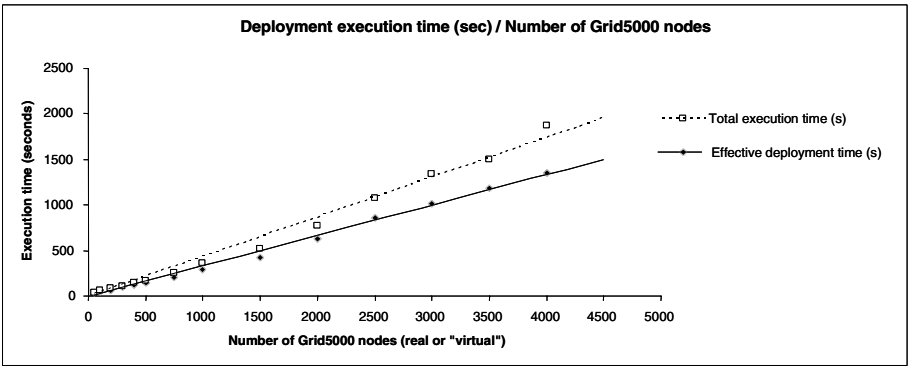


Fig. 5. Execution time of the deployment process of OpenCCM middleware on Grid5000

time (without the overhead introduced by the framework when parsing the user's configuration file and that is due to the Fractal ADL tool) takes less than 300 seconds, with a parallel execution. Indeed, deployment process is composed of two main steps: 1) loading of configuration and instantiation of components by the Fractal ADL tool, and 2) execution of the deployment process itself. What's more, effective deployment time is approximatively growing linearly with the number of nodes. All experimental results are showed in Figure 5. Grid5000 "virtual" nodes means here that many application servers may have been started on a same physical node.

4 Related Works

Many works are done on the deployment of distributed applications on large scale systems. Concerning application servers and gateways, technological platforms such as OSG*i* or J2EE address the problem of automatic deployment of applications. But, OSGi does not address deployment of distributed applications. Concerning both OSGi and J2EE, nothing is said about deployment of the middleware platforms themselves on gateways/servers. This task has to be done manually by administrators. OpenCCM encountered the same lack as it can only deploy CCM-based applications. Other works in the *Autonomic Computing* domain are based on the software component paradigm. JADE [12] uses Fractal components to reify the software for deployment and dynamic reconfiguration of J2EE systems. Nevertheless, JADE infrastructure must have been installed before on targeted platforms.

In the field of Grid Computing [13], ProActive allows the deployment of both ProActive-based servers and applications but is limited to this technology. *S. Lacour* proposes in [14] a software architecture for the automatization of the deployment process of distributed applications executing on grid infrastructures. Our goal is the same but the approach is different. Main key points are that, in one hand we rely on existing infrastructures, tools or paradigms for applications and middleware, regardless of the granularity, and, in the other hand we simplify the description of complex deployment's configurations (in a single file if needed).

5 Conclusion and Future Works

Composing heterogeneous mechanisms in order to automatize the deployment process of software executed on large scale infrastructures such as grids is the next challenge. Our approach is based on the *software component* paradigm. We have defined and implemented in Fractal a generic deployment framework and fundamental components that can be used to build specific deployment tools. Users only have to describe their configurations in a simple language and use the framework to instantiate the deployment. To validate the approach, we have successfully deployed the OpenCCM middleware on one thousand nodes of Grid5000.

Our future work will go in many directions. We first plan to implement personalities for other technologies such as J2EE or OSGi servers. Secondly, in order to optimize very large scale deployment, we will enhance the framework with the monitoring and control of local resources (sockets, threads, etc.) by distributing it. Finally, we would like to add *dynamic adaptation* aspects in our framework. For

instance, as for computation of Grid5000 hostnames, composites reifying services could be automatically assembled or reconfigured at runtime thanks to a dynamic discovery of available protocols, ports or shells for each node.

References

[1] F. Cappello *et al.*, "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform", In proceedings of the *6th IEEE/ACM International Workshop on Grid Computing*, Grid'2005, Seattle, Washington, USA, November 13-14, 2005.

[2] R. Hall and H. Cervantes, "Challenging in Building Service-Oriented Applications for OSGi", IEEE Communications Magazine, May 2004.

[3] F. Exertier, "J2EE Deployment: the JOnAS case study", In proceedings of *1ère Conférence sur le Déploiement et la (Re) Configuration de logiciels, DECOR'04*, Grenoble, France, October 2004.

[4] OpenCCM team, *The OpenCCM Project*, http://openccm.objectweb.org, ObjectWeb Consortium, 2002.

[5] F. Baude, L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, "Programming, Composing, Deploying for the Grid", In *GRID COMPUTING: Software Environment and Tools*, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, 2006.

[6] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components". In *Cluster computing and Grid 2005* (CCGrid05), 2005. Web site: http://oar.imag.fr

[7] The Kadeploy project, Web site: http://gforge.inria.fr/projects/kadeploy

[8] F. Briclet, C. Contreras, P. Merle, "OpenCCM : une infrastructure à composants pour le déploiement d'applications à base de composants CORBA", In proceedings of *DECOR'04*, France, October 2004.

[9] E. Bruneton, T. Coupaye, J.B. Stefani, "The Fractal Component Model" specification, version 2.0-3, ObjectWeb Consortium, February 2004.

[10] C. Szyperski, "Component Software: Beyond Object-Oriented Programming" – 2nd edition, New York, U.S.A., Addison Westley Longman, November 2002.

[11] A. Flissi, P. Merle, "Vers un environnement multi-personnalités pour la configuration et le déploiement d'applications à base de composants logiciels", In proceedings of *DECOR'04*, France, October 2004.

[12] S. Bouchenak, N. de Palma, D. Hagimont, "Autonomic Administration of Clustered J2EE Applications", IFIP/IEEE International Workshp on Self-Managed Systems & Services, Nice, France, May 2005.

[13] Ian Foster and Karl Kesserman, "The Grid: Blueprint for a new Computing Infrastructure", ISBN:1-55860-933-4, 2004.

[14] Sébastien Lacour, "Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul", PhD Thesis, Université de Rennes 1, IRISA, Rennes, France, December 2005.

CBIR on Grids

Oscar D. Robles[1], José Luis Bosque[1], Luis Pastor[1], and Ángel Rodríguez[2]

[1] Dpto. de Arquitectura y Tecnología de Computadores, Ciencias de la Computación e Inteligencia Artificial,
U. Rey Juan Carlos, C/ Tulipán, s/n. 28933 Móstoles. Madrid. Spain
{oscardavid.robles, joseluis.bosque, luis.pastor}@urjc.es
[2] Dpto. de Tecnología Fotónica, U. Politécnica de Madrid,
Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
arodri@dtf.fi.upm.es

Abstract. From the computational point of view, Content-based Image Retrieval systems are potentially expensive and have user response times growing with the ever-increasing sizes of the databases associated to them. This paper presents a grid implementation of a Content-based Image Retrieval system that offers a good cost/performance ratio to solve this problem due to their excellent flexibility, scalability and fault tolerance. This approach allows a dynamic management of specific databases that can be incorporated to or removed from the CBIR system in function of the desired user query. Experimental performance results are collected in order to show the feasibility of this solution.

1 Introduction

Grid computing is becoming nowadays a feasible solution for computer applications with high levels of computational power demand. This is due to the good price/performance ratio offered by the networks that compose this type of systems and because of both the high flexibility and availability offered by this computation paradigm, making easier the cooperation and resource sharing among institutions, named "Virtual Organizations" (VO) [1]. One application area where the concept of grid computing fits in a natural way is Content Based Image Retrieval (CBIR). The main purpose of CBIR systems is to help users to perform automatic information retrieval over image databases considering relevant visual features extracted from the image data in a preprocessing stage [2,3]. The complexity of this task depends heavily on the number of items stored in the system, but usually, when dealing with image or video databases, large volumes of data are considered, and therefore, alternative strategies to the conventional centralized server must be seeked to manage the storage and processing of data in an efficient and flexible way.

The tremendous improvements experimented by computers in aspects such as price, processing power and mass storage capabilities have resulted in an explosion of the amount of information available to people. But this same wealth makes finding the "best" information a very hard task. CBIR systems try to solve

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1412–1421, 2006.
© Springer-Verlag Berlin Heidelberg 2006

this problem by offering mechanisms for selecting the data items which resemble most a specific query among all the available information.

From a computational point of view, CBIR systems are potentially expensive and their user response times grow with the ever-increasing sizes of the databases associated to them. One of the most common approaches followed to reach acceptable price/performance ratios has been to exploit the algorithms' inherent parallelism at implementation time [4,5,6]. However, the novelty of CBIR systems hinders finding references dealing with this aspect. Some contributions that can be cited are Zaki's compilation [7], and the contributions of Srakaew *et al.* [8] and Bosque *et al.* [9].

A CBIR cluster based implementation was first introduced in [9], comparing its performance with a shared-memory solution. The results showed a definite advantage of the cluster in terms of scalability and price-performance ratio. In this paper, we extend the horizon of our previous work sharing the available resources of different VO. This paper presents a grid implementation of a CBIR system that offers a good cost/performance ratio to solve this problem due to their excellent flexibility, scalability and fault tolerance. The flexibility of the system herein presented allows the dynamical addition or removal of nodes from the grid between two user queries, achieving reconfigurability, scalability and an appreciable degree of fault tolerance. This approach allows a dynamic management of specific databases that can be incorporated to or removed from the CBIR system according to users queries. Experimental performance results are collected in order to show the feasibility of this solution taking into account several database setups.

The rest of this article is organized as follows. Section 2 presents a brief description of the operations involved in a CBIR system, and the grid implementation developed is discussed in Section 3. Section 4 shows some performance results achieved by this implementation and finally, section 5 presents some conclusions and ongoing work.

2 CBIR System Operation

Smeulders *et al.* [3] reveal the importance of CBIR techniques. An example of this is the management of the huge quantity of multimedia information that anyone can access on Internet. This information must be efficiently accessed in a user friendly way and CBIR systems can provide proper solutions. In our case, we are dealing with 128x128 colour bidimensional databases that range from four hundred thousand to thirty two million images.

The search for images contained in a CBIR system can be broken down into the following stages:

1. **Input/query image introduction.** The user first selects an image to be used as a search reference and the system computes its signature[1]. Details about the retrieval techniques involved in the CBIR system can be found in

[1] The signature is a vector of some features that represents the content of the images.

[10,11]. The whole process can be implemented using an *O(k)* order algorithm, being k the image's size, that performs in a very efficient way [12]. This stage does not require quite high computational resources since the system only deals with one image.

2. **Query and DB image's signature comparison and sorting.** The signature obtained in the previous stage is compared with all the DB images' signatures using a metric based on the Euclidean distance. The identifiers of the p most similar images are extracted. Not being a quite costly operation, the volume of computations to be performed is very high though, since the signature of the input image must be compared with each of the image signatures stored in the system. Should it become necessary to incorporate a new signature to the group of the best p, the one with the worst ranking within the group would be discarded and the set then newly sorted. A bubble sorting algorithm with *O(np log(p))* order has been used for this purpose, being n the number of images.
3. **Results display and query image update.** The system provides the user a data set with the p images considered most similar to the query one. If the result does not satisfy the user, he/she can choose one of the selected images or enter a new one that presents some kind of similarity with the required image returning to stage 1.

Upon observing the operations involved, it is possible to notice that the comparison and sorting stage involves much larger computational load than the others. Luckily, since there are no dependencies the exploitation of data parallelism can be done just by dividing the workload among n independent nodes. This can be accomplished by distributing off-line the CBIR image's signatures across the processing nodes, which can then compare the query image's signature with every available signature. The storage capacity becomes also a problem whenever the size of the CBIR system grows beyond a certain limit. The most efficient approach to solve this point, relaxing additionally the per-node storage demands, is to distribute signatures, images and computations over an ensemble of n processing and storage nodes.

3 Grid Implementation

3.1 Software Architecture

The grid implementation of a CBIR system corresponds to a distributed architecture of grid nodes with different complexity levels, ranging from standalone PCs or workstations to parallel systems like shared memory multiprocessors or clusters. Therefore, each one of the grid nodes' may run sequential or parallel versions of the software components of the whole application, configuring a very heterogeneous system.

The CBIR grid application programmed can be decomposed in the following modules: User interface, CBIR grid management and Local search per grid node. All these modules have been programmed using the Globus toolkit vs. 4 [13]. Next sections will describe each one of these elements.

3.2 User Interface

From a functional point of view, the main features of a grid system are flexibility and versatility. The software application must make feasible a user specification of a whole environment on every single system execution. This allows to update the working configuration adding or deleting specific nodes or databases taking into consideration the type and contents of the queries at a given moment. In this sense, the system presents to the user a simple interface where he/she can define parameters with two clearly different orientations:

- Grid features.
- CBIR features.

Following, both group of features will be described with some detail.

Grid Features. The grid parameters cover aspects such as:

- Number of grid nodes where the query will be run.
- Selection of specific nodes where the execution will be performed.
- Computational power of each one of the system nodes. This is and optional parameter. If the user has some *a priori* knowledge about the performance of the processing nodes that belong to the grid, these data can be introduced to the application. Anyway, the application collects some statistics about each node performance in order to make in a future a better assignment of the work load among all available resources in the grid.

CBIR Features. Among the programmable CBIR parameters, we can mention:

- Name of the process that will perform first the search and the the sort of the signatures in each node. This is an optional parameter, and if it is not specified by the user, a default option is chosen.
- Search criterion: available criteria include colour, shape and a combination of both features. For each criterion, several possibilities exist: energies, colour histograms, multiresolution colour primitives, Hu and Zernike invariants, etc. [11,14].
- Metric used in the similarity computation stage for the computed input image signature and the precomputed signatures stored in the database.
- File name of the query image.
- Name of the selected database where the query is performed. This is an optional parameter and if this entry is left blank, the default value is selected.

3.3 CBIR_GRID Process

This process is in charge of collecting all the parameter specifications defined by the users and starting the execution in the distributed environment. It is also in charge of keeping the fault tolerance of the whole system: if a grid node can not finish the search in its own local database, a report of this fact is sent to the user, but the rest of the answers are not lost. This process can be decomposed in the following stages:

- Read the system information provided by the users. At this moment, the setup information about the query performed by the users is received and all the local data structures involved during the search process execution are initialized, as well as those sent to the remote nodes.
- Setup of the testbed or grid system. At this point, the number of nodes that compose the testbed is fixed, then their availability is checked and finally, the permissions for executing remote jobs over them are verified.
- Selection of the remote jobs that have to be executed in each node of the grid.
- Compute the signature of the input image considering its energies or histograms, like has been described in [10,11,14].
- Send the previously computed input image signature to all the nodes that compose the grid. The service `GridFTP` provided by the Globus Toolkit vs. 4.0 is used for this purpose. Specifically, the secure command `globus-url-copy`.
- Once the signature is distributed, it must be launched the execution of the jobs in each one of the grid nodes, searching over their own local databases. A script that receives as input the list of nodes that compose the grid with its corresponding class of node has been programmed to perform this task. The remote execution of the jobs is based on the command or service `globus-job-submit` available on the Globus ToolKit vs. 4.0. This command is used instead of `globus-job-run` since it provides a non-blocking service, while `globus-job-run` performs a blocking execution of the submitted job. This command returns an identifier for each job, so with the command `globus-job-status`, its state can be checked passing as an argument the obtained job identifier. Monoprocessor nodes receive a sequential job, while cluster nodes receive a distributed job, and as it can be deduced, in this case, the search is performed over a distributed database.
- Then, the `CBIR_GRID` process performs a loop that controls the state of each one of the launched jobs. The control interval of the jobs is also an application parameter, setting two seconds as default value. When a job ends, it must collect through the `gridFTP` service the partial results generated by the corresponding grid node. This way, a set of files with partial results is generated, one for each node of the grid.
- The last step in this process is to gather all the partial results and to select the best N. Then, the process picks up the N images from the corresponding local nodes and presents them to the user in a sorted mosaic view inside a user window.

3.4 Local_Retrieval Process

This process is in charge of performing the local search in each one of the grid nodes. It has assigned two main functions:

- The comparison of the input signature with respect to all the stored signatures in the database considering the search criteria and metric distance specified by the user. These comparisons produce a set of similarity values that are stored in an output file.

– Once the result of all the comparisons are available, the output file is sorted, achieving an output sorted according to the similarity value.

4 Experimental Results

4.1 Experimental Setup

A set of experiments has been executed for testing the behavior of the CBIR grid implementation presented here. Several objectives are stated for the tests:

– to verify the feasibility of applying a distributed solution based on a grid,
– to estimate the overhead introduced by the Globus software.
– to analyze the grid response in order to optimize the distribution of the CBIR data among the grid nodes to achieve better performance.

The testbed used in the experiments presented next is composed by the aggregation of the resources of two VO: the Department of Arquitectura de Computadores (DAC) of the Rey Juan Carlos University (URJC) and the Department of Arquitectura y Tecnología de Sistemas Informáticos (DATSI) of the Universidad Politécnica de Madrid (UPM). Each one of these VO contributes with the following resources[2]:

– DAC-URJC:
  Africa: CPU Intel(R) Pentium(R) IV, 2.80GHz;
    - cache size L1 : 512 KB
    - main memory: 1 GB DDR RAM
    - hard disk: ST380011A PCI, 20 GB
    - operating system: Linux version 2.6.8-2-686, Debian 1:3.3.5-12
    - network card bandwidth: 1 Gbps

  Artico: CPU Pentium III (Katmai) at 450 MHz
    - cache size : 512 KB
    - main memory: 128 MB DDR RAM
    - hard disk: Maxtor 91080D5 PCI, 10 GB
– DATSI-UPM:
  Baobab: One biprocessor node with the following features:
    - 2 CPUs, Intel(R) Xeon(TM) CPU 2.40GHz
    - cache size : 512 KB (each CPU)
    - main memory: DDR RAM 1 GB
    - hard disk:
      * local: 4,6 GB
      * NFS: NAS Intel Pentium 4 CPU 2.8GHz. . Raid 0 over 4 disks with 160 Gb.
    - message passing libraries: LAM/MPI 7.1.1
    - linux kernel vs 2.6.13 (Debian): shared by all the nodes

[2] Non mentioned items have the same setup than the previous resource.

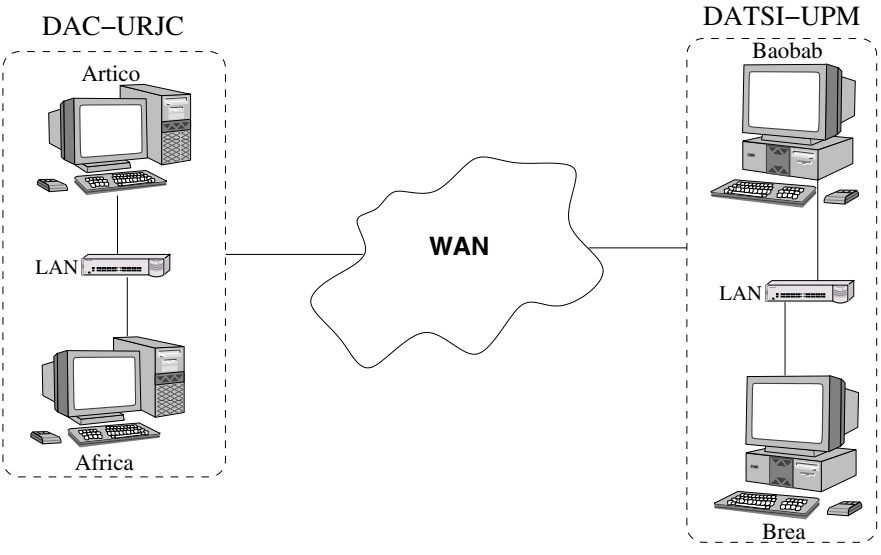


Fig. 1. Scheme of the grid used in the experiments

- internal network with 2 network interfaces per node with a bandiwth of 1Gb per interface

Brea: One biprocessor node with the following features:

- 2 CPUs Intel(R) Xeon(TM) CPU 3.00GHz
- cache size: 1024 KB (each CPU)
- hard disk:
  - local: 9,2 GB
  - NFS: Intel(R) Pentium(R) 4 CPU 2.80GHz. NAS. Raid 0 over 4 disks with 160 Gb.

Figure 1 shows the grid described above. Dashed lines group the computational resources available in each VO.

4.2 Performance Results

Table 1 collects the response user time per grid node considering different sizes of local database stored in each one of the grid nodes. The values are measured in seconds. Each node of the grid stores the same amount of images for the same row of the table, and the value gives the total amount of time spent by each node of the grid to give an answer to the user. It can be observed the wide range of values obtained because of the heterogeneous nature of the available processing nodes.

Table 2 shows the response user time of the grid (in seconds) considering different database sizes. This Table includes also the overhead introduced by the

Table 1. User response time of grid nodes for a query considering different local database sizes

Database size per grid node	Africa	Artico	Baobab	Brea
100000	1.63	6.81	0.98	0.87
1000000	17.30	84.36	11.58	9.96
2000000	33.89	173.94	23.05	20.55
4000000	68.45	390.63	56.73	44.39
8000000	136.75	816.54	125.50	88.48

Table 2. User response time for a query considering different database sizes with fair distribution of the images over the available grid nodes

DB size	Grid	Globus overhead	Efficiency
400000	29.25	22.44	0.23
4000000	106.00	21.64	0.80
8000000	202.89	28.95	0.86
16000000	420.21	29.58	0.93
32000000	846.16	29.62	0.96

grid implementation, which remains almost constant in all cases. Finally, the efficiency reached by the system is also included, computed as:

$$E_n = S_n/n \tag{1}$$

where

$$S_n = T_1/T_n \tag{2}$$

and T_1 is the execution time of the algorithm running on the slowest node of the grid and T_n is the execution time of the algorithm carried out over n nodes of the grid. As can be noticed, the efficiency of the grid increases as the size of the database grows up. This fact is explained by the almost constant overhead introduced by the grid implementation and the lack of data dependencies in the most demanding stage of the posed CBIR application that allows a fully parallelization of the signature comparison and sorting. For small database sizes (400000 images), efficiency values are quite low in comparison with values achieved for greater database sizes. The reason for this behaviour is that query response times of nodes are smaller than time overhead values introduced by the grid implementation and therefore the efficiency is weakened.

The grid overhead is due to several reasons like:

- Management of the auxiliary temporal files needed by the implemented algorithm.

- Globus overhead that includes:
  - Security control mechanisms.
  - State system management.
- Network traffic.

The grid herein presented allows a dynamic management of the CBIR system since Globus provides a set of service implementations focused on infrastructure management. Specifically, `GRAM` (Grid Resource Allocation Management) supports the control of the available resources in the grid at a given moment.

The values of Table 1 suggest a redistribution of the workload among the grid nodes considering the response times achieved in the experiments, instead of considering an homogeneous size of the image databases for all the grid nodes. Considering a static assignment of the workload, the redistribution should be made as a function of the response time of each node in relation with the slowest node of the grid:

$$DBs_i = \frac{T_1 \cdot DBs_1}{T_i} \tag{3}$$

where T_1 is the execution time of the algorithm running on the slowest node of the grid, T_i is the execution time of the algorithm on the node i of the grid and DBs_1 is the database reference size in the slowest grid node.

5 Conclusions and Future Work

This paper is focused on the analysis of the feasibility to apply a grid solution to CBIR systems. In this work we have measured the efficiency reached by the grid considering a real environment with very heterogeneous nodes. The implementation has been very satisfactory, achieving small user response times. This has been originated by the lack of data or algorithmic dependencies and to the small communication overhead. Thanks to the heterogeneity of the system, the communication overhead overlaps with the execution time in other grid nodes. These features result also in very good performance figures for the largest databases, where efficiency values higher than 90% have been achieved and the efficiency curve shows a tendency to approximate to one. The experiments presented here show that the amount of overhead introduced by this implementation is almost constant, so the system is scalable with respect to the database size. In fact, this overhead is hidden by the improvements achieved to take into account grid heterogeneity. To conclude, definite advantages of the grid implementation are its good price-performance ratio and system scalability.

Finally, further work will be devoted to the analysis on the response of the system after distributing the database of the CBIR system on different Virtual Organizations that include more complex grid nodes, like shared-memory multiprocessors or clusters of PCs or workstations. We also plan to incorporate load balancing mechanisms to dynamically redistribute the workload corresponding to the sorting stage and increase the global performance of the grid.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Education and Science (grant TIC2003-08933-C02) and Government of the Community of Madrid (grants GR/SAL/0940/2004 and S-0505/DPI/0235).

References

1. Berman, F., Fox, G., Hey, A.J., eds.: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons (2003) ISBN 0-470-85319-0.
2. del Bimbo, A.: Visual Information Retrieval. Morgan Kaufmann Publishers, San Francisco, California (1999) ISBN 1-55860-624-6.
3. Smeulders, A.W.M., Worring, M., Santini, S., Gupta, A., Jain, R.: Content-based image retrieval at the end of the early years. IEEE Transactions on PAMI **22**(12) (2000) 1349–1380
4. Pitas, I., ed.: Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks. John Wiley & Sons (1993)
5. Maresca, M., et al.: Special issue on parallel architecture for image processing. Proceedings of the IEEE **84**(7) (1996) 913–1056
6. Muntz, R.R., Golubchik, L.: Parallel data servers and appications. Parallel Computing **24** (1998) 1–4
7. Zaki, M.J.: Parallel and distributed association mining: A survey. IEEE Concurrency **7**(4) (1999) 14–25
8. Srakaew, S., Alexandridis, N., Nga, P.P., Blankenship, G.: Content-based multimedia data retrieval on cluster system environment. In Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., eds.: High-Performance Computing and Networking. 7^{th} International Conference, HPCN Europe 1999, Springer Verlag (1999) 1235–1241
9. Bosque, J.L., Robles, O.D., Rodríguez, A., Pastor, L.: Study of a parallel CBIR implementation using MPI. In Cantoni, V., Guerra, C., eds.: Proceedings on International Workshop on Computer Architectures for Machine Perception, IEEE CAMP 2000, Padova, Italy (2000) 195–204 ISBN 0-7695-0740-9.
10. Rodríguez, A., Robles, O.D., Pastor, L.: New features for Content-Based Image Retrieval using wavelets. In Muge, F., Pinto, R.C., Piedade, M., eds.: V Iberoamerican Simposium on Pattern Recognition, SIARP 2000, Lisbon, Portugal (2000) 517–528 ISBN 972-97711-1-1.
11. Robles, O.D., Rodríguez, A., Córdoba, M.L.: A study about multiresolution primitives for content-based image retrieval using wavelets. In Hamza, M.H., ed.: IASTED International Conference On Visualization, Imaging, and Image Processing (VIIP 2001), Marbella, Spain, IASTED, ACTA Press (2001) 506–511 ISBN 0-88986-309-1.
12. Stollnitz, E.J., DeRose, T.D., Salesin, D.H.: Wavelets for Computer Graphics. Morgan Kauffman Publishers, San Francisco (1996)
13. Foster, I.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: IFIP International Conference on Network and Parallel Computing. Volume 3779 of Lectures Notes in Computer Science. Springer Verlag (2005) 2–13
14. Robles, O.D., Toharia, P., Rodríguez, A., Pastor, L.: Towards a content-based video retrieval system using wavelet-based signatures. In Hamza, M.H., ed.: 7th IASTED International Conference on Computer Graphics and Imaging - CGIM 2004, Kauai, Hawaii, USA, IASTED, ACTA Press (2004) 344–349 ISBN: 0-88986-418-7, ISSN:1482-7905.

Performance Evaluation of Group Communication Architectures in Large Scale Systems Using MPI

Kayhan Erciyes[1], Orhan Dagdeviren[1], and Reşat Ümit Payli[2]

[1] Izmir Institute of Technology
Computer Eng. Dept., Urla, Izmir 35430, Turkey
{orhandagdeviren, kayhanerciyes}@iyte.edu.tr
[2] Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology
Indiana University-Purdue University
Indianapolis, Indiana 46202, U.S.A.
rpayli@iupui.edu

Abstract. Group communication is an important paradigm for fault tolerance in large scale systems. We describe various group architectures as pipelined, hierarchical, daisy and hypercube groups each consisting of separate clusters, investigate the theoretical performance bounds of these architectures and evaluate their experimental performances using MPI group communication primitives. We first derive time bounds for multicast message deliveries in these architectures and then provide tests to measure the times taken for the same operation. The multicast message delivery times are tested against the number of clusters within a group and the size of the multicast message. We conclude that daisy architecture is favorable both in terms of delivery times and message sizes theoretically and experimentally.

1 Introduction

A group is a logical name for a set of computing elements whose membership may change with time. Replication using process groups for fault tolerance has attracted many researchers for many years [1,2,3]. There are several systems which provide fault tolerant group communication such as Transis [4], Horus [5] and Totem [6]. Moshe [7] extends these services to a WAN. The common goal of these projects is to provide a reliable multicast communication for process groups.

MPI is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementors and users [8,9]. *MPI* provides enhanced group communication primitives [10]. An *MPI* communication operation always specifies a communicator. This identifies the process group that is engaged in the communication operation and the context in which the communication occurs. In this study, we investigate and evaluate Group Communication

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1422–1432, 2006.
© Springer-Verlag Berlin Heidelberg 2006

primitives in the pipelined, daisy, hierarchical and hypercube architectures using *MPI*. Background on process groups and *MPI* Group Communication primitives is reviewed in section 2. Sections 3, 4, 5, 6 briefly describe the architectures with the test results obtained. Finally a comparison of the results with the discussions is presented in Section 7.

2 Background

2.1 Group Communication

Replication is a common approach to achieve fault tolerance in a distributed system such that replicas provide redundancy in case of a failure of a server. Two main classes of replication are the *active* and *passive* replications. In passive replication, client deals only with one replica and the primary sends messages to the secondaries to update their views. A client sends a message to all of the replicas in active replication and the states of the replicas are maintained as identical, in general, using finite state machines. To ensure consistency of the replicas, a group communication primitive called the Total Order Multicast may be used which guarantees that the requests by the clients are received by all replicas in the same order. The group management module should also provide the two primitives; *send_multicast* to send a message to all members and *receive_multicast* to receive a message sent by a member of the group. These two primitives can be realised using various approaches such as *reliable broadcast, reliable FIFO broadcast and total order multicast. Reliable Broadcast* of a message in a group ensures that messages are delivered by all processes or none.

2.2 MPI Group Communication Primitives

In MPI, process groups allow a subset of processes to communicate among themselves using local process identifiers and to perform collective communication operations without involving other processes. The context forms part of the envelope associated with a message. A receive operation can receive a message only if the message was sent in the same context. Hence, if two routines use different contexts for their internal communication, there can be no danger of their communications being confused. All communication operations have used the default communicator *MPI _COMM_WORLD* , which incorporates all processes involved in an *MPI* computation and defines a default context. A set of group communication routines which are used for creating new groups and communicators, multicasting messages to all process in a communicator is defined in *MPI* is summarized below:

- *MPI_Comm_group* : Function *MPI_Comm_group (MPI _Comm comm, MPI _Group *group)* returns the process group associated with the communicator.
- *MPI_Comm_create*: The collective function *MPI_Comm_create (MPI_Comm old_comm, MPI_Group group, MPI_Group *new_comm*) creates a new communicator from the processes listed in *group*.

- *MPI_Comm_split* : Collective function *MPI_Comm_split (MPI_Comm old _comm, int partition, int new_rank, MPI_Comm *new_comm)* partitions the processes in an existing communicator(*old_comm*) into one or more subgroups.
- *MPI_Group_incl* : Function *MPI_Group_incl (MPI_Group old, int new_size, int *old_ranks, MPI_Group *new*) produces a new group from an existing group. The size of the the new group is specified by parameter *new_size*.
- *MPI_Barrier* : *MPI_Barrier*(*MPI_Comm*) is a collective communication function that performs a barrier synchronization among all processes in the specified communicator.
- *MPI_Bcast* : Function *MPI_Bcast (void *buffer, int cnt, MPI_Datatype dtype, int root, MPI_Comm comm)* is a collective communication operation allowing one process to broadcast a message to all other processes in a communicator [11].

3 Pipelined Architecture

Pipelined architecture is constructed by the serial arrangement of groups which includes equal size of processors. Each group has a leader which is responsible to send and receive messages from another group and to multicast messages to its group elements. A pipelined architecture with 24 processors and 4 groups is shown in Fig. 1. Each group has 6 processors in which 0, 6, 12 and 18 are the group leaders.

Pipelined architecture can be constructed by row wise partitioning of the *MPI_COMM_WORLD* communicator. Leaders of each group can send messages using *MPI_Send* and *MPI_Recv* primitives in *MPI_COMM _WORLD* communicator.

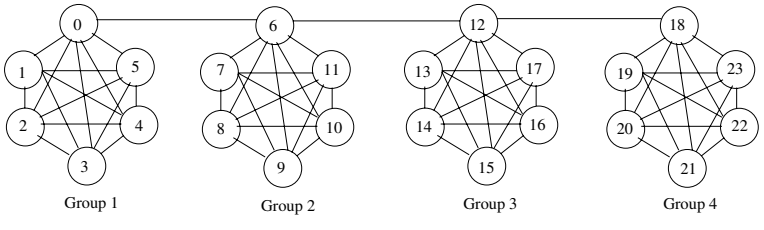


Fig. 1. Pipelined Architecture with 24 processors and 4 groups

Theorem 1. *Multicast communication in pipelined architecture takes* $\Theta(mk)$ *time where* m *is an upperbound on the number of clusters and* k *is an upperbound on the number of processors in a cluster,*

Proof. Assume for simplicity, each multicast operation within a cluster of the pipelined group will consist of k one-to-one communications. Each of these muticast communications is performed sequentially, one after each other along the

pipeline. Therefore, we would have a total of $m * k$ message deliveries within the clusters and $m-1$ messages among the clusters, resulting in a total of $mk+m-1$ steps to deliver the multicast message to all group members. The total time can therefore be approximated by $\Theta(m(k+1)-1) \sim \Theta(mk)$

We performed the tests on the 120 processors of the AVIDD cluster of Indiana University, U.S.A. AVIDD is an IBM eSeries cluster which consists of 96 nodes. Each node has 2 2.4GHz Intel Pentium 4 Xeon CPUs and a 2.5GB memory with the Myricom Myrinet interconnect. Peak performance of the cluster is 0.55 Tera Flops. One-to-all broadcast performance of pipelined group in this environement is measured with respect to message size and total group number. In Fig. 2, message size is varied from 4 bytes to 40K bytes, processor number is fixed at 24 and the total group count is varied between 2 and 8. A pipelined architecture with 8 groups results in the minimum mean time of 806 microseconds for a group multicast operation.

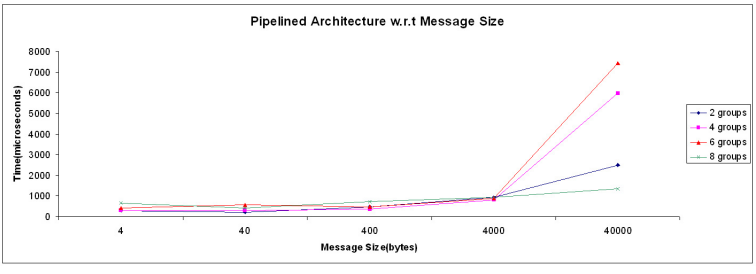


Fig. 2. Pipelined Architecture Multicast Run-times against Message Size

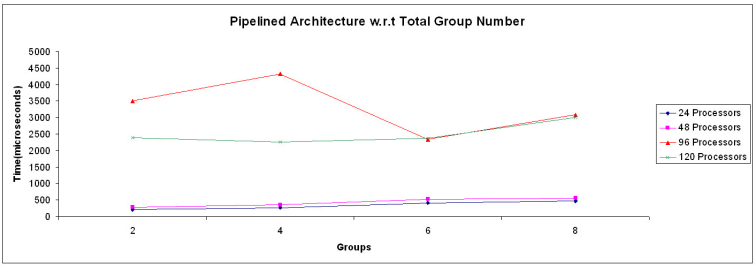


Fig. 3. Pipelined Architecture Multicast Run-times against Total Group Number

In Fig. 3 total group number is varied from 2 to 8 groups, message size is fixed at 4 bytes and the total processor number is varied from 24 to 120. Pipelined architectures with 24 and 48 processors seem to finish one-to-all broadcast operation in approximate time values but a sharp increase occurs when we double 48 processors to 96 processors.

4 Daisy Architecture

Daisy architecture is constructed by a number of groups in which leaders are connected to form a group which overlaps with all other groups. Each group leader is responsible to multicast messages to its group. A daisy architecture with 24 processors and 5 groups are shown in Fig. 4.

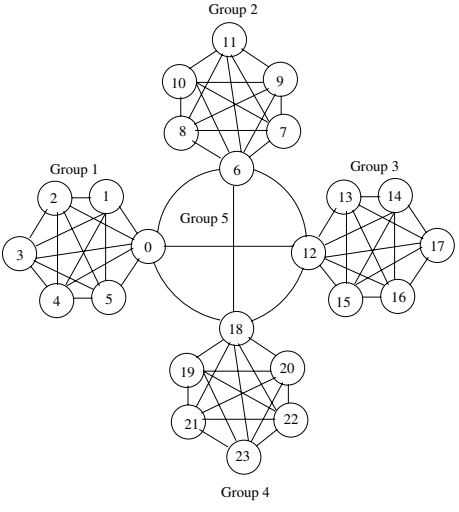


Fig. 4. Daisy Architecture with 24 processors and 5 groups

Theorem 2. *Multicast communication in the daisy architecture takes* $O(3k)$ *time where* k *is an upperbound on the number of processors.*

Proof. A multicast operation within a cluster of the daisy group will take k one-to-one communications to reach the central cluster. There will be k one-to-one mesages by the central cluster and another k mesages by the outer clusters which are performed in parallel resulting in a total of $O(3k)$ message times.

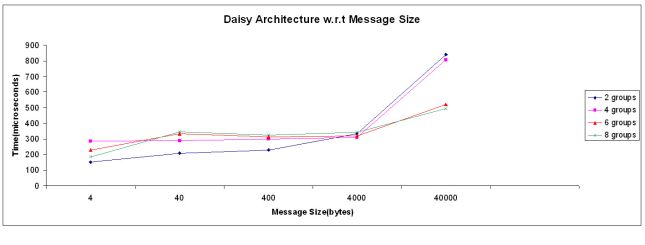


Fig. 5. Daisy Architecture Multicast Run-times against Message Size

It should be noted that the message delivery time of the daisy architecture is independent of the number of clusters and the final delivery in the outer clusters are performed as parallel multicast communications. Daisy architecture can be constructed by row wise and column wise partitioning of the *MPI_COMM_WORLD* communicator. Column wise communicators except group leaders's communicator are destroyed by *MPI_Comm_free*. Experimental setup is the same as in pipelined test. A daisy architecture with 8 groups results in the minimum mean time of 338 microseconds which is shown in Fig. 5. In Fig. 6 when we double the 48 processors a sharp increase occurs.

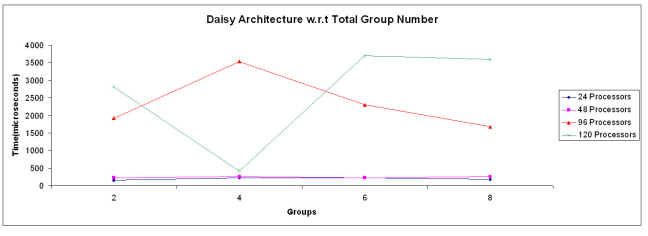


Fig. 6. Daisy Architecture Multicast Run-times against Total Group Number

5 Hierarchical Architecture

Hierarchical Architecture is constructed by different levels of groups which can contain different number of processors as in [12]. A hierarchical architecture with 24 processors and 7 groups is shown in Fig. 7. Level 1, Level 2 and Level 3 groups contains 3,4 and 4 processors respectively. Group leaders are responsible to exchange messages between upper level of connected groups. Processor groups can be created by MPI_Comm_group and MPI_Group_incl. Communicators of these groups can be created by MPI_Comm_create.

Theorem 3. *Multicast communication in the hierarchical architecture takes* $O(2kl)$ *time where* l *is an upperbound on the number of levels of hierarchy.*

Proof. In the worst case, it would take $2l-1$ steps for a multicast message to reach the farthest node where l is the count of levels of the hierarchy. Since there are k nodes in each cluster, there would be $O(2k(l-1)) \sim O(2kl)$ unicast messages to accomplish a multicast communication in the worst case.

Corollary 1. *For the special case of a binary tree of the hierarchical architecture, multicast communication takes* $O(2k \log m)$ *time.*

Proof. For the binary tree of the hierarchical architecture, there would be a total of $2^l - 1$ tree nodes meaning $m = 2^l - 1$. We can therefore approximate $l = \log m$. The total count of multicast communication steps is $2l - 1$. Each multicast communication has k unicast communications resulting in a $O(k * (2l-1))$ steps for the total multicast. Substituting for l in the complexity equation yields $\sim O(2k \log m)$ for the total time of multicast in the binary tree in the worst case.

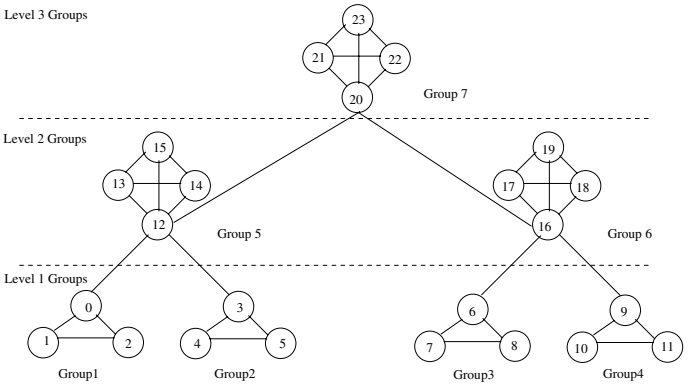


Fig. 7. Hierarchical Architecture with 24 processors and 7 groups

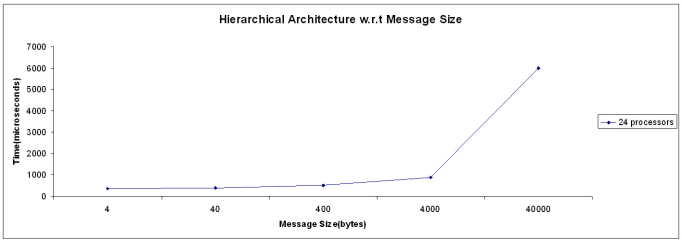


Fig. 8. Hierarchical Architecture Multicast Run-times against Message Size

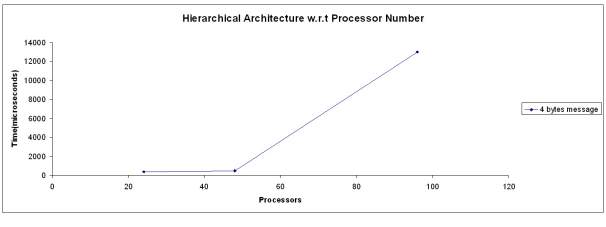


Fig. 9. Hierarchical Architecture Multicast Run-times against Total Processor Number

In Fig. 8 one-to-all broadcast times are measured against message size on a binary tree with 24 processors and 7 groups and 1625 microseconds is the mean time. In Fig. 9 one-to-all broadcast times are measured with respect to total processor number with fixed 7 groups and 4 bytes messages. Processor number is varied between 24 and 96. A significant increase happens when we double the processor number to 96 processors.

6 Hypercube Architecture

A hypercube architecture is a widely used architecture for parallel/distributed applications [13]. A two-dimensional hypercube of four processors is constructed from one-dimensional hypercubes by connecting corresponding nodes. In general a $d-$dimensional hypercube is constructed by connecting corresponding nodes of two $(d-1)$ dimensional hypercubes. A hypercube architecture with 24 processors and 8 groups is shown in Fig. 10. One-to-all broadcast in hypercube topology can be implemented by modifiying the procedure in [14].

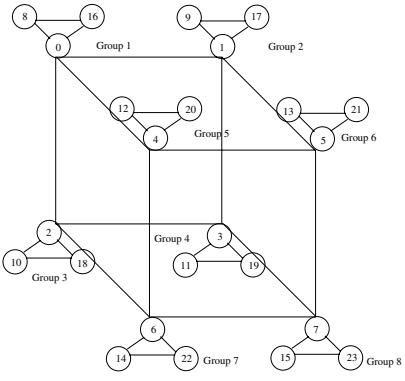


Fig. 10. Hypercube Architecture with 24 processors and 8 groups

Theorem 4. *Multicast communication in the hypercube architecture takes* $O(k \log m)$ *time where* m *is an upperbound on the number of clusters (number of hypercube vertices) and* k *is an upperbound on the number of processors in the hypercube vertex cluster.*

Proof. A one-to-all communication in a hypercube of m nodes takes $\log m$ time. Since a one-to-all communication of $O(k)$ is performed within each hypercube vertex cluster prior to sending the message along another dimension, we have $O(k \log m)$ message times ignoring the times taken for the single message deliveries along the edges of the hypercube.

One-to-all broadcast performance in a hypercube architecture with 24 processors and 3 dimensions is obtained with respect to message size in Fig. 11 and a mean time of 1168 microseconds is measured.

We also measured the one-to-all broadcast performance with respect to processor number and dimension in Fig. 12. A hypercube architecture with 3 dimensions can be realized with 24, 48, 96 and 120 processors, 4 dimensions can be realized with 48 and 96 processors and 5 dimensions can be realized with 96 processors. A sharp increase can be seen when we double the 48 processors in Fig. 12.

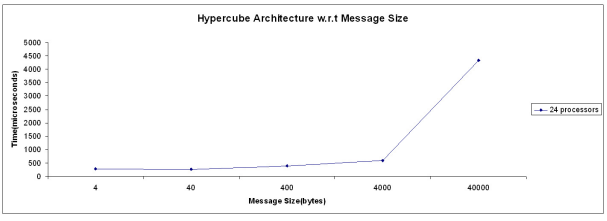


Fig. 11. Hypercube Architecture Multicast Run-times against Message Size

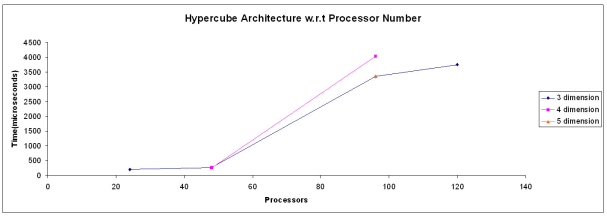


Fig. 12. Hypercube Architecture Multicast Run-times against Total Processor Number

7 Discussions and Evaluations

We evaluated various group communication architectures in a cluster of 120 processors using MPI. The groups in the tests were connected in pipelined, daisy, hierarchical and hypercube architectures. In Fig. 13, architectures are compared with respect to message size with a constant total number of processors of 24. In Fig. 14, architectures are compared with varying total number of processors.

The comparison of message complexities and the message delivery times of these architectures are displayed in Table 1. It can be seen that the daisy architecture provides the minimum multicast message delivery times for both the fixed number of processors and the fixed size of message cases as expected. Theoretically, the next best performance should have been the hypercube and

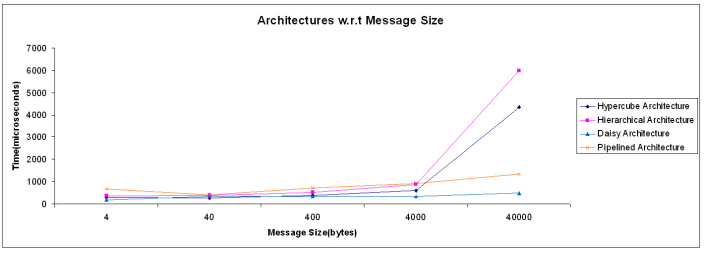


Fig. 13. Architectures Multicast Run-times Comparison against Message Size

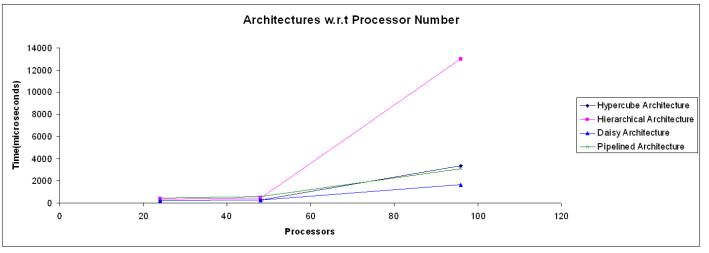


Fig. 14. Architectures Multicast Run-times Comparison against Total Processor Count

Table 1. Comparison of the Group Architectures

Topology	Time Complexity	Fixed Proc.(t) (μs)	Fixed Msg. Size(t) (μs)
Pipelined	$O(km)$	1776	806
Daisy	$O(3k)$	1429	338
Hierarchical(BT)	$O(2k \log m)$	4610	1625
Hypercube	$O(k \log m)$	1888	1168

the pipelined architecture should have provided the worst performance but the experimental results did not wholly support this view. Pipelined architecture resulted in similar performance to the hypercube. A larger cluster count may affect the results. Based on the analysis, we expected that the binary tree architecture should have twice longer times than the hypercube which conforms to the tests.

Our general conclusion is that the daisy architecture is favorable for multicast communication using *MPI* in terms of message delivery times and message sizes. We are planning to use the daisy architecture based group communication in the Grid architecture [15] where nodes do fail frequently and hence fault tolerance using replication is an important paradigm to provide a reliable service.

References

1. Birman K. P., van Renesse, R., Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, Los Alamitos, Ca., (1994).
2. Chockler, G, Keidar, I., Vitenberg, R., Group communication specifications: a comprehensive study, ACM Computing Surveys, (2001), 33(4) , 427-469
3. Cristian F., Synchronous and Asynchronous Communication, Communications of the ACM. Special Section on Group Communication, 1996, 39(4)
4. Y.Amir et all, Transis: A communication subsystem for high availability. Proc. of 22nd IEEE Int'l Symp. on Fault-Tolerant Computing, IEEE Press, NJ, 76-84
5. Van Renesse R., Birman K. P.,Maffeis S., Horus : A Flexible Group communication System, CACM, Special sect. on Group Comm., (1996), 39(4)
6. Y.Amir et all, The TOTEM Single Ring Ordering and membership Protocol, ACM Trans. Comp. Systems., 1995, 13(4)

7. Keidar, I. et al, Moshe: A group membership service for WANs, ACM Transactions on Computer Systems (TOCS) (2002), 20(3) , 191-238
8. Gropp, W., Lusk, E., Doss N. and Skjellum, A. : A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, MPI Developers Conference, (1995)
9. Squyres, J. M., Lumsdaine, A., George, W.L., Hagedorn, J.G. and Devaney, J.E., : The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI, MPI Developers Conference, Ithica, NY, (2000)
10. Yuan, X., Daniels, S., Faraj, A., Karwande, A. : Group Management Schemes for Implementing MPI Collective Communication over IP Multicast, The 6th Int. Conf. on Computer Science and Informatics, Durham, NC, (2002), 8-14
11. Quinn, M. J., : Parallel Programming in C with MPI and OpenMP, International Edition, Mc Graw Hill, (2003).
12. Tunali, T, Erciyes,K., Soysert, Z.: A Hierarchical Fault-Tolerant Ring Protocol For A Distributed Real-Time System, Special issue of Parallel and Distributed Computing Practices on Parallel and Distributed Real-Time Systems, (2000), 2(1), 33-44
13. Allahverdi, N, Kahramanli, S, Erciyes,K. : A Fault Tolerant Routing Algorithm Based on Cube Algebra for Hypercube Systems , JSA, 2000, 46(2), 201-205
14. Grama, A., Gupta, A., Karypis, G., Kumar, V. : Introduction to Parallel Computing, Second Edition, Addison Wesley Longman, Inc., (2003).
15. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Int. Journal of High Performance Computing Applications, (2001), 15(3), 200-222

DOA 2006 International Conference (Distributed Objects and Applications) PC Co-chairs' Message

Welcome to the Eighth International Symposium on Distributed Objects and Applications (DOA 2006). This year DOA was held in Montpellier, France, from October 30 to November 1, 2006.

The DOA conferences provide a forum for exchanging the latest research results on distributed objects, components, services, and applications. Research in distributed objects, components, services, and applications establishes new principles that open the way to solutions that can meet the requirements of tomorrow's applications. Conversely, practical experience in real-world projects drives this same research by exposing new ideas and posing new types of problems to be solved. With DOA 2006 we explicitly intended to provide a forum to help this mutual interaction occur, and to trigger and foster it. Submissions were therefore welcomed along both these dimensions: research (fundamentals, concepts, principles, evaluations, patterns, and algorithms) and practice (applications, experience, case studies, and lessons). Contributions attempting to cross over the gap between these two dimensions were particularly encouraged. Towards this goal, we accepted both research and experience papers.

The resulting high-quality program would not have been possible without the authors who chose DOA as a venue for their publications. All papers were submitted to a rigorous reviewing process with three reviews per paper, and considerable discussion took place among the Program Committee before decisions were taken. Out of 72 submitted papers we finally selected 24 full papers and 11 posters. Our congratulations and thanks to the successful authors, who came from many parts of the globe to present their work in Montpellier.

To round up this excellent program, Alois Ferscha from the Johannes Kepler Universität Linz, agreed to be our keynote speaker, with a presentation on "Every Object in the Pervasive Computing Landscape."

We are grateful for the dedicated work of the 56 top experts in the field from all over the world who served on the Program Committee, and whose names appear in this Volume. Special thanks go to the external referees who volunteered their time to provide additional reviews. Finally, we are indebted to Kwong Yuen Lai, who was immensely helpful in facilitating the review process and making sure that everything stayed on track.

August 2006

Judith Bishop, University of Pretoria, South Africa
Kurt Geihs, University of Kassel, Germany

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, p. 1433, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Everyobjects in the Pervasive Computing Landscape

Alois Ferscha

Johannes Kepler Universitat Linz
ferscha@soft.uni-linz.ac.at

Abstract. Information and communication technologies seamlessly and invisibly pervade into everyday objects and environments, delivering services through the 'things' (artefacts) of the physical world, and adapted to the context of their use. As an approach to link each artefact in the physical world with the software objects in the digital world we propose 'everyobjects', the symbiosis of technology rich artefacts and a P2P based coordination framework.

A spatial proximity based spontaneous interaction approach for 'shared nothing' P2P systems has been developed and implemented in our P2P coordination framework. Mobile peers, once in spatial proximity to each other, coordinate their activities based on the exchange and analysis of self descriptions in metadata format. Each peer individually conducts a similarity analysis of the role and profile data received from other peers in vicinity, and invokes triggers if a certain similarity threshold is reached. I will present aspects of the underlying profile description language expressing peer preferences with respect to the particular context in which the interaction is attempted (like time, geoposition, owner, environmental conditions, etc.), the mathematical formulation of similarity metrics and the matching analysis of semistructured data, the mechanisms for consensus finding and coordination in peer ensembles, and the component model within which the framework is implemented. Findings will be demonstrated in scenarios involving our 'Peer-it' miniaturized hardware platform, serving as a stick-on solution for artefacts.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, p. 1434, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Using Selective Acknowledgements to Reduce the Memory Footprint of Replicated Services*

Roy Friedman[1] and Erez Hadad[2]

[1] Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
roy@cs.technion.ac.il
[2] IBM Haifa Research Labs
Haifa University
Mt. Carmel, Haifa 31905, Israel
erezh@il.ibm.com

Abstract. This paper proposes the use of *Selective Acknowledgements* (SACK) from clients to services as a method for reducing the memory footprint of replicated services. The paper discusses the general concept of SACK in replicated services and presents a specific implementation of SACK for an existing replication infrastructure. Performance measurements exhibiting the effectiveness of SACK are also presented.

1 Introduction

Context of this study: With our growing dependence as a society on computing infrastructure, fault-tolerance and high availability have become implicit requirements from any service. Yet, a grand challenge is how to provide the required levels of fault-tolerance and high availability at affordable costs. One of the main techniques for obtaining these goals is by using replication, as it allows harnessing together standard hardware and software components, thereby enjoying the economy of scale in terms of price/performance.

At the same time, such fault-tolerant implementations must take care to handle consistency and invocation semantics correctly. Specifically, in order to ensure the availability of the service to the client, a client may time out on a request to a server (or a set of servers), in which case the client will reissue the same request. Notice that such a time-out and reissuing of the request can be caused either due to real server failure, or due to networking problem. Either way, it is possible that the request has already been executed by the server, and it is only the reply that did not reach the client on time. Thus, in order to provide an *at-most-once* invocation semantics, it is important to be able to detect when the original request was already executed by some server. The server (or servers) receiving the reissued request must then return the reply that was

* This research was partially supported by an IBM Faculty Award. Most of the hardware used for the performance measurements was donated to our lab by IBM.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1435–1448, 2006.
© Springer-Verlag Berlin Heidelberg 2006

generated for the original request and avoid re-executing it.[1] This is typically obtained through a *request/reply* cache (or a request log).

Hence, the request/reply cache (or request log) is a fundamental component of server replication [3,19,22] and also of other distributed services [24,28] that need to maintain *at-most-once* request execution safety. The cache stores client requests received by the replicas of the service. When a client issues a request to a server, the server checks the request ID at the cache to determine whether it, or some other replica of the service, has already processed this request. If so, the server returns the matching stored reply instead of re-processing the request. This ensures that a request is processed at most once.

Due to the need to store all requests and replies, the cache is often one of the most memory-consuming elements of a replicated server. Excessive cache growth might eventually impede performance due to memory swapping and, in languages like Java, dynamic memory management overhead. Furthermore, once memory is exhausted, servers might crash and the service might be disrupted[2]. Hence, slimming down the cache may increase both the replicated service's performance and scalability. Of course, it is possible to store some of the cache's contents on disk in order to save memory, but this entails a significant latency.

In most client/server middleware standards, clients may perform multiple independent invocations on a server object using different threads or asynchronous invocations. As a result, a new request arriving from the same client machine cannot be assumed to implicitly acknowledge receiving replies for any previous requests that have finished processing. Furthermore, most middleware implementations do not provide the server with any indications of a successful delivery of replies to clients. Thus, the common method of eliminating old requests from the cache is based on a globally known time-limit after which the request is assumed not to be re-issued.

Several ad-hoc solutions might be considered for reducing the cache size. Shortening the timeout could be effective, yet below a certain point, it might compromise the ability to guarantee at-most-once semantics. Also, adding purging cycles might remove requests with greater time-out accuracy but would also require more CPU, which is likely to degrade performance without considerably reducing the cache since the time-out remains the limiting factor. Hence, we considered adding information to the purging process to make it possible to remove stored requests from the cache even before the timeout without violating safety conditions of at-most-once and at-least-once request execution.

Contribution of this work: In this work we propose to use a selective acknowledgement (SACK) mechanism in which clients notify servers about which replies they have

[1] Some solutions involve the client joining a reliable multicast group with the server, typically implemented with a group communication toolkit [14,17]. This can eliminate the need for reissuing requests. However, such approaches suffer from serious scalability problems; either a new group must be created for each client, or a new view must be generated for each client entering and leaving the system. Both options are very costly and do not scale well in the number of clients.

[2] In fact, the motivation for working on this problem began after noticing that the servers running a replication infrastructure we developed, called FTS, crash-fail under high loads due to excessive memory requirements.

already received. These acknowledgements are typically piggy-backed on existing messages, thereby rarely generating traffic of their own.

We also describe how SACK is implemented within a replication infrastructure we have developed, called FTS, for CORBA objects. In this implementation, SACK is completely transparent to the client's application, as is the rest of FTS.

Finally, we present a performance analysis that exhibit the benefits of using SACK. It shows a significant drop in the memory footprint, and even a slight improvement in throughput.

The rest of this paper is organized as follows: Section 2 presents the implementation of SACK, both as a general concept and the specific implementation in FTS. Section 3 includes the performance evaluation, and we compare SACK with related ideas in Section 4. Finally, we conclude with a discussion in Section 5.

2 Implementing SACK

2.1 Basic Idea

As discussed above, the basic idea of SACK is fairly simple: have each client send to the server(s) the list of requests ids for which this client has already received a reply. When the servers receive such a message, they can eliminate the corresponding requests and replies from their cache. However, there are several additional issues and pitfalls that need to be handled, as we discuss below.

Ideally, the SACK mechanism, like the rest of the fault-tolerant infrastructure, should be as transparent to the application as possible. Several middleware standards, such as CORBA and Microsoft .NET, offer standard interception mechanisms (Portable Interceptors in CORBA [23,8] and Custom Sinks and Filters in .NET). These can be used to hide SACKs from the application's code. Below, we give a specific example of a CORBA implementation.

Another issue in replicated services is that the request and reply should be propagated to the caches of all servers. This is in order to handle a situation in which the re-issued request arrives at a different server. In active replication [25], where all servers execute the request, the request is propagated to all servers using some totally ordered delivery mechanism (ABCAST), e.g., [1,4,9,16]. Each server executes the request after receiving it from the ABCAST. Thus, implicitly, each cache includes the request. However, in primary/backup replications styles [5], the request is executed by only one server, and an explicit propagation of the request and reply to the caches of all other replicas is needed.

Yet, if care is not taken, a race between a SACK and the propagation of the original request to all replicas could occur, which may cause some servers to either never execute a request, or to drop the SACK. For this reason, SACK propagation should be made consistent with the actual requests propagation. More details appear below in Section 2.4.

2.2 Overview of FTS

We have implemented the SACK mechanism inside a CORBA replication infrastructure called FTS [11]. FTS is entirely written in Java, and operates according to *active*

replication, or *replicated state machine* [25], with a slight scalability twist: With FTS, a client connects to a single server using CORBA's standard IIOP (over TCP/IP) connection. Whenever a server receives a request, it first broadcast the request to all other replicas in total ordering (ABCAST), by relying on an underlying group communication toolkit [5]. Once the request is delivered from the group communication toolkit, it is locally executed by each of the servers (under the assumption of deterministic methods). After execution, only the server that holds the IIOP connection with the client sends the reply as a standard IIOP reply. All servers, however, keep the reply in a local cache in case the request is later resent by the client, e.g., if the client timed out on the reply.

FTS is designed to work with any virtually synchronous group communication system using a generic Group Communication Interface layer. Specifically, in the measurements reported in this paper, we have used Ensemble [13], which runs over UDP and is known to offer very good performance in LAN based clusters [7,13].

If the connection between a client and a server breaks, or a request is timed out, this is caught by a *Portable Interceptor* at the client side [2,8,10,23], which automatically redirects the request to an available server. The exact details of such client redirections are beyond the scope and interest of this paper (here we are interested in the benefit of batching), and can be found in [11]. The significant piece of information is that this is done completely transparently to the client's code, which can be any unmodified CORBA client application.

Due to various CORBA limitations, and in order to remain transparent to the server's code, the logic for handling client requests at the server side is implemented inside a *Group Object Adaptor* (GOA), as illustrated in Figure 1. An *object adaptor* is the CORBA entity that mediates between the request broker and the actual server implementation; it is responsible for locating the implementation of a service, loading the object and starting it if needed, and for invoking the actual request on the object's implementation. In FTS we have extended the standard *Portable Object Adaptor* to include the replication and consistency logic, to form a GOA. The exact details are very technical, and the reader is referred to [11] for more information.

The important aspect of the above design for this paper is that the SACK mechanism is implemented inside the Portable Interceptor (PI) on the client side and inside the GOA on the server side. The details appear below.

2.3 Implementing SACK in FTS

On the client side, SACKs are transparently handled by the FTS PI, which collects request IDs of replies and attaches them as SACKs to subsequent invocations that target the same object the replies were received from. This also ensures that SACKs of multiple independent invocations, such as by multiple client threads or using asynchronous invocations, are also collected and sent in a subsequent request from the same client to the same object. Thus, even if a client thread interacts with one service object and then with another, the SACKs that were collected from its invocations of the first object may still be sent along with a different thread's invocation.

Furthermore, FTS replicates multiple objects together in replication context groups. Accordingly, FTS also allows to accumulate SACKs according to the target replication

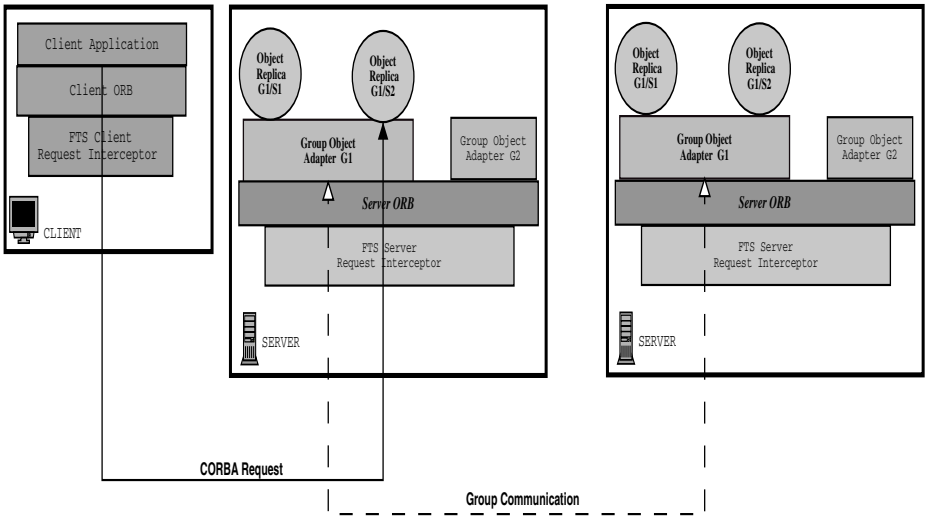


Fig. 1. FTS Architecture

context group (which may contain multiple objects) instead of only the target object in order to further increase SACK coverage. That is, if a thread invokes an object inside a particular context, the request carries SACKs of requests invoked by any thread at the same client that previously invoked an object in the same context.

On the server side, SACKs are handled inside a `SackHandler` object in each GOA. This object holds two SACK stores called `arrived` and `cast`. The `arrived` store contains SACKs that arrived from clients and are pending ABCAST. The `cast` store contains SACKs that were delivered back from ABCAST and need to be processed. The `SackHandler` object is operated by a dedicated daemon thread that periodically (once per second) collects and sends the `arrived` SACKs in batches, and then processes `cast` SACKs by locating their matching requests in the cache and tagging them as removable. The actual removal of a request takes place only after the request is done executing. After being processed, the `cast` SACKs are discarded. Further implementation details can be found in [12].

2.4 Rationale

In many common distributed applications, such as a database service, a directory service, etc., a client interacts with a service object using a sequence of invocations rather than a single invocation. Thus, subsequent invocations can be used by the client for acknowledging successfully-received replies of previous invocations. Consequently, on the server side, acknowledged requests can be discarded from the cache.

This concept resembles the technique used by TCP [27] and SACK TCP [18] to remove acknowledged segments from the sending window. However, unlike TCP or SACK TCP, a generic middleware SACK mechanism, such as inside FTS, cannot use

the more space-efficient sequence ACKs, only selective acknowledgments (SACKs). The reason is that in many cases, request IDs are only guaranteed to be unique, but not necessarily ordered. For example, this is the case with request IDs of FT-CORBA's `FTRequestServiceContext` [22]. Of course, a specific implementation of FT-CORBA might provide ordered request IDs, but others may not. Hence, with the lack of guaranteed ordering among request IDs, acknowledging one request ID cannot be used to acknowledge additional request IDs that might have preceded it.

Sending SACKs through the ABCAST transport is justified as following. FTS does not assume clock synchronization between a client and any server, so the client PI does not expunge collected SACKs that have become invalid due to expired time-out at the servers. Therefore, a SACK that arrives at a server may refer to a request that is not in the cache, either since that request has not been delivered yet or since it has already been removed from the cache. With the server being unable to determine which case, the SACK is simply discarded with no additional effect. However, in order to ensure that a SACK is not discarded before the request it refers to is delivered to the server, SACKs are not processed upon reception from a client, but rather are sent over the inter-server ABCAST transport and used only when delivered back from it. Since the SACK, by definition, acknowledges a request that has already been delivered in at least one server, the total ordering property of ABCAST ensures that the SACK is delivered after the matching request in all the servers. This further ensures that all the servers receive the SACKs and use them to shrink their caches.

Note, however, that this method of SACK processing also results in an overhead in the form of ABCAST messages that are not used for relaying requests among the servers. This overhead is both in bandwidth and in message processing (sending and receiving messages), and is added to the processing overhead of the SACK daemon thread. The SACK message processing overhead is aggressively reduced by batching multiple SACKs together into SACK ABCAST messages, using simple time-based batching with a window size of 1 second[3]. This delay is still much smaller than the request expiration limit which is set to 10 seconds.

From an operational correctness point of view, SACKs are merely an optimization to the existing timeout-based purging mechanism. If a request has not been processed by a server yet, it may be tagged as removable by a SACK but it will not be removed from the cache until is is done processing. Also, clients send SACKs only after receiving a correct reply so a request removed through a SACK is guaranteed not to be needed again. Last, SACKs that refer to non-existing requests are simply discarded with no other effect. Therefore, these operating rules do not violate the underlying safety conditions of each request being processed at least once as well as at most once by each server.

3 Performance Evaluation

We used FTS to test the performance impacts of the SACK implementation. The platform on which FTS was tested is an IBM JS20 Blade Center with 28 blades (machines). Each blade has two PowerPC 970 processors running at 2.2GHz with 4GByte RAM.

[3] Actually, in our tests, SACK batches are also limited to 5000 SACKs each due to Ensemble daemon settings.

The blades are divided into two 14-node pools, each internally connected with a Gbit Ethernet switch, and both switches are connected to each other. Additionally, the cluster includes an Intel X346 server with dual Pentium 4 Xeon 3.0GHz (hyper-threaded) with 2GByte RAM, used also as an NFS server for the common file system on which FTS is installed.

The blades and the server run Suse Linux Enterprise Server 9 (2.6.5 kernel) service pack 3. All FTS components run on IBM JVM v1.4.2 service pack 2 for PPC and Intel platforms, respectively. The ORB used is ORBacus for Java [15] v4.1.3. Ensemble [13] v2.01 is used for group communication.

FTS is deployed as following. Clients are running in blade pool 1 and servers in blade pool 2, so that all clients are equally distant from all the servers. Each client blade runs 250 clients sharing a common ORB. Each server blade runs a single server and a local Ensemble daemon. Last, the Intel server runs a *test manager*, whose purpose is to conduct a single test of clients vs. servers and an *executive* that generates sequences of tests, passes each test to the test manager and collects the results.

We tested SACK under varying amounts of client loads, ranging from 500 clients to 2500 clients. The server cluster consists of 4 servers. We used short requests, where the overhead of ABCAST bandwidth should be close to maximum. The invocations are of the `set_greeting()` method of the target Hello object with a string parameter of 15 characters.

We conducted two simple sets of tests, one with SACKs and one without SACKs. The measurements that we used to compare the results are FTS throughput, memory usage and overhead estimates. Memory usage data consists of JVM allocated memory and cache size (counted in request records). This data is averaged over a sequence of read-outs performed once every two seconds during the test phase. SACK overhead is estimated as SACK percentage of total traffic. Thus, SACK bandwidth overhead is estimated in byte percentage and SACK message processing overhead is estimated in message percentage. The processing overhead of the SACK thread is harder to evaluate, so we use the throughput measurement as a global indicator for the entire processing overhead.

Figure 2 compares the cache sizes between runs with and without SACKs. Figure 3 shows a similar comparison of total server memory consumption. Note that the scale of 3 is in percents of the JVM memory allocation limit. The top allocation limit is the same in all runs (1GB) but is not relevant and therefore omitted. Next, Figure 4 shows FTS throughput with and without SACKs. Last, Table 1 shows the SACK overhead of ABCAST traffic in bytes and messages.

3.1 Results Analysis

The results for the cache size, as shown in Figure 2 show that applying SACKs reduces the cache size by at least a factor of 2. In loads of 750 and 1000 clients the cache size is reduced by a factor of 3, which means that SACKs are indeed very effective.

An additional evidence of the positive impact of using SACKs is presented in Figure 4 that shows an improvement of about 20% in cluster throughput in high client loads. This means that despite the added processing and network overhead imposed by SACKs, the overall contribution improves not only server scalability but also

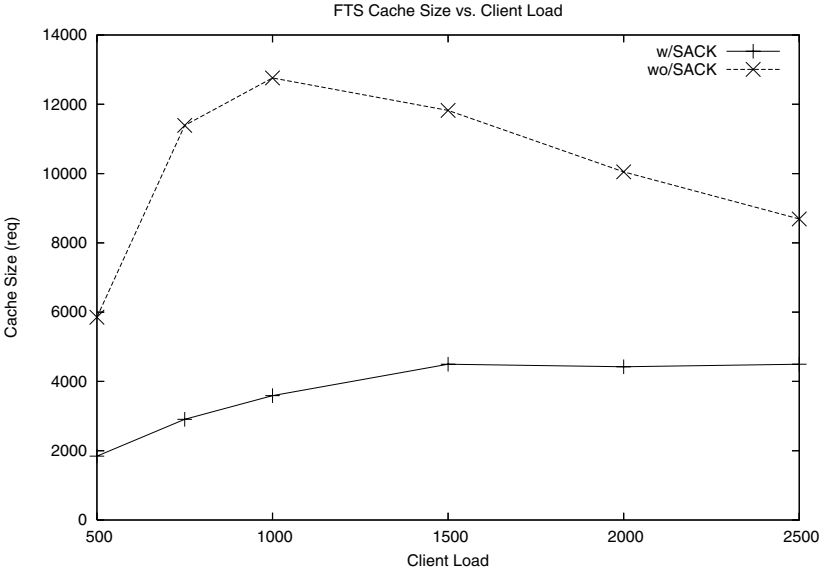


Fig. 2. Performance Evaluation: Cache Size vs. Client Load

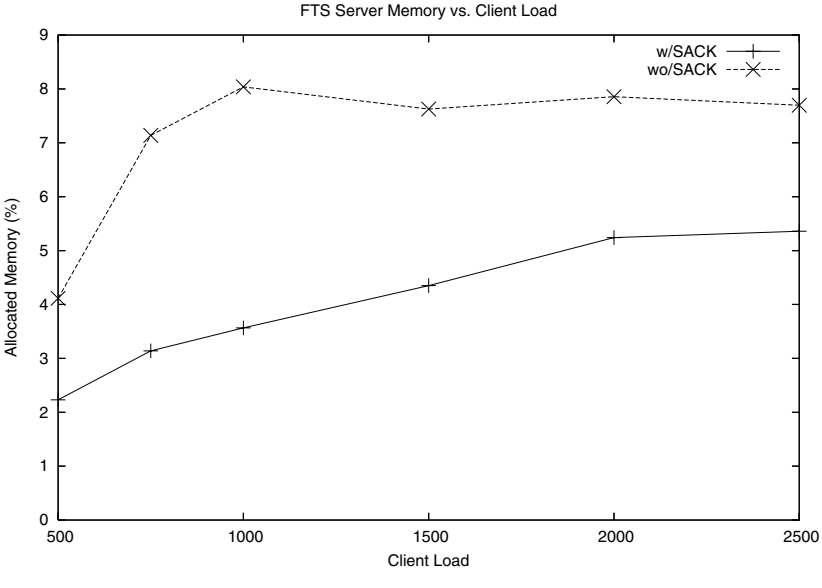


Fig. 3. Performance Evaluation: Server Memory vs. Client Load

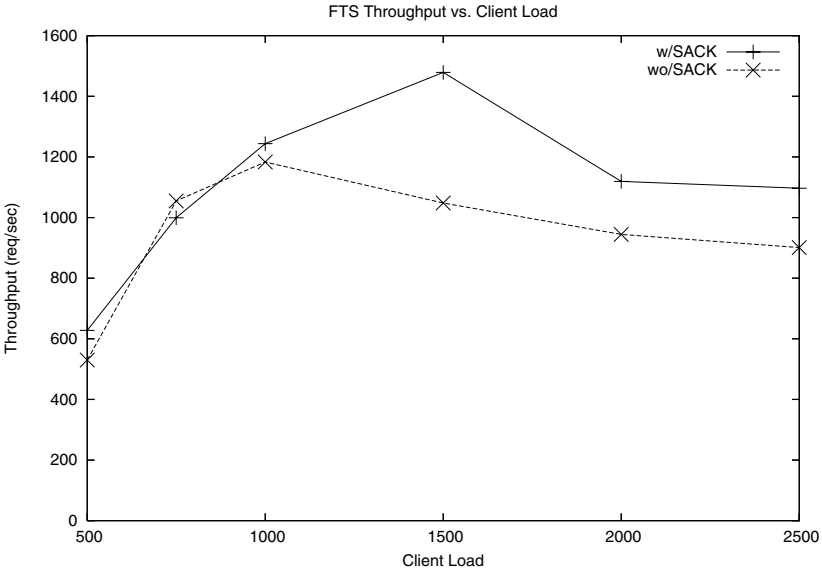


Fig. 4. Performance Evaluation: Cluster Throughput vs. Client Load

Table 1. Performance Evaluation: SACK Overhead vs. Client Load

Client Load	Message Overhead (%)	Byte Overhead (%)
500	1.31	32.9
750	1.23	32.9
1000	1.29	32.9
1500	1.25	32.9
2000	1.11	32.9
2500	0.93	32.9

performance. The throughput improvement can be mostly attributed to two factors. First, SACKs are handled in FTS by a dedicated thread that does not directly serve client requests. This may actually increase processor utilization by running in the idle periods in which all other threads are waiting for I/O operations. Second, the accelerated out-of-order removal of requests from the cache decreases the work of the cache purging thread, thus decreasing its contention with client threads that results from cache access synchronization and CPU utilization.

In the more general case (i.e., not necessarily FTS), any change to throughput following SACK usage may depend on the cache and SACK implementation. However, it seems reasonable to expect that applying SACKs would cause at worst a small decrease of throughput (due to the additional SACK processing), with each stored request being accessed and removed only once, either because of a SACK or because of a time-out. On the other hand, SACKs may also cause a positive throughput improvement through reduced cache contention and reduced memory swapping.

When comparing Figures 4 and 2, there are a couple of observations to be made. First, one can easily verify that the size of the cache when not using SACKs indeed matches the product of the cluster throughput and request expiration timeout (10 seconds). Second, the largest reductions of cache size occur at loads in which the cluster throughput is higher with and without SACKs compared to other loads. This can be explained as following. Each request occupies cache space as soon as it arrives at the server from the client/ABCAST transport, and stays there until being removed. Later, a SACK that matches the request arrives from the same client as that of the request. The higher the throughput, the lower the latency until that client sends the SACK attached to the next request. Thus, the time each request gets to stay in the cache is reduced with throughput increase.

Figure 3 clearly shows how applying SACKs reduces the overall memory consumption of the server by at least 50% for the client loads below 1000 and at all loads in no less than 40%. Also, following the previous observations, note that the larger memory reductions occur in the same client loads that exhibit higher throughputs with and without SACKs. This can be explained as following. The `SackHandler` causes removal of requests at the same rate as it accepts and processes SACKs, which matches the throughput. Therefore, the `SackHandler` on average contains as many SACKs as there are requests in the cache. Since each SACK consumes much less memory than a request (the size of only the request ID vs. the size of the entire request object), and the cache shrinks at least by a half (Figure 2), the overall effect is substantial memory reduction. Another observation arising from Figure 3 is the considerable weight of the request/reply cache in the total server memory consumption, which is due to our test objects consuming little memory.

Last, we consider the message and bandwidth (byte) overhead as presented in Table 1. As this Table shows, the bandwidth overhead is fixed at approximately 33%. This is actually expected, since the number of SACKs received by each server is roughly the same as the number of requests. Thus, the ABCAST byte overhead roughly matches the ratio between the average size of a serialized SACK (approx. 90 bytes) and the average size of a serialized request (about 280 bytes). Note that since we explicitly used short requests in the test, the measured byte overhead is close to the maximum and will become much smaller when using typical requests with many more parameters and complex data types. The message overhead, on the other hand, grows smaller at higher loads. This can be understood when considering that SACKs are batched, so at higher client loads more SACKs get packed in fewer messages, yielding lower SACK message overhead. This also reduces the total processing overhead involved in SACKs, which can explain why the throughput improvement grows with client load as shown by Figure 4.

4 Related Work

Caches are employed in numerous types of applications. Typical examples can be found in areas of memory caching [29], web caching [30], file-system caching [24], database caching [26] and middleware object caching [6]. All of these cache implementations share a common quality of an *information proxy*. That is, the cache is used to store a

copy of remote information (e.g., data, instructions, objects) in a location that is closer to the component that uses it than the source of the information, in order to improve access latency and/or throughput. In that sense, a cache may be considered an optimization that replaces direct access to the remote data, since it still maintains similar guarantees. Continued access to information (liveness) is ensured at the cache by techniques for retrieving remote data that are missing from the cache, e.g., cache miss and pre-fetching. Additionally, cache contents are kept correct w.r.t. the origin (safety) by having the cache guarantee a certain consistency condition, such as linearizability or sequential consistency [6].

In contrast, the request cache in fault-tolerant services is not an optimization at all. Rather, it is used to guarantee a safety condition of *at-most-once* semantics of request execution at the server. In other words, the cache ensures that the same request is not executed more than once, even if the client that sent it originally re-invokes the request due to not receiving the reply in the first time, e.g., following a temporal communication disconnection. In such cases, at-most-once semantics are maintained by having the client receive its reply from the cache instead of by re-executing the request, once the request is done executing for the first time.

A subsequent second difference between the request cache and typical caches is how the cache size can be limited. In typical caches the cache size can be arbitrarily limited by using various policies, e.g., LRU or LFU to enforce removing items when the cache becomes over-full. These policies do not harm the availability of the information since removed items can be retrieved back to the cache from the origin, e.g., when needed. In fault-tolerant services, however, an update request that is removed from the cache without first ensuring that the client will not re-invoke it, may result in the request being re-executed, thereby violating the at-most-once safety condition. Thus, all update requests must be placed in the cache. A completed request can be removed only after ensuring that the client will not require the reply again. This may cause the cache to grow indefinitely depending on the cluster throughput and client load, instead of up to a fixed size.

As one can see from the text above, much more resemblance can be found between the request cache and the send-buffer management algorithm of reliable transport protocols, such as the sliding window of TCP [27]. Both the request cache and the send-buffer basically guarantee that the respective other side, namely the client in replicated services and the other peer of the TCP connection, eventually either receives the stored data or fails/disconnects. However, the request cache guarantees this property at a higher layer than transport, where a client may re-open a connection more than once in order to receive a reply to the same request. Thus, despite the typical transport layer of many middlewares being also TCP, the cache cannot rely on the guarantees of a single connection to deliver the reply. Furthermore, typically when working above TCP, there is no indication of successful delivery of the reply data to the client, and most middleware standards do not implement such a mechanism themselves. Consequently, one has to rely either on timing assumptions or on application-level acknowledgments in order to be able to safely discard the reply. TCP, on the other hand, uses timing assumptions only to detect whether the opposite peer has failed/disconnected and all of its packets

are removed from the network. In contrast, TCP discards data from the send-buffer only upon receiving acknowledgments of successful delivery.

There are several other well-known distributed services that employ a server-side time-based request cache for the purpose of ensuring at-most-once request execution semantics, most notably the *Remote Procedure Call* (RPC) [28] and the RPC-based *Network File Service* NFS [24] services. However, to the best of our knowledge, no published works consider the scalability problem that could be imposed by such a cache and how to reduce such a problem using additional client acknowledgments.

The logging mechanism of FT-CORBA [22] operates quite similarly to a request cache. It stores completed requests and uses a time-out to discard them. Thus, applying our suggested improvement of adding SACK can contribute to reduce the log size of compatible implementations. Our specific implementation of SACKs in FTS where the receiving server propagates received SACKs to its peers using ABCAST to ensure causality (SACKs after matching requests) and to make SACKs effective at all the cache replicas, would also suit the warm-passive replication mode of FT-CORBA where a single primary updates the state of its backups. Active replication in FT-CORBA is less relevant since it assumes that a client receives its replies from all the servers using an ABCAST transport, in which case the log may not be needed for caching replies. One specific variant allows the client to communicate with the servers via a gateway, in which case the gateway should cache replies, and then SACK support can be applied to gateway replication.

However, not all FT-CORBA-compliant implementations use time-based logging. In Eternal [19,21,20], for example, a request/reply pair is removed from the log only when the next request from the same client arrives. This is because Eternal assumes a certain behavior model for clients in which a client may issue at most one request at a time, sending another request only after receiving a reply to the first one. Thus, the arrival of a new request is an implicit acknowledgment of receiving reply for the previous request, which can now be safely discarded. This results in a cache that is only as large as the number of clients, but severely limits client behavior[4]. Contrary to this, by using SACK, there is no need to limit the client's behavior. In particular, an FTS client may invoke requests asynchronously or be multi-threaded. Consequently, requests can only be discarded based on explicit acknowledgments or time-outs. On the other hand, using time-outs impose stronger synchronization requirements in FTS (and in FT-CORBA) compared to Eternal, in the sense that there need to be known upper bounds on the times of request execution, client retry and client-server communication.

5 Conclusions

In this work we have shown the applicability of SACKs as an effective means to reduce the memory consumption of the request cache in replicated services, and, consequently, that of the servers running them. This work is based on an underlying observation that the request cache essentially resembles a sending window in sliding window protocols

[4] In fact, without a backup evacuation timeout, the Eternal cache can overflow when many clients interact for short periods with the same service and then cease to. However, we found no documented evidence of such a timeout.

such as TCP, instead of a regular cache. Thus, it can use similar means, such as SACKs, for evacuating stored data.

After designing and testing the SACK implementation in FTS, we realize that SACKs are indeed a worthwhile modification. In addition to being a safe optimization, SACKs' incurred overhead is outweighed by their contribution to both server scalability and performance. Furthermore, the message and bandwidth overheads associated with SACKs are expected to grow smaller at higher client loads and with more typical usage that results in larger requests.

The proposed method of propagating SACKs to different cache replicas is a derivative of FTS architecture, where each server is responsible for relaying information received from its clients to other servers. Thus, it cannot be applied as-is to, e.g., FT-CORBA-compliant active replication, where a client communicates with all the servers through a proprietary protocol that guarantees request atomicity. Still, it is applicable to the warm replication style, where each primary server propagates client data/state updates to the backup servers. As a general concept, SACKs should be attempted as a means to improve scalability in various middleware implementations such as RPC [28] implementations that maintain at-most-once semantics by caching requests and using time-outs to purge the cache.

Last, SACKs may be especially applicable to 3-tier applications. In these applications, most of the state is stored in the 3rd layer (the database), making the reply cache of the 2nd layer a dominant memory consumer.

References

1. Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.
2. T. Bennani, L. Blain, L. Courtes, J.-C. Fabre, M.-O. Killijian, E. Marsden, and F. Taïani. Implementing simple replication protocols using corba portable interceptors and java serialization. In *2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE, 2004.
3. A. Bhide, E. Elnozahy, and S. Morgan. A Highly Available Network File Server. In *Proc. of the USENIX Conference*, pages 199–205, 1991.
4. K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.
5. K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.
6. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. CASCADE: CAching Service for CorbA Distributed objEcts. In *Proc. Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms*, pages 1–23, April 2000. Best Paper Award.
7. V. Drabkin, R. Friedman, and A. Kama. Practical byzantine group communication. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS)*, Lisbon, Portugal, July 2006. to appear.
8. R. Friedman and E. Hadad. Client-side Enhancements using Portable Interceptors. *Computer System Science and Engineering*, 17(2):3–9, 2002.
9. R. Friedman and A. Vaysburd. Fast Replicated State Machines Over Partitionable Networks. In *Proc. of the 16th Symposium on Reliable Distributed Systems*, October 1997.

10. F.G.P Greve and J.-P. Le Narzul. Implementing ft-corba with portable interceptors: Lessons learned. In *Workshop on Fault-Tolerant Computing, in conjunction with SBRC 2002: Brazilian Symposium on Computer Networks*, May 2002.
11. E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical Report CS-2004-03, Technion, Israel Institute of Technology, 2004.
12. E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical report, Technion, Israel Institute of Technology, 2006.
13. M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
14. Isis Distributed Systems Inc. and IONA Technologies Limited. Orbix+ISIS Programmer's Guide.
15. IONA. IONA Technologies. http://www.orbacus.com.
16. L. Lamport. The Part-Time Parliament. *IEEE Transactions on Computer Systems*, 16(2):133–169, May 1998.
17. S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, April 1997.
18. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgement Options*. Network Working Group, April 1996. RFC 2018.
19. P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Dept. of Electrical and Computer Eng., University of California, Santa Barbara, December 1999.
20. P. Narasimhan and L.E. Moser P.M. Mellier-Smith. Strong Replica Consistency for Fault-Tolerant CORBA Applications. In *Sixth IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pages 16–23, January 2001.
21. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In *Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 81–90, June 1997.
22. OMG. Fault Tolerant CORBA specification, v1.0. ptc/00-04-04.
23. OMG. Portable Interceptors. ptc/01-03-04.
24. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.
25. Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.
26. A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5 edition, 2005.
27. W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, October 1995.
28. W.R. Stevens, B. Fenner, and A.M. Rudoff. *Interprocess Communication*, volume 2 of *UNIX Network Programming*. Addison-Wesley Professional, November 2003.
29. A.S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall Inc., 4 edition, 2001.
30. D. Wessels. Squid web proxy cache.

Modularization of Distributed Web Services Using Aspects with Explicit Distribution (AWED)*

Luis Daniel Benavides Navarro[1], Mario Südholt[1],
Wim Vanderperren[2], and Bart Verheecke[2]

[1] OBASCO project; EMN - INRIA, LINA
École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes cedex 3, France
{lbenavid, sudholt}@emn.fr
[2] System and Software Engineering Lab
Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussels, Belgium
{wvdperre, bverheec}@ssel.vub.ac.be

Abstract. With the adoption of Web services technology to realize Service Oriented Architectures, the need arises for more flexible and dynamic technologies for the just-in-time integration and composition of services. As the runtime integration, selection and management of services involves a variety of crosscutting concerns, such as error handling, service monitoring, and QoS enforcements, Aspect Oriented Programming (AOP) is useful to modularize such concerns.

In this paper we investigate aspect-oriented support for crosscutting concerns of distributed management of web service compositions. We propose to use a distributed AOP approach, Aspects with Explicit Distribution (AWED), to modularize such concerns in a distributed variant of the Web Services Management Layer (WSML). Concretely, we present three contributions. First, we present an extension of the WSML to distributed compositions. Second, we present two extensions to AWED's aspect language which are useful for the modularization of crosscutting concerns of web services: support for chains of (a)synchronous remote advice that communicate through futures, and support for different modes of parameter passing between remote pointcuts and advice. Third, we illustrate our approach by investigating error handling in distributed web compositions.

1 Introduction

Because of their platform and hardware independent nature, web services have become popular to realize Service Oriented Architectures (SOA) [31]. It is well known that low-level infrastructure-related code frequently abounds in the implementation of SOAs based on web services [29,10]. For instance, exception handling code has often to be added around service invocations in order to log

* This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (http://www.aosd-europe.net).

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1449–1466, 2006.
© Springer-Verlag Berlin Heidelberg 2006

failures and trigger backup services. The resulting code frequently crosscuts the business code required by the services.

Aspect Oriented Programming (AOP) [3,17] is useful to modularize such crosscutting concerns of web service based applications. Several previous approaches have investigated this claim in the context of web service infrastructures relying on centralized service composition (as is common in BPEL-based approaches). We have, in particular, proposed the Web Services Management Layer (WMSL) [28] that focuses on achieving dynamic and flexible client-side integration and management of services in the clients. Concretely, the WSML deals with redirecting client requests to services while taking into account QoS selection policies, service composition and client-side management concerns such as monitoring, logging, security and caching.

Recently, the first approaches for decentralized management of web service compositions have been put forward, in particular, to avoid performance bottlenecks arising from centralized web service composition (see, *e.g.*, [8]). However, the handling of crosscutting concerns in such decentralized web service management systems is an open issue. In this paper we investigate the modularization of crosscutting concerns of distributed web service management. We propose an infrastructure enabling distribution of web service compositions based on a variant of the WSML where crosscutting concerns are modularized using a recently proposed aspect language for distributed applications, Aspects with Explicit Distribution (AWED). AWED allows distribution-related crosscutting concerns to be modularized in terms of sequences of remote execution events, advice that executes on remote hosts, and aspects providing for a distributed notion of state.

Here, we present three contributions. First, we present a model extending the WSML to support distributed web service compositions. Second, we extend the AWED language by two new features: chains of advice which may integrate synchronously and asynchronously executed remote advice, and a richer model of parameter passing between remote pointcuts and remote advice. Third, we illustrate how crosscutting concerns of distributed web service compositions may be better modularized than with previous approaches by implementing a non-trivial distributed error handling strategy using our approach.

The paper is structured as follows. Section 2 provides further motivation of crosscutting concerns in distributed web service compositions. In Sect. 3 we introduce our model of the distributed WSML. The AWED language for aspects with explicit distribution and an overview of the implementation of our system is presented in Sect. 4. Section 5 presents how error handling for distributed web services can be implemented using our approach. Related work is discussed in Sect. 6. Section 7 gives a conclusion and presents future work.

2 Motivating Example: Travel Agent Application

In order to motivate our approach, we first present an example motivating distributed WS infrastructures and why crosscutting is a major issue in such a setting. We then give an overview of the distributed web service platform we base our extension on, the distributed Web Services Management Layer.

As a motivating example for our approach, let us first discuss how a travel agent application can benefit from a distributed web service infrastructure and why crosscutting concerns are a particular hard problem for such an application.

A travel agent application is a typical example of a SOA used by customers to, for instance, book an online holiday. For this purpose, the travel agent needs to communicate with a wide variety of services to obtain hotel and flight information and to arrange bookings. In a centralized fashion, this SOA would implement a business process where for instance, first the flight is booked, and based on these results, a hotel is booked at the destination city for the corresponding period. In the distributed approach, the different parts of the WS composition become decentralized and each node deals with a particular subset of the business process at different locations within the distributed system. The nodes communicate directly with each other to transfer data and control, instead of relying on a central coordinator. A distributed implementation of this setting is obviously attractive because, *e.g.*, it allows concurrent execution of independent parts of the composition. However, distributed WS infrastructures must frequently be able to dynamically accommodate changes to the current WS composition. Let us consider three typical scenarios for travel agent applications:

- *Error handling.* If both the hotel booking and flight booking sub processes are handled concurrently by two WSML instances, and one process results in a failure, the other process can be rolled back, too. More generally, a failure situation may require the termination of some executing parts of the composition and rollbacks at a large number of nodes.
- *Performance optimization.* SOAs are frequently performance-critical. One way to measure performance consists in setting up measurement points as part of the composition for each subprocess, network communication and the involved services. Based on this, bottlenecks can be analyzed and actions undertaken. For instance, calls to slow services can be distributed over multiple semantically equivalent services, network traffic on congested networks can be optimized by installing caching functionality and service invocations that take a long time can be executed in advance.
- *Evolution of business requirements.* A change to the business requirements, for instance, relaxation of the rules for booking travel tickets (an operator may start offering the possibility of later reservations for premium customers), frequently call for adaptations of web services compositions.

Each of these three require communication between many nodes. Furthermore, most of these are difficult to anticipate because they often depend on the specific composition at hand. In a context where new compositions may be developed, anticipation would be next to impossible. A dynamic distributed WS infrastructure therefore seems most appropriate for them.

Note that all three scenarios include crosscutting behavior. Error handling [18] is a typical crosscutting concern whose implementation requires modifications at a large number of places in the code that partially depend on the node where an error occurs and where the corresponding error handlers are executed. (see [8] for more information on error handling in distributed WS infrastructures and

an approach to deal with part of this problem). Similarly, the monitoring and performance optimization scenario also requires modifications to many code locations on many nodes. Furthermore, optimization actions may vary much from node to node. Finally, note that even business requirements are frequently crosscutting if a legacy code base has to be adapted (see [6] of a detailed discussion of this issue in the context of replicated transactional caches).

This paper presents a distributed WS infrastructure supporting such dynamic adaptations and, in particular, the modularization of such crosscutting concerns.

3 Overview of Distributed WSML

The Web Services Management Layer (WSML) is an architectural framework for the mediation of Web services in client applications. The WSML is placed in between the client and the web services management infrastructure. All service related code is extracted from the clients and placed in the WSML, where each service related concern is enforced transparently for both the client and the services.

Concretely, the WSML is able to redirect client requests to functionally compatible services, while dealing with possible mismatches and incompatibility. If necessary, multiple services can be composed so that they can cooperate to deal with a given request. Additionally, the WSML enforces a more advanced selection mechanism that contemplates service criteria based on non-functional Quality-of-Service properties of services and deals with other client-side management concerns such as monitoring, logging, security and caching.

The original WSML [28] provided a web service composition model based on a centralized coordinator, similar to common WS-BPEL engines. We have

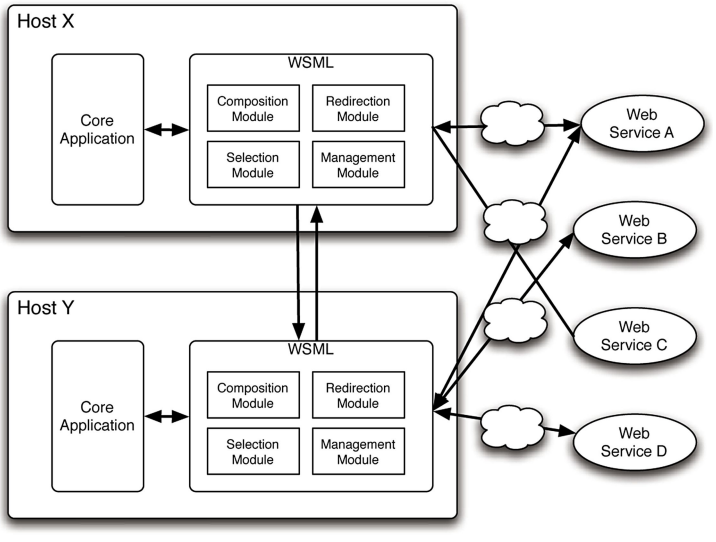


Fig. 1. Overview of distributed Web Services Management Layer

extended this model to a decentralized setup as depicted in Fig. 1. There, two hosts that are part of the composition are equipped with a separate instance of the WSML, which takes care of all service related coordination with third-party services (which are shown in the figure on the right-hand side). The WSML instances communicate with each other to transfer data and control, while not relying on a central coordinator. This approach differs from the one described in [8] where existing composition descriptions are partitioned and statically deployed on multiple WS-BPEL engines. With the WSML, the composition is realized by multiple aspects that are combined together, possibly at runtime. By changing or adding aspects, the composition is dynamically adapted. This runtime flexibility is needed to accommodate the composition of unanticipated changes in the third-party service environment and possible business requirements imposed by the SOA.

4 The AWED Language

Modularization of crosscutting concerns for web services using an aspect language (*i.e.*, in terms of pointcut, advice and aspect abstractions), suggests support for the following issues: (i) a notion of remote pointcuts allowing to capture relationships between execution events occurring on different hosts, (ii) a notion of groups of hosts which can be referred to in pointcuts and manipulated in advice, (iii) execution of advice on different hosts in an asynchronous or synchronous way and (iv) flexible deployment, instantiation, and state sharing models for distributed aspects.

AWED provides such support through three key concepts at the language level. First, *remote pointcuts*, which enable matching of join points on remote hosts and include remote calls and remote cflow constructs (*i.e.*, matching of nested calls over different machines). As an extension of previous approaches AWED supports remote regular sequences which smoothly integrate with JAsCo's stateful aspects [27] but also include features of other recent approaches for (non-distributed) regular sequence pointcuts [13,14,15,4]. Second, support for *distributed advice*: advice can be executed in an asynchronous or synchronous fashion on remote hosts and pointcuts can predicate on where advice is executed. Third, *distributed aspects*, which enable aspects to be configured using different deployment and instantiation options. Furthermore, aspect state can be shared flexibly among aspect instances on the one hand, as well as among sequence instances which are part of an aspect on the other hand.

In this paper, a subset of the AWED language is presented. In particular, we focus on two novel features of AWED with respect to our previous work [5], namely advice chains involving synchronous as well as asynchonous advice with futures, and parameter passing by reference.

AWED's syntax is shown in Fig. 2 using EBNF formalism (*i.e.*, square brackets express optionality; parentheses denote multiple occurrences, possibly none; terminal parentheses are enclosed in apostrophes).

4.1 Pointcuts

Pointcuts (which are generated by the non-terminal Pc) are basically built from **call** constructors (**execution** allows to denote the execution of the method body), field getters and setters, nested calls (**cflow**) and sequences of calls (non-terminal Seq).

AWED employs a model where, upon occurrence of a join point, pointcuts are evaluated on all hosts where the corresponding aspects are deployed. Pointcuts may then contain conditions about (groups of) hosts where join points originate (term **host**(Group)), *i.e.*, where calls or field accesses occur. Furthermore, pointcuts may be defined in terms of where advice is executed (term **on**(Group)). Advice execution predicates may further specify a class implementing a selection strategy (using the term **on**(Group, Select)) which may act as an additional filter or define an order in which the advice is executed on the different hosts. Groups are sets of hosts which may be constructed using the host specifications **localhost**, **jphost** and `adr:port`, which respectively denote the host where a pointcut matches, the host where the corresponding join point occurred and any specific host. Alternatively, groups may be referred to by name. (Named groups are managed dynamically within advice by adding and removing the host which an aspect is located on, see Sec. 4.3 below.)

Finally, pointcut definitions may extract information about the executing object (**target**) and arguments (**args**), and may test for equality of expressions (**eq**), the satisfaction of general conditions (**if**), and whether the pointcut lexically belongs to a given type (**within**). Pointcuts may also be combined using common logical operators.

As a first example, the simple pointcut shown in Fig. 3 could be part of a distributed monitoring aspect. The pointcut matches calls to the bookHotel method that originate from the host that has the specified address. Advice attached to this pointcut will be executed on any host where the aspect is deployed (possibly multiple ones) as there is no restriction on the advice execution host.

In Fig. 4, the pointcut restricts the execution hosts to be different from the host where the joinpoint occurred. Here, the pointcut matches calls to book hotel operations on any type by using wildcards. The advice is executed on hosts other than the joinpoint host and binds the corresponding arguments. Note that in this case the clause !**host**(**localhost**) could replace !**on**(**jphost**) to achieve exactly the same effect of matching non-local joinpoints.

AWED also supports the concept of stateful pointcuts [14] through sequences derived by the non-terminal Seq. A stateful pointcut triggers on a specified sequence of events instead of a single event. For more information about sequences, we refer to [5].

4.2 Parameter Passing

Sharing of remote object information is an inherent need in distributed applications. AWED's new proposal includes a mechanism to manage explicitly how parameters of a given joinpoint are distributed.

```
// Pointcuts

Pc          ::= call(MSig)  | execution(MSig)
              | get(FSig)  | set(FSig)
              | cflow(Pc)  | Seq
              | host(Group)  | on(Group[, Select])
              | target({Type})  | args({Arg})
              | eq(JExp, JExp)  | if(JExp)
              | within(Type)
              | passbyval({Id})
              | Pc || Pc  | Pc && Pc  | !Pc
Seq         ::= [Id:] seq({Step})  | step(Id,Id)
Step        ::= [Id:] Pc [→Target ]
Target      ::= Id  | Id || Target
Group       ::= { Hosts }
Hosts       ::= localhost  | jphost  | "Ip:Port"
              | GroupId
GroupId     ::= String
Select      ::= JClass

// Advice

Ad          ::= [asyncex] Pos({Par}) : PcAppl '{' {Body} '}'
Pos         ::= before  | after  | around
PcAppl      ::= Id({Par})
Body        ::= JStmt  | proceed({Arg})  | localproceed({Arg})
              | addGroup(Group)  | removeGroup(Group)

// Aspects

Asp         ::= [Depl] [Inst] [Shar] aspect Id '{' {Decl} '}'
Depl        ::= single  | all
Inst        ::= perthread  | perobject  | perclass
              | perbinding
Shar        ::= local  | global  | inst  | group(Group)
Decl        ::= [Shar] JVarD  | PcDecl  | Ad
PcDecl      ::= pointcut Id({Par}) : Pc

// Standard rules (intensionally defined)

MSig, FSig  ::= // method, field signatures (AspectJ-style)
Type        ::= // type expressions
Arg,Par     ::= // argument, parameter expressions (AspectJ-style)
Id          ::= // identifier
Ip,Port     ::= // integer expressions
JClass      ::= // Java class name
JExp        ::= // Java expressions
JStmt       ::= // Java statement
JVarD       ::= // Java variable declaration
```

Fig. 2. AWED language

```
1    call(void bookHotel()) && host("134.184.2.3:1432")
```

Fig. 3. Simple pointcut limiting the joinpoint host

```
1  hotelReservation(Hotel hotel, Period period):
2      call(* *.bookHotel(Hotel, Period))
3      && !on(jphost) && args(hotel,period)
```

Fig. 4. Pointcut limiting the execution host

The pointcut language model allows joinpoint information like parameters, caller object and target object to be bound to specific variables in pointcut or advice definitions. The model also allows those variables to be explicitly distributed by value or by reference (the latter is the default behavior). The first option, by value, creates a copy of the object in the hosts where remote advice are executed, binding the new value to the formal parameter used in the advice body. The second option, by reference, creates a remote reference to the original object. Once the remote reference or the copy are bound, they can be treated as local objects without any distinction in the advice body.

Figure 5 shows an example of the usage of the pass by value behavior. The pointcut definition has four variables `i, y, v` and `c`. The parameters of the matched method, `foo`, and the target object of that method are bound to the pointcut's variables. The pointcut **passbyval** is used to annotate the variables `i` and `c` to be passed by value.

```
1  pointcut myDef(Integer i, Object y, Vector v, MyObject c):
2      call(* foo(Integer,Object, Vector, String)) &&
3      args(i, y, v, String) && target(c) &&
4      passbyval(i, c);
```

Fig. 5. Parameter passing example using a pointcut definition

As usual, by value passing has to be used with care, in particular, because it implies a copy of the whole object graph below the object that is passed by copy. AWED allows all objects to be referenced remotely, copied and distributed. Reflective information queried through the `thisJoinPoint` keyword is always passed by reference.

The remote referencing model of AWED is fully transparent with respect to the object model. This means that there is no distinction between the remotely referenced objects and the locally referenced objects. However, it is important to note that the language provides a richer and finer grained model for parameter passing in the pointcut definition language than the one provided in direct

method invocation over objects. When a method is invoked directly in a remote referenced object, the parameters are passed by value as with normal Java method invocations. There is no language support to specify pass by reference behavior (although a work-around using a proxy object is possible). This is motivated by the fact that we aim to stay as close as possible to Java and changing the method invocation syntax and semantics would be a drastic measure.

4.3 Advice

Advice (non-terminal Ad) is mostly defined as in AspectJ: it specifies the position (Pos) where it is applied relative to the matched join point, a pointcut (PcAppl) which triggers the advice, a body (Body) constructed from Java statements, and the special statement **proceed** (which enables the original call to be executed).

In an environment where advice may be executed on other hosts (which is possible in AWED using the **on** pointcut specifier), the question of synchronization of advice execution with respect to the base application and other aspects arises. AWED proposes one unified model for (local and remote) advice execution: all advice (including remote ones) are triggered by one controller. This means that there is only one advice chain per joinpoint. The host where the joinpoint occurs is responsible for managing the advice chain. As such, there is a well-defined precedence as defined by the AspectJ precedence rules (e.g. **declare precedence**), even for advice executed on remote hosts.

Per default, advice executes synchronously to the base application, meaning that the application waits until completion of the advice in order to proceed to the original behavior or to execute the next advice in the chain. Both remote and local advice conform to this general model. The programmer may also choose to execute an advice asynchronously with respect to the application on the joinpoint host by marking the advice with the **asyncex** keyword. This means that the base application proceeds with the original behavior or executes the next advice in the chain while the asynchronous advice is executing. Of course, asynchronous advice are still treated in the same advice chain and thus are only executed when the previous synchronous advice are finished or when previous asynchronous advice are started.

Multiple around advice applying at a joint point is executed as usual in a nested fashion as part of a chain controlled by the invocation of **proceed** (see figure 6). Such advice is expected to return a value that can be processed by the previous advice that invoked proceed or by the original application in case of the first advice. In case of asynchronous advice, this value is possibly not yet computed, so the invocation of such an advice returns a future[3] object [22] that synchronizes with the remote advice in case the object is claimed. The future object is implicit, as such the advice caller can safely treat the return value as a real value. The AWED infrastructure and run-time weaver take care

[3] A future object is an object that represents the result of an asynchronous computation, the actual value of the computation is bound to the object once the concurrent computation is finished.

of generating a transparent future and claiming it whenever its value might be accessed. It is also possible to make the future explicit by casting it to a standard Java Future object[4] when useful. This way, the invoking advice may manage its behavior depending on the availability of the result.

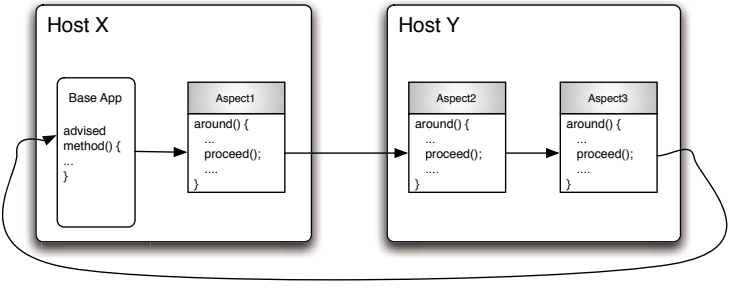


Fig. 6. Around advice chaining in AWED. Advice applicable to the same joinpoint execute in a single advice chain, regardless of execution host.

AWED introduces one general model for (a)synchronous distributed advice. The semantics of AWED remains backward compatible with AspectJ. The semantics of advice is also independent of whether the joinpoint host is different or the same as the execution host. This is in contrast to the previous version of AWED [5], where advice was treated differently because every host had its own advice chain. In that model, advice always executes asynchronously to advice on different hosts, impeding, for instance, remote authentication that blocks all other advice and the original behavior until authentication has been successful. The new general model of AWED is able to support such behavior as illustrated by the code fragment of figure 7. The authentication advice is guaranteed to always execute before other advice that applies to that joinpoint such as, for instance, a billing advice [12]. The billing advice will not be executed when the authentication fails because the authentication advice does not invoke **proceed** in that case.

```
1 around(User user,Object ttarget): toAuthenticate(user,ttarget) {
2     if(userMayAccess(user,ttarget))
3         proceed();
4     else throw new AccessSecurityException(user,ttarget);
5 }
```

Fig. 7. Simple authentication around advice

The advice body has one important additional keyword in comparison to AspectJ: **localproceed**. The invocation of **localproceed** makes sure that the

[4] Java supports futures since version 1.5 through the `java.util.concurrent` API.

original behavior (*i.e.*, the joinpoint) is executed on the host where the advice occurs instead of on the host where the original behavior originated from (*i.e.*, the joinpoint host). As an example, consider the aspect shown in Fig. 8 that implements the distribution concern. It is well-known that distribution can be seen as a crosscutting concern that can be modularized using aspects (see, *e.g.*, [24]). The `distribution` pointcut selects all calls to `Facade` methods on the client and makes sure that the accompanying advice is only executed at the server side. By employing the negation of the `host` designator, calls on the server side will not match the pointcut themselves. The redirection behavior is encapsulated in a synchronous around advice. As the around advice gets executed on the server host, the `getInstance` method of the `Facade` class will retrieve an instance which is local to the server host (this could be generalized in order not to rely on a single object.) The **localproceed** expression makes sure to invoke the original behavior on the server host instead of on the joinpoint host. An interesting variation of the distribution concern would be to mark the advice asynchronous. The result is that the original sequential application is not only distributed but also parallelized. In such a case a future object is immediately returned to the base application and the advice is executed in parallel with the base program. The base program and the advice get synchronized once the actual value of the computation is claimed through the future.

```
1 pointcut distribution(Facade f):
2   target(f) && call(* *(..)) &&
3     && !host("Serveripadr:port") && on("Serveripadr:port");
4
5 syncex Object around(Facade f): distribution(f) {
6   return localproceed(Facade.getInstance()); }
```

Fig. 8. Distribution as an aspect

Finally, note that AWED enables advice to manage named groups of hosts: **addGroup** adds the current host to a given group, **removeGroup** allows to remove the current host from a group.

4.4 Aspects

Aspects (non-terminal Asp) group a set of fields as well as pointcut and advice declarations. Aspects may be dynamically deployed (Depl) on all hosts (term **all**) or only the local one (term **single**).

Furthermore, aspects support four instantiation modes (Inst): similar to several other aspect languages, aspects may be instantiated per thread, per object, or per class.

4.5 Implementation

New Language Features. The AWED language is implemented on top of the dynamic AOP framework JAsCo [25]. JAsCo has two main advantages over

other approaches: its dynamism and run-time performance. JAsCo allows adding and removing aspects while an application is running through its novel run-time weaver. The performance cost of advice execution is minimal and similar to a regular Java method execution. The AWED DJAsCo extension has been made publicly available as a part of the regular JAsCo distribution [16].

We have revised and extended DJAsCo to reflect the improved AWED language semantics as presented in this paper. More concretely, we have implemented support for the new advice chaining strategy including transparant futures. Furthermore, the pass by reference strategy for parameters has been added and installed as default. An interesting fact of the extension is that some parts of the language are themselves implemented by making use of basic AWED aspects. As such, we are able to tackle crosscutting concerns in our own implementation. Due to space constraints, we cannot provide more details about the implementation. For an in-depth discussion of the AWED implementation we refer the interested reader to a companion technical report [6].

Integration with WSML. The Web Services Management Layer can be easily implemented on top of AWED because it offers all the features of the original AOP platform used by WSML (JAsCo). The WSML generates AWED aspects that realize a distributed composition based on higher level templates. By exploiting AWED's distributed capabilities, the WSML is able to solve distribution specific crosscutting concerns. The following section discusses an interesting example of such a concern, namely error handling.

5 Error Handling Revisited

In Sect. 2 we have presented three scenarios motivating the need for aspect-oriented support for distributed WS compositions. In this section we show how to use the AWED language in order to concisely implement such scenarios.

Let us reconsider the problem of error handling in the context of distributed WS compositions. We only consider a model in which errors cause the current composition to be aborted (The same assumption is common in other approaches to distributed WS composition [8]). Error handling requires errors to be identified on all nodes taking part in a WS composition and two corrective actions to be applied when an error occurs: first, active components of the composition have to be terminated; second, effects performed by the composition before the error occurred have to be rolled back. (This obviously requires WS actions to be reversable, a property we assume here.)

Fig. 9 shows how such an error handling strategy for distributed WS compositions can be implemented using AWED. The first two advice (lines 3–11) monitor undo information for two basic web service actions (represented by calls to `store` and `invoke`) to a set of monitors. The monitoring of composition errors is defined in a distributed fashion by means of the pointcut `compositionError` (line 19). This pointcut defines a composition error as a call to `webServiceError` occurring at a host that is part of the host group `CompGroup` and correlated to the

```
1  // ---- 2 example advice for monitoring purposes-----------------
2
3  // monitor database operations within the composition group
4  asyncex after(): call(store(key, val)) && host(CompGroup) {
5    send2Monitors(oldVal(key));
6  }
7
8  // monitor service invocations
9  asyncex after(): call(invoke(target, service)) && host(CompGroup) {
10   sendUndoInfo2Monitors(thisCompositionNode, INVOKE, target, service);
11 }
12
13
14
15 // ---- error handling ------------------------------------------
16
17 // composition errors:
18 // call to webServiceError() after service initialization
19 pointcut compositionError:
20   seq(initCompositionMonitors(CorrelationIdMonitors) && host(Monitor),
21       initCompositionNode(CorrelationIdNode)
22         && host(CompGroup) && eq(CorrelationIdMonitors, CorrelationIdNode),
23       webServiceError() && host(CompGroup)
24      )
25
26 // error: abort local service
27 asyncex after(): compositionError {
28   abortComposition();
29 }
30
31 // service abortion: register abortion with monitor
32 asyncex after(): call(abortComposition()) &&
33 jphost(CompGroup) && on(Monitor) {
34   registerAbortion(joinPointHost());
35 }
36
37 // service abortion registered: terminate blocked services
38 // synchronized to perform undo on terminated services later
39 after: call(aborted(_, _, _)) && on(CompGroup, blockedOnInput) {
40   terminateService();
41 }
42
43 // service abortion registered:
44 // undo previous actions of terminated actions
45 asyncex after: call(aborted(_, undoInfo, compositionHandle)) &&
46                passbyval(undoInfo) &&
47                on(CompGroup, terminated) {
48   undoService();
49 }
```

Fig. 9. Error handling using AWED

relevant set of monitors (using the equality test `eq(CorrelationIdMonitors, CorrelationIdNode)`).

The last four advice define the error handling strategy: First, on occurrence of a composition error the computation at the current node is aborted (line 28). Second, the abortion has to be registered with the monitor (line 34). Third, other still active components (which have to be blocked and waiting for input) have to be terminated (line 40). To this end, the pointcut matches a call to `abort` on the monitor, which provides access to the undo information and a handle to the

current WS composition that is used in case local undos depend on the undos of other activities. Fourth, all terminated activities are undone (line 48).

This example presents the major features of AWED's features for explicit distribution:

- The *pointcuts* are triggered on different hosts and the pointcut `compositionError` line 19) consists of a *sequence* of remote events.
- Steps 2–4 (lines 31–49) use AWED's *remote execution feature*: In step 2, the call `abortComposition` is executed on a host of group `CompGroup`, while the triggering action `registerAbortion` occurs in the monitors.
- Steps 2 and 3 (lines 31–41) illustrates the first new language feature introduced in this work, a *mixed synchronous/asynchronous advice chain*: both advice are triggered on the monitor. In Step 2, termination is triggered synchronously in order to ensure that all activities are terminated before rollbacks are executed. The rollbacks (step 3), however, may be executed concurrently. Note that the body of the advice of steps 2 and 3 are executed on different but overlapping sets of hosts (in fact the set of hosts identified by the predicate `terminated` strictly includes the set identified by `blockedOnInput`).
- Step 4 (lines 43–49) illustrates the use of the new *parameter passing* feature. The (immutable) local undo information is passed by value. Dependencies to other nodes may appear dynamically depending on how their respective undos evolve. A handle to the composition information stored in the monitors is therefore passed by reference, so that those dependencies can be updated dynamically via the monitors.

Finally, note that this application provide clear evidence that we can better handle typical crosscutting issues in distributed WS compositions. We improve, in particular, on the recent work by Chafle et al. [8], who propose a fully centralized monitor and do not accommodate concurrent error handling actions. Our solution improves on these two issues. First, monitoring is done in a distributed fashion in Fig. 9 thus reducing the danger of bottleneck of a single monitor. Second, our implementation directly exploits parallelism in the error handling strategy by the use of asynchronously executed advice that are synchronized only if necessary. Such synchronization occurs in the above example, if the remote reference `compositionHandle` is accessed in step 4 (lines 43–49).

6 Related Work

Since we have proposed a new model for distributed web service composition using AOP, four kinds of related work are presented in the following: approaches using AOP in the context of de/centralized web service composition, distributed web service composition infrastructures, and approaches for AOP in distributed systems.

AOP and Decentralized Web Service Composition. There are only very few approaches applying AO techniques to distributed web service composition. A notable exception is Aspect-Sensitive Services (CASS) [11], which provides a distributed aspect platform that targets the encapsulation of coordination, activity lifecycle and context propagation concerns in service-oriented environments. In contrast to our approach is does not support features for explicit distribution, especially asynchronous advice chaining, as we do.

AOP and Centralized Web Service Composition. Some more recent AOP approaches are explicitly targeted at Web services. With Padus [7] and Ao4BPEL [9], aspects can be (un)plugged into BPEL composition processes. Since BPEL processes consist of a set of activities, joinpoints in Padus and AO4BPEL are well-defined points in the execution of the processes: each BPEL activity is a possible joinpoint. The attributes of a business process or certain activity can be used as predicates to choose relevant joinpoints. BPEL, Padus and AO4BPEL differ from our approach as they realize centralized compositions.

Singh et al. [23] present a software architecture for web services: Aspect-Oriented Web Services (AOWS). It is targeted at describing crosscutting concerns between web services to give more complete description of Web services, supporting richer dynamic discovery and seamless integration. An implementation is made on the .NET platform and all AOWS subsystems and their relationships have been formally modelled. While aiming to achieve similar goals as the WSML, AOWS does not support third-party independent services as services need to be modelled in an AOWSDL language, and registered in a dedicated AOUDDI registry.Multiple services are bundled in a centralized fashion in an AOComposite. The aspectual features of the AOWS framework are used to provide more efficient and effective dynamic description, discovery and integration. Similar as in our approach, service related code is extracted from the client, and the client only needs to communicate with the AOconnectors.

Decentralized Web Service Composition. A few approaches have recently been put forward for decentralized web service composition that do not employ AOP techniques. Most notably, Chafle et al. [8] propose techniques for the partitioning of web service compositions and error handling mechanisms for distributed web service compositions. As presented in Sect. 5 our approach allows the concise definition of more general error handling strategies than they propose. Furthermore, their partitioning technique results in static overall architecture which precludes dynamic changes to the composition.

AOP for Distributed Applications. There are only very few AOP languages and system that support features for explicit distribution (notable exceptions being D [30], JAC [20] and DJCutter [19]). These systems could potentially be used to modularize crosscutting functionalities in web services. However, the distributed features of all of these systems are much more restricted than those of AWED. None of these systems support asynchronous advice execution. Recently, ReflexD [26] has been proposed as a kernel supporting a range of

distributed AO languages. ReflexD provides a meta-object protocol like interface to features for the implementation of distributed pointcuts, advice and bindings. As is, *i.e.*, without language support, ReflexD is not suitable for the concise definition of crosscutting concerns in web services. Furthermore, it is an open question whether the system can be reasonably integrated with existing web service infrastructures, as we show here for AWED and distributed WSML.

Most approaches that can be used currently to apply aspects dynamically to distributed applications (*e.g.*, JBoss AOP [1], Spring AOP [2]) as well as many academic approaches (see DAOP [21] to name just one), essentially allow non-distributed aspects to manipulate applications implemented using an existing framework for distribution, such as industrial component platforms or other middlewares. These approaches do not allow to express web service related crosscutting functionalities concisely because all communication and coordination issues that are handled using AWED by remote advice execution, would have to be implemented manually using the corresponding middleware libraries.

7 Conclusion

In this paper we have investigated the problem of crosscutting concerns of distributed composition of web services. We have presented an approach integrating a new model for such compositions, distributed WSML, with the AWED language for aspect-oriented programming of distributed applications to modularize crosscutting concerns of web services compositions. We have introduced two language extensions for AWED: chains of mixed a/synchronous executed remote advices and an extended model of parameter passing between remote pointcuts and advice. Finally, we have given evidence that these extensions allow the concise modularization of crosscutting concerns for the case of an error handling concern.

This work paves the way for different leads of future work. Most importantly, previous work on AOP has shown that domain-specific abstractions are often useful. In the context of web services a predefined set of composition operators should be useful as well as for the more declarative definition of web service compositions as for reasoning about the correctness of compositions. Second, in this paper we mainly applied AWED's features for remote pointcuts and advice; the application to web services of some other features, especially for distributed state, remains to be done.

References

1. *JBoss AOP*. http://labs.jboss.com/portal/jbossaop.
2. *Spring AOP*. http://www.springframework.org/.
3. M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.

4. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.
5. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM Press.
6. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, and B. Verheecke. Modularization of distributed web services using AWED. Research report, INRIA, June 2006.
7. M. Braem, K. Verlaenen, N. Joncheere, W. Vanderperren, R. Van Der Straeten, E. Truyen, W. Joosen, and V. Jonckers. Isolating process-level concerns using Padus. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, Vienna, Austria, Sept. 2006. Springer-Verlag. (to appear).
8. G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.
9. A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. In L.-J. Zhang, editor, *ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2004.
10. A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.
11. T. Cottenier and T. Elrad. Dynamic and decentralized service composition: With contextual aspect-sensitive services. In J. Cordeiro, V. Pedrosa, B. Encarnação, and J. Filipe, editors, *WEBIST 2005, Proceedings of the First International Conference on Web Information Systems and Technologies, Miami, USA, May 26-28, 2005*, pages 56–63. INSTICC Press, 2005.
12. M. D'Hondt and V. Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 132–140. ACM Press, Mar. 2004.
13. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE'02*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, Oct. 2002.
14. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of AOSD'04*, pages 141–150. ACM Press, Mar. 2004.
15. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.
16. JAsCo. *JAsCo website*. http://ssel.vub.ac.be/jasco/.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

18. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of the 22nd Int. Conf. on Software Engineering (ICSE'00)*, pages 418–427, New York, NY, USA, 2000. ACM Press.
19. M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed aop. In *Proc. of the 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 7–15, New York, NY, USA, 2004. ACM Press.
20. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, LNCS 2192, pages 1–24. Springer, 2001.
21. M. Pinto, L. Fuentes, and J. M. Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
22. J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
23. S. Singh, J. Grundy, J. Hosking, and J. Sun. An architecture for developing aspect-oriented web services. In *Proceedings of the third European Conference on Web Services (ECOWS)*, pages 72–82, Vxjo, Sweden, 2005. IEEE Computer Society.
24. S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of OOPSLA'02*, pages 174–190. ACM Press, 2002.
25. D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press, Mar. 2003.
26. É. Tanter and R. Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, 2006. Springer-Verlag.
27. W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful aspects in jasco. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, LNCS 3628, pages 167–181. Springer, 2005.
28. B. Verheecke, M. A. Cibran, W. Vanderperren, D. Suvee, and V. Jonckers. AOP for dynamic configuration and management of web services. *International Journal of Web Services Research (JWSR)*, 3(1):25–41, July 2004.
29. B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, Jan. 2006.
30. C. Videira Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, 1997.
31. O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 301–312, New York, NY, USA, 2005. ACM Press.

ANIS: A Negotiated Integration of Services in Distributed Environments

Noha Ibrahim and Frédéric Le Mouël

INRIA ARES, CITI Lab., INSA Lyon
21 Avenue Jean Capelle
F-69621 Villeurbanne Cedex, France
{noha.ibrahim, frederic.le-mouel}@insa-lyon.fr

Abstract. The development of highly dynamic distributed environments modifies the runtime behavior of applications. Applications tend to use services available everywhere in the environment and would like to, whenever it is possible and/or needed, integrate services offered by the local environment. In particular, if no single service can satisfy the functionality required by the application, combining existing services together should be a possibility in order to fulfill the request.

In this article, we propose *ANIS*: A Negotiated Integration System. Our system provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `IntegrationLifeCycle` - and the tools implementing these interfaces. These tools offer different techniques of integration (local/remote composition, local/remote weaving, deployment by downloading/uploading), negotiation by contracts and the capability to manage the life cycle of the integration.

A prototype based on Java platform and OSGi technology is implemented as a proof-of-concept to demonstrate the potential of ANIS[1].

Keywords: services, integration, negotiation, life cycle, OSGi, distributed systems.

1 Introduction

For more than one decade, integration of services: the problem of combining different separate services into a unified one to achieve new functionality [1], is generating considerable interest in several computer science communities. The middleware is a particularly interesting domain for service integration for several reasons. Firstly, increasing numbers of service providers are moving into the middleware, providing a collection of useful services for applications [2]. Another reason is the development of highly dynamic distributed environments that modifies the execution of applications at runtime. Applications tend to use services available everywhere in the environment and would like to, whenever it is possible and/or needed, integrate services offered by the local environment.

[1] This work is part of the ongoing European project: IST Amigo-Ambient Intelligence for the Networked Home Environment.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1467–1484, 2006.
© Springer-Verlag Berlin Heidelberg 2006

In particular, if no single service can satisfy the functionality required by the application, combining existing services together should be a possibility in order to fulfill the request. Third, services integration has the potential to reduce the effort of applications that continuously look for services, by combining complementary middleware services provided by independent providers to achieve the end-application's needs [1].

The service integration is particularly challenging because of the diversity of the services available in the middleware. Integrating these services may require different technologies or techniques. Our objective is to propose a system with A customizable Negotiated Integration of Services, *ANIS*. *ANIS* provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `IntegrationLifeCycle` - and the tools implementing these interfaces. These tools offer different techniques of integration (local/remote composition, local/remote weaving, deployment by downloading/uploading, etc..), negotiation by contracts and the capability to manage the life cycle of the integration.

The idea behind *ANIS* is:

- To propose a generic model of service that is independent of any implementation and technology. Our system can be applied to all types of services, from Entreprise Java Beans [3] to Web Services [4].
- To propose a developing framework including APIs for managing the integration of services. The framework includes three essential interfaces: `Integrable` for integrating services, optional `Negotiable` for negotiating the integration and `IntegrationLifeCycle` for managing the life cycle of the resulting integration.
- To propose tools that implement the defined APIs. These tools offer different techniques of integration (deployment of services by downloading a service from a platform or uploading a service into an environment, composition of services and weaving of services), a negotiation protocol based on contracts between the services willing to integrate and a manager of the life cycle of the integration.

Nowadays, number of existing framework do the integration of services but impose, very often, a precise toolkit to do so. In *ANIS*, the framework is independent of the tools implementing it and a service can use our framework with his own toolkit implementation. The framework supplies the general road map for an efficient integration. Applications, depending on their needs, can use the provided toolkit or another one.

We demonstrated the feasibility of *ANIS* by implementing a prototype. We choose to implement it using Java platform and OSGi technology. The tools we propose are specific to the choices we made. The choice of Java was motivated by its portability and its capability to provide a strong separation between the APIs and their implementations. This separation reflects the one we made in *ANIS* between the framework and the toolkit. OSGi [5] was chosen for its facility to provide the Inversion Of Control (IOC) [6]. OSGi simplifies the development,

deployment and management of services by decoupling the service's specification from its implementation.

The strength of *ANIS* is in its framework that can be adapted to all different existing platform (.Net, OSGi, Fractal, J2EE, etc.). The limit of the current version of *ANIS* is in its toolkit; the applications that need to use it has to support the Java Runtime Environment and the OSGi platform.

The paper is organized as follows: section 2 explains the requirements we impose to *ANIS*. Section 3 details the architecture of the *ANIS* system. Section 4 shows implementation of our framework prototype. Section 5 compares our work to existing related integration works. Finally, section 6 concludes and gives future research directions.

2 Requirements for the *ANIS* System

A service integration framework must adhere to certain requirements in order to provide an efficient and viable integration. These requirements depend a lot on the application field we wish to apply *ANIS* to. For the current version of *ANIS*, we choose to limit ourselves to three properties of the integration [7]. We consider these three properties to be essential for integrating services in general distributed environments. Many other properties are interesting to study, such as interoperability or adaptability, but we consider them to be more specific to certain environments such as mobile or pervasive environments. According to future application domain, these properties will be studied later.
The integration has to verify these three core properties:

- Genericity: In distributed environments, all kinds of applications would like to integrate services available around. To make it possible, the framework must be generic. It has to be applied to all types of services, and also to be used by all types of applications.
- Life-time cycle: The integration can be set for a certain time. It can have a life cycle. An integrated service, can therefore last a certain span and disappear once completed its cycle. This property enables applications to integrate services for a purpose, and be sure of the disappearance of the integrated service once the purpose reached.
- Reversibility: As an integrated service has to disappear once its cycle end, integration must be reversible. We mean by reversible the faculty to return to the state previous to the integration process. The integrated service disappears and the services, that were involved in the integration, are still in an operational state.

The integration verifies another property not related to the distributed nature of applications, efficiency. The integration gains in efficiency if previously to integration, services negotiate on terms and conditions of this one. This property is optional and services can integrate without negotiating. Our toolkit propose negotiation by contracts between services but other toolkits can implement the framework without specifying any negotiation protocol.

3 *ANIS* Framework Architecture

This section introduces the architecture of the *ANIS* framework. We consider that all services are gathered in a middleware layer providing a run-time environment. The application layer uses the services of the middleware and the services of the middleware rest on the system layer which provides the management of the hardware and the network. We define a special service *IntegServ*. This service can reside on every host wishing to integrate services. The integration of services, whether it is local or remote, will be managed by an *IntegServ* service. As we will consider integration of services, we will first define our model of service. Then, we detail the *IntegServ* architecture and its sub-parts.

3.1 Service Model

As shown in the figure 1, our service is composed of three parts:

- Interfaces: an interface specifies methods that can be performed on the service. Service's interfaces are public and so published for an external use. A service can hold two kinds of interfaces: functional interfaces defining the functional behavior of the service (e.g. for a video streaming service, a functional interface can allow to specify frame's size, frame's rate, etc.) and integration management interfaces defining the way to manage the integration of this service (e.g. a Integrable interface can allow to specify that the service can be integrated).
- Bindings: a service can provide and/or require functionalities from other services. Bindings express these run-time dependencies (e.g. if a video streaming service requires a QoS communication interface, at one moment, it can bind to a H.323 service, and at some later moment, to a SIP/RTP service).
- Objects: objects realize the functionality expected from the service (e.g. in our video streaming service, objects multiplex/demultiplex, order/reorder video and audio frames, etc.).

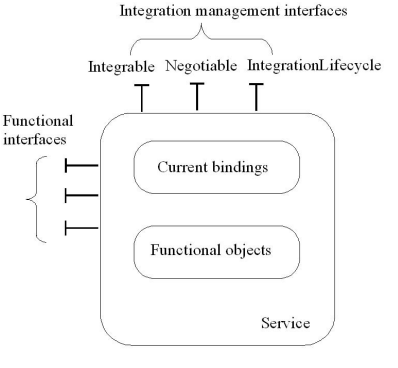


Fig. 1. Service model

Our service model is independent of any implementations and can be applied to EJBs [3], CORBA Components [8], Fractal components [9], OSGi bundles/services [10] or Web Services [4].

Technically, interfaces can be expressed in language-native way, such as Java interfaces or by using an Interface Description Language (IDL), such as in CORBA [8]. Bindings can be expressed by an Architecture Description Language (ADL) [11] . Objects implementations are language-dependent and result from the instantiation of classes.

3.2 *IntegServ* Architecture

Several services run on different hosts in a distributed environment (cf. figure 2). We consider that these services can be final or integrated. A service has to implement the `Integrable` interface in order to be integrated. As specified in the requirements above, it can optionally negotiate before integrating, and can so implement the `Negotiable` interface. After the integration, the integrated service will hold the `IntegrationLifeCycle` interface.

Two middleware services are involved in the integration process. Once an application use the `Integrable` interface of a service the interface can execute an integration defined by the service provider, redirect the call to a local *IntegServ* (step 1 figure 2) or to a remote one if no local one is found. A *Discovery service* takes in charge the task of searching and reporting for the available *IntegServ* (step 2 figure 2). *ANIS* do not provide the *Discovery service* and relies on existing systems such as Jini Discovery Service [12] or UPnP [13].

ANIS applies an integration reflexivity model on its own services. The *IntegServ* service results itself from the integration of several special services: the

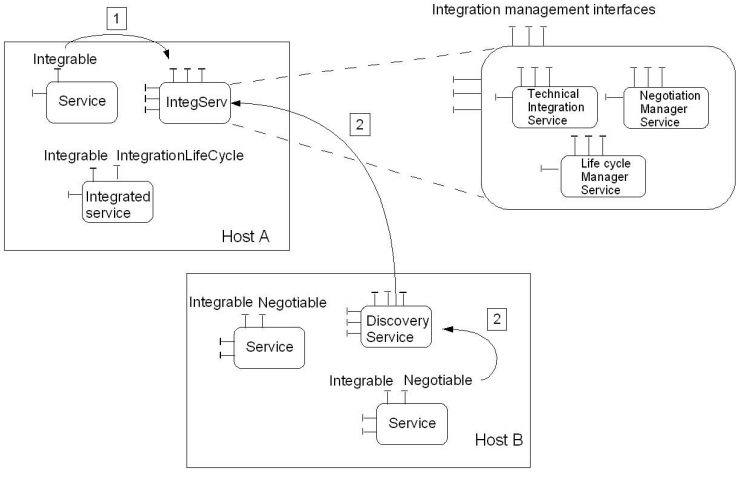


Fig. 2. Architecture of the *IntegServ* service

Technical Integration Service, the *Negotiation Manager Service* and the *Life Cycle Manager*.

3.3 Technical Integration Service

The *Technical Integration Service* applies different techniques of integration involving internal parts of services (i.e. bindings and objects) but also external parts (i.e. interfaces). These different techniques can be applied one before the other and/or combined. They can be applied to a set of services. The *Technical Integration Service* provides three techniques of integration:

- Composition of services is the technique of integrating services by connecting their interfaces. The *Technical Integration Service* defines two methods for choosing how to connect the interfaces. The first one is satisfied by republishing the same interfaces. The composed service provides all the interfaces of the services composing it (cf. figure 3). The second one chooses to provide only one interface which represents all the others (cf. figure 4). This method is used if the output of an interface matches the input of another, making a chain of matching interfaces.
- Weaving of services is the technique of interlacing the object code of a service into object code of another one. Only object parts of services are concerned (cf. figure 5). The weaved service publishes the same functional interface it possessed before the weaving process, but the functionality that the service offers has changed after the weaving.
- Deployment of services is the technique of distributing services in a new environment by downloading or uploading them. Only binding parts of services are concerned. Once a service deployed, new bindings are created and old ones deleted (cf. figure 6).

Nothing prevents from adding new techniques to the *Technical Integration Service*. The only requirement, is that these new techniques have to respect the *ANIS* service model.

3.4 Negotiation Manager Service

Negotiation is the process of getting an agreement between two services about terms and conditions of the integration. This step occurs during the integration process, just before applying the techniques of integration. Three main parts compose the negotiation process: contracts, strategies and negotiation phases.

Contract is conceptually composed of several distinct parts [14]:

- The contract descriptions. Contracts [15] synthesize the terms and conditions of the negotiations in clauses.
- The clauses of the contract. Clauses contain elements that can be measured and quantified, such as QoS or memory resources. We enrich clauses with integration parameter such as duration of the integration, etc.
- The penalties to apply in case clauses are not respected.

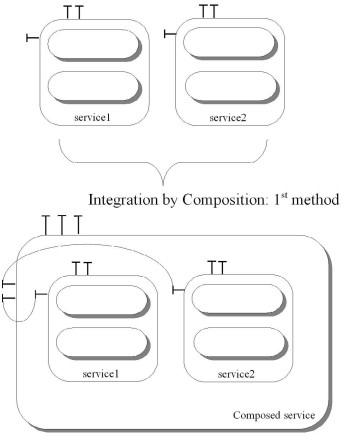


Fig. 3. Composition of services: first method

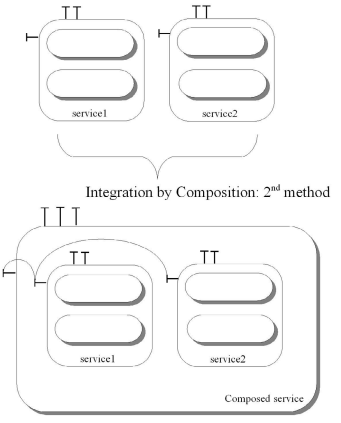


Fig. 4. Composition of services: second method

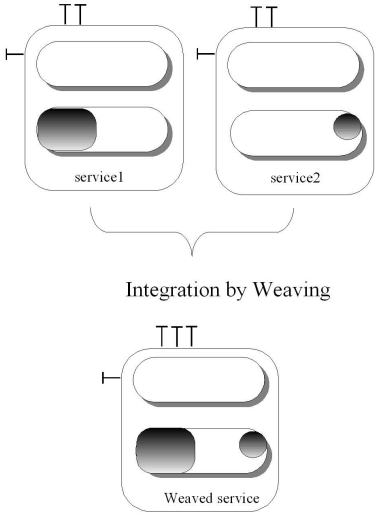


Fig. 5. Weaving of services

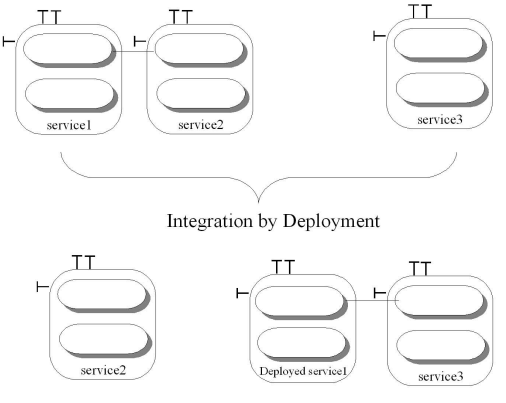


Fig. 6. Deployment of services

Strategies define the methodologies of negotiation. The framework defines two kinds of strategies: service level strategy vs host level strategy. We need these two strategies to make viable integrations on any host. Service level strategy is local to the service. Each service know the integrations it is part of and can establish contracts according to this information. The host level strategy is managed by the *Negotiation Manager Service*. This 'global' information is necessary to prevent services, on the same host, from making contract that can not be honored.

For example, if the negotiation between two services is about memory resources, the service level strategy will take into account only the memory consumed by the service whereas the host level strategy will consider the memory consumed by all services on the host.

The negotiation is split into three phases: Initialization, Active Negotiation and Destruction.

- **Initialization**: This list of stages occurs in this specific order and for a fixed number of time.
  1. propose contract: the service S1 that requires the integration initiates the negotiation with the service S2 by proposing a contract.
  2. analyze contract: S2 analysis the clauses of the contract and verifies if it can satisfy them. Two kinds of analyze may occur, service level analyse and/or host level analyse.
  3. accepte/refuse contract: in case of acceptance, the contract is signed. If not, another proposition can be made (back to step 1).
- **Active Negotiation**: This stage may occur multiple times during the life time of the contract.
  1. isAlive contract: verify the duration of the contract and notify when it ends or when one of the services violate the contract. This verification can be monitored by the services which signed the contract or the negotiation manager service.
  2. re-negotiate contract: services can re-negotiate a contract.
- **Destruction**: This stage occurs only once during the life time of the Contract.
  1. invalidate contract: the contract is invalidated.

3.5 Life Cycle Manager Service

The *Life Cycle Manager Service* is in charge of the integration life time. The integration can have, from its creation, a life time cycle, known and managed by the *Life Cycle Manager Service*. Once this time expires, the integrated service must be disintegrate.

To manage the life cycle of integrated services, each integrated service implements a *IntegrationLifeCycle* interface.

The Life cycle of an integrated service is divided into three parts: Initialization, Active Integrated Service, and Destruction.

- **Initialization**: This list of stages occurs in this specific order, and occurs only once during the life of the integrated service. Beside each stage, we precise the needed interface.
  1. enableLogging: activate and save logs [`IntegrationLifeCycle`].
  2. verify that the service implements the `Negotiable` interface: make sure that the service to integrate accepts negotiation [`Integrable`].
  3. if the service implements `Negotiable` then propose contract: establish a negotiation [`Negotiable`] else proceed to step 4.

4. integrate: integrate services using one of the techniques defined by the *Technical Integration Service* for a specified life-time [`Integrable`].
5. initialize: initialize the integrated service [`IntegrationLifeCycle`].
6. start: start the integrated service [`IntegrationLifeCycle`].

– **Active Integrated Service**: This list of stages occurs in this specific order, but may occur multiple times during the life of the integrated service. Beside each stage, we precise the needed interface.
1. suspend: suspend for a certain time the integrated service [`IntegrationLifeCycle`].
2. isAlive: verify the duration of the life cycle [`IntegrationLifeCycle`].
3. resume: re-start the integrated service [`IntegrationLifeCycle`].

– **Destruction**: This list of stages occurs in the order specified, and occurs only once during the life of the integrated service. Beside each stage, we precise the needed interface.
1. stop: stop the integrated service [`IntegrationLifeCycle`].
2. invalidate contract: to invalidate a contract [`Negotiable`].
3. disintegrate: separate the integrated service into several services [`Integrable`].

The *Life Cycle Manager Service* satisfies the life-time requirement of the *ANIS* system.

4 *ANIS* Prototype

In this section, we present our developing API framework and the toolkit implementation using Java and OSGi technologies. In this article, we detail the three integration techniques proposed by the *Technical Integration Service*. Negotiation and integration life cycle implementations will be the subject of another article.

4.1 ANIS Framework: APIs

`IntegServ` package contains three other packages implementing the integration management interfaces. Package `TechnicalService` (cf. figure 7) is composed of three classes implementing the three integration techniques. Package `NegotiationService` (cf. figure 8) is composed of classes referring to the three main parts of negotiation: Contracts, strategies and negotiation phases and package `LifecycleService` for managing the integrated services. Interface `Integrable` provides several methods allowing to manage the integration of services: method `integrate`, method `disIntegrate`, method `getIntegratedServices` and method `isNegotiable` (to verify if a service accept negotiation). Interface `Negotiable` provides several methods allowing to manage the negotiation between at least two services: method `propose` (to propose a contract), method `accept` (to accept a contract), private method `analyzeContract` (to analyze the contract whether with a service level or host level strategy), method `monitoringContract` (to monitor the contract).

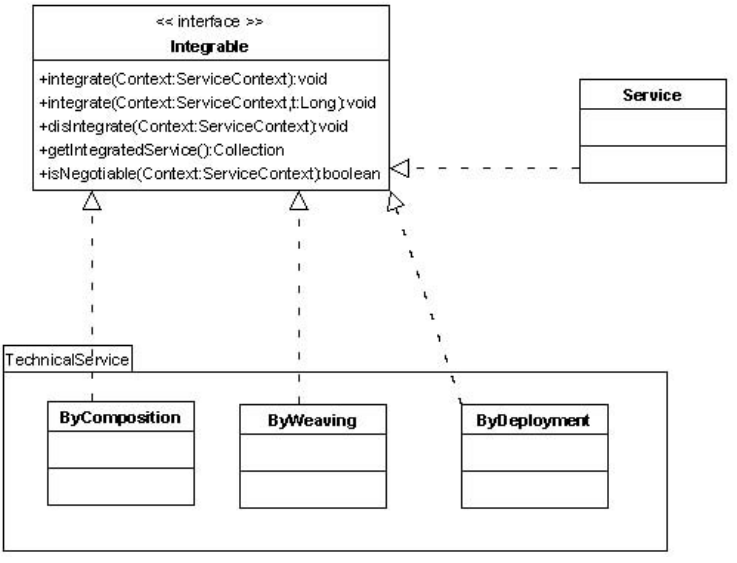


Fig. 7. UML class diagram of *TechnicalService*

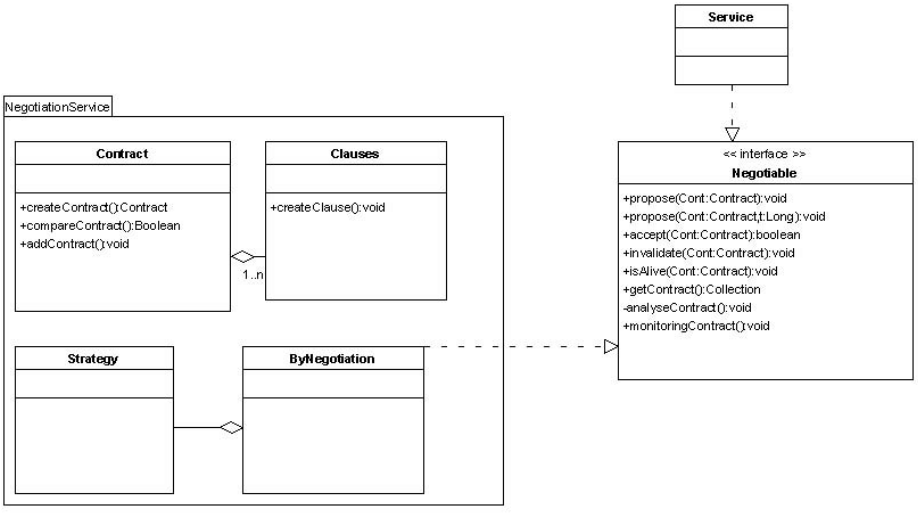


Fig. 8. UML class diagram of *NegotiationService*

4.2 *ANIS* Toolkit: OSGi Implementation

The OSGi specifications define a standardized, component oriented, computing environment for networked services. Adding an OSGi Service Platform to a

networked device (embedded as well as servers), adds the capability to manage the life cycle of the software components in the device from anywhere in the network. We implement our developing framework on Oscar [16] which is open source implementation of OSGi framework specification. We enrich it with an instantiation of our *IntegServ* service.

We apply our service model to OSGi's bundle and service interfaces; objects are Java runtime objects instantiation of classes started by an `Activator`; bindings are modeled by the `manifest.mf` file. A service is provided by a bundle. A bundle can provide diverse services.

Use Case Scenario. In this article we choose to detail the implementation of our deployment (download with RMI), composition by redirection call and weaving techniques. We implemented the following scenario. Max, an architect, wants to accomplish a video of one of his models. Equipped with his PDA, he enters a studio of production of video clip. The studio has a camera and movie maker software. The PDA integrates the software of the video camera as soon as Max enters the studio. Max begins taking the shots he needs of his model. He can command the camera by using a familiar interface installed on his PDA. He doesn't need to know how the camera works! The PDA downloads the driver of the camera (cf. figure 9) and weaves it with the interface software that Max always uses (cf. figure 9). Then, Max uses the movie maker software that the PDA integrated to create his video clip. The PDA composes the movie maker software with the driver of the camera proposing then a new service that enables Max to take shots and create the sequence of the video clip (cf. figure 9). This new service, initially not offered by environment is now available on Max's PDA.

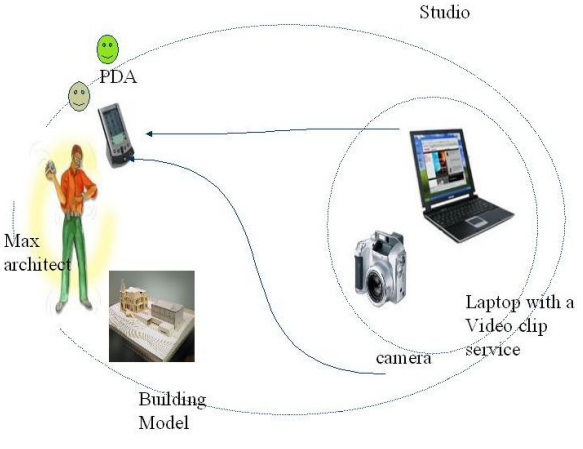


Fig. 9. Use case scenario

Integration Call. The service interface of the PDA calls `integrate` method of `Integrable` interface with the remote context that includes the host of the camera (step 1 figure 10):

```
serviceInterface.integrate(context);
```

Listing 1.1. Service interface integrates context containing service camera

The `context` parameter of `integrate` method is a `ServiceContext` type that inherits the `BundleContext` class of OSGi. This `ServiceContext` contains the list of services available in the context of the service we wish to integrate. It can be `ConcretContext` class when the context is the host of the service. It can also be `ConcretContextSerializable` class when the context is another host. This call to `Integrable` interface will be redirected to the local *IntegServ* on the PDA.

Deployment Technique. Technically, services are provided by bundles. On each host we define a distributed decentralized Oscar Bundle Repository (OBR) where all bundles are stored. The *IntegServ* service running on the PDA analysis the location of the camera service and decides to pull it from its host and run it locally before integrating it with service interface on the PDA. Service camera is stopped and the bundle is serialized and streamed (step 3 figure 10).

The bundle is then downloaded from the stream and installed in the local OBR of the PDA.

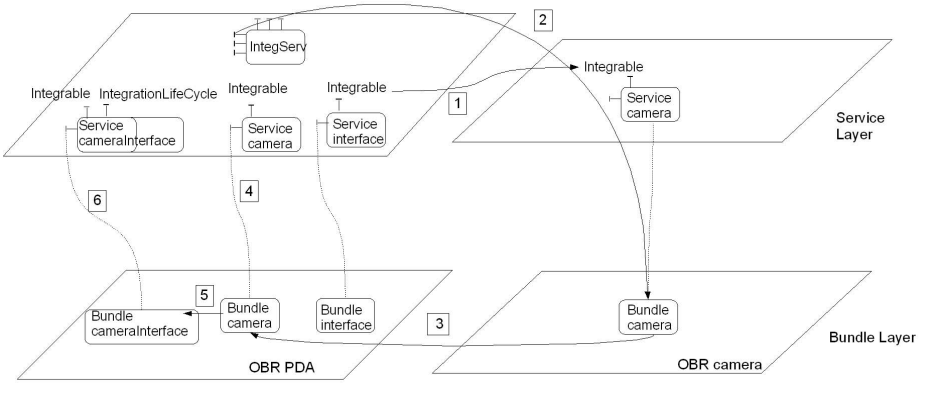


Fig. 10. Integration of the camera service in the PDA

Weaving Technique. Once downloaded, bundle camera is installed and started (step 4 figure 10), service camera is now available on the PDA. *IntegServ* can now apply a local weaving to integrate service interface of the PDA and service camera. The weaving technique weaves into the code of service interface the call for the service camera. As shown in figure 11, this technique adds a `Aspect.class` to the bundle interface and creates a new bundle cameraInterface after weaving this class to `serviceInterface.class`.

The service interface is stopped, classes are extracted from the bundle interface. The AspectJ compiler, given an aspect in source form, produces a binary aspect and runs the weaver. Byte-code weaving takes classes and aspects in `.class` form and weaves them together to produce binary-compatible `.class` files that run in any Java VM and implement the AspectJ semantics. Practically, AspectJ takes `Aspect.class` and weaves it with `serviceInterface.class`. Finally, the whole are jarred into a new bundle cameraInterface offering service cameraInterface (figure 11).

The `Aspect` aspect defines a `callMethodInterface` pointcut that captures calls to all public static methods with names `methodInterface` taking any arguments. Then, the aspect defines one advice after the `callMethodInterface` pointcut : calling `methodCamera` of service camera, `callMethodCamera(context)`. We are testing another way to weave the two services. For now weaving is done using AspectJ. The aspect simply do a redirection call to service camera after the execution of `methodInterface` of service interface. Another way is to write aspects that weave not a call for `methodCamera` in `serviceInterface.class` but the implementation of `methodcamera` as it is in `serviceCamera.class` (cf. figure 11). To do that, service interface and service camera are stopped, classes are extracted from the two bundles interface and camera. Byte-code weaving takes `serviceInterface.class`, `serviceCamera.class` and `Aspect.class` and weaves them together to produce a new `serviceInterface.class` file. Finally, we jar the whole into a new bundle cameraInterface that offers service cameraInterface.

Composition by Redirection Call Technique. A composition by redirection call (cf. figure 12) is then decided by the *IntegServ* to integrate the cameraInterface with the videoMaker. A bundle C is created offering a new service C accessible by the method `methodC()` and redirecting to services cameraInterface and videoMaker (if these two services are available). Method C launches method cameraInterface followed by method videoMaker. The `ByComposition` class figure 7 implements here the second method of composition (figure 4) because the method cameraInterface and videoMaker match.

The *IntegServ* creates the bundle C by writing source file, compiling them and jarring them. The `compile()` method uses the `sun.tools.javac.Main()` of the `com.sun.tools.javac.Main` package and the `jar()` method the `sun.tools.jar.Main()` method.

Nothing prevents from adding a new technique and enrich the OSGi toolkit. The procedure is simple. All we need is to add a new implementation of the method `integrate` in the `TechnicalService` package defined figure 7.

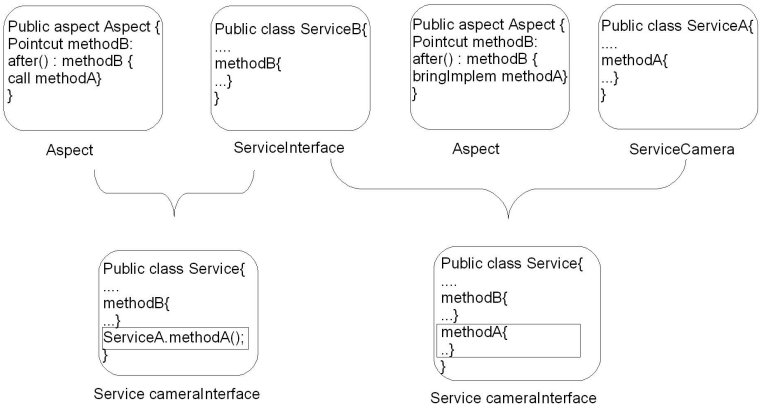


Fig. 11. Two different weaving methods for integrating the camera service (service A) and the interface service (service B)

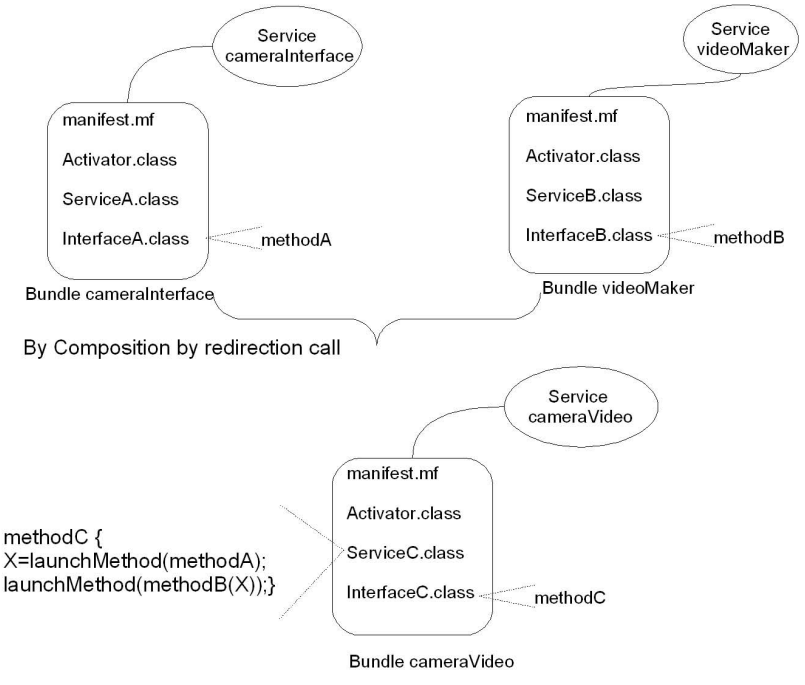


Fig. 12. Composition by redirection call to the cameraInterface service and video maker service

Nothing also prevents from using another Toolkit based on another technology that respects *ANIS* API Framework and *ANIS* generic service model.

5 Related Work

Three major domains of object-oriented programming lean over the concept of integration: the Component-Based Software Engineering (CBSE) [17], the Aspect-Oriented Programming (AOP) [18] and the Service-Oriented Programming (SOP) [19]. Each of these domains has several definitions and techniques of integration according to different existing platforms.

Different types of models based on components as EJBs [3], CORBA Component Model [8] and Fractal [9] allow the interaction between distant components. The integration of components in these different models is often reduced to the deployment and/or the parameterization of these components. In these models, the definition of new components is rather difficult during execution, so the integration of components is often predefined beforehand.

Fractal is a modular and extensible component model. Fractal uses the separation of concerns: separation of interface and implementations, component oriented programming and inversion of control. Fractal vision of integration is to provide composite components defined by their sub-components and the bindings between them.

Recursive and Dynamic Software Composition with Sharing [20] presents a Fractal model which allows sharing of sub-components between components. It defines a structural composition based on containment and binding relationships between components. Two types of bindings are defined, but still these bindings generalize the notion of connectors in architecture definition language. A Fractal framework, which is a projection of the Fractal model in the Java programming language, is presented. This framework offers three different forms of configuration, from static to dynamic configuration. The fact remains that composition is seen as an assembly of components in order to form new, higher-level components. It is reduced to dynamically change bindings to reuse and share components.

Aspect-Oriented programming allows to establish independent transverse concerns (aspects) and to combine them (the weaving) later to produce final application. AspectJ [21], Fac [22] and [23] are models based on aspect, applying the weaving of aspect as method of integration. Fac is an extension of the Fractal component model to support Aspect-Oriented programming. FAC realizes a twofold integration of Component-Based Software Engineering (CBSE) and Aspect-Oriented Software Development (AOSD). Integration is defined as introducing a concern (usually a non functional one such as persistence or security) to a set of components.

Validation of Context-Dependent Aspect-Oriented adaptations to Components [24] aim to take advantage of the expressive power of Aspect-Oriented Programming to modularize concerns about non-functional properties within domain-specific frameworks and components. In this case, both frameworks and components can be refined to satisfy specific requirements at deployment time, composition time as well as at runtime. The key to the integration of Aspect-Oriented Software Development to Component-Based Software Engineering lies in the ability to derive and validate the specification of a component that has

been subject to aspect weaving. By explicitly specifying the desired properties of components and aspects, this article aims to increase the capability to reason about aspect weaving, so that the correctness of a refined component can be verified with respect to a specification. The fact remains that weaving is not done between two component but between a concern and a component.

In the terminology of Service-Oriented Programming, the composition of service very often refers to integration of services. Web services [4] provided a new solution for reusing and assembling web software or components under the distributed service-oriented architecture and across different platform environment with a series of XML-based protocol. Nowadays, researches aim at developing an architecture which allows the composition of service by using a logical reasoning given by the languages of description of service as the DARPA Agent Markup Language (DAML) [25], Universal Description, Discovery and Integration (UDDI) and Web Service Description Language (WSDL) [26]. These languages define standard ways for service discovery, description and invocation (message passing).

SWORD [1] is a developer toolkit for building composite web service. It does not deploy the emerging service description standards such as WSDL and DAML-S, instead, it uses rule-based plan generation, it specifies the web services by using Entity-Relation model.

Architecture-based Web Service Composition Framework and Strategy [27] proposes a kind of relationship-oriented service composition description language, RSCDL, for description the composition process. RSCDL language can guarantee that the complex service can be composed of the reused atomic services by different structural types and can provide a value-added service to different users. Relying on this language description, an architecture-based service composition model was proposed providing a mechanism for dynamic services management and deployment. These works focus more on services matching and searching services for composition than on defining different techniques to compose them.

6 Conclusion and Future Works

In this paper, we have presented *ANIS* A Negotiated Integration System. *ANIS* provides a framework including a set of integration management interfaces - `Integrable`, `Negotiable`, `LifeCycle` - and the tools implementing these interfaces.

The contributions of *ANIS* are three-fold:

- A simple and efficient integration system for a generic model of service.
- An API developing framework for the integration offering different techniques of integration, negotiation by contracts and the capability to manage the life cycle of the integration.
- The implementation of OSGi toolkit that demonstrates the feasibility of *ANIS* and details the integration techniques (deployment, composition by redirection call and weaving).

Our system verifies the defined requirements. The *ANIS* framework is generic. It applies to all types of services, and can be used by all types of applications. The *ANIS* service model is also generic and can be applied to existing implementations. We provide a special API the *IntegrationLifeCycle* to manage the life cycle of the integrated service. We provide within the `Integrable` API a `disintegrate` method that reverse the integration. Our negotiation protocol based on contract permits the negotiation before the integration assuring more efficient integration. Our integration process is simple, services only call `integrate` method of `Integrable` interface without knowing the techniques used behind.

We implement our prototype using Java platform and OSGi technology. We enriched OSGi platform with our *IntegServ* service. In this article, we detail the different techniques of integration: deployment, composition by redirection call and weaving. The limit of the current version of *ANIS* is in its toolkit; the applications that need to use it has to support the Java Runtime Environment and the OSGi platform.

We are improving ANIS by enriching our service model and negotiation process with semantic description. Services but also contracts will be described in OWL-S. For services, this will allow to integrate services having the same semantic description but belonging to different platforms. For contracts, we will define an ontology for the terms of the clauses of the contracts. The strategies of negotiation that are now based on simple matching rules will be enriched using this ontology.

We are also improving our *ANIS* system by adding a new requirement, automation. *ANIS* will become Automatic Negotiated Integration of Services system. This requirement is particularly interesting in pervasive and context-aware environments. We can define context-aware strategies for choosing the services to integrate depending on the execution context. In such environments, we also aim to test the performance of our ANIS prototype for instance, its reactivity to service availability.

References

1. Ponnekanti, S.R., Fox, A.: SWORD: A Developer Toolkit for Web Service Composition. In: 11th World Wide Web Conference. (2002) Honolulu, USA.
2. Kien, T.N., Erradi, A., Maheshwari, P.: WSMB: a middleware for enhanced web services interoperability. In: Interop-ESA'05, First International Conference on Interoperability of Entreprise Software and Applications. (2005) Geneva, Switzerland.
3. Monson-Haefel, R.: Entreprise JavaBeans. O'Reilly & Associates (2000)
4. Iverson, W.: Real Web services. O'Reilly (2004)
5. Alliance, O.: OSGi Service Platform, Core Specification Release 4. Draft (2005)
6. Loritsch, B.: Developing With Apache Avalon. Technical report, Apache Software Foundation (2001)
7. Le Mouël, F., André, F., Segarra, M.T.: AeDEn: An Adaptive Framework for Dynamic Distribution over Mobile Environments. Annales des Télécommunications **57**(11-12) (2002) 1124–1148

8. Zahavi, R.: Entreprise Application Integration with Corba Component and Web-Based solutions. Johne Wiley & sons (1999)
9. Bruneton, E.: Developing with Fractal. The ObjectWeb Consortium, France Telecom (R&D). (2004) version 1.0.3.
10. OSGIalliance: About the OSGI service platform. Technical report, OSGI alliance (2004) revision 3.0.
11. Rennie, M.W., Misic, V.B.: Towards a Service-Based Architecture Description Language. Tr 04/08, University of Manituba (2004)
12. Kumaran, S.I.: JINI Technology An Overview. Prentice Hall PTR (2002)
13. Corporation, M.: Understanding UPnP : A white paper. Technical report, UPnP Forum (2000)
14. Jin, H., Wu, H.: Semantic-enabled Specification for Web Services Agreement. International Journal of Web Services Practices **Vol.1**(No.1-2 pp. 13-20) (2005)
15. Clotet, D.P., Pallotta, V., Rajman, M.: Systematic definition and assent to eContracts for Web Services. In: Workshop on Contract Architectures and Languages (CoALa 2005), in conjunction with the 9th IEEE International Enterprise Computing Conference (EDOC 2005). (2005) Enschede, The Netherlands.
16. Hall., R.S.: Oscar an OSGI framework implementation. Technical report, Objectweb organisation (2005)
17. Heineman, G.T., Councill, W.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley (2001)
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP). (1997) 220–242
19. Singh, M., Huhns, M.N.: Service-Oriented Computing. Wiley (2005)
20. Bruneton, E., Coupaye, T., Stefani, J.: Recursive and Dynamic Software Composition with Sharing. In: Seventh International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002. (2002) Malaga, Spain.
21. Laddad, R.: AspectJ in Action: practical Aspect-Oriented Programing. Manning publications (2003)
22. Pessemier, N., Seinturier, L., Duchien, L.: Components, ADL and AOP: Towards a common approach. In: Workshop ECOOP Reflection, AOP and Meta-Data for software Evolution (RAM-SE04). (2004)
23. Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with Arachne. In: 4th International Conference on Aspect-Oriented Software Development (AOSD'05). (2005)
24. Cottenier, T., Elrad, T.: Validation of Context-Dependent Aspect-Oriented adaptations to Components. In: WCOP, Oslo (2004)
25. Sheshagiri, M., des Jardins, M., Finin, T.: A planner for composing services described in DAML-S. In: International Conference on Automated Planning and Scheduling (ICAPS) 2003 Workshop on planning for web services. (2003)
26. Walsh, A.E.: UDDI, SOAP and WSDL: the Web Services specification Reference book. Pearson Education (2002)
27. Yuan, R., li Zunchao, Boqin, F., Jincang, H.: Architecture-based Web Service Composition Framework and Strategy. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems(ECBS'05). (2005)

Towards a Generic Group Communication Service*

Nuno Carvalho[1], José Pereira[2], and Luís Rodrigues[1]

[1] University of Lisbon
{nunomrc, ler}@di.fc.ul.pt
[2] University of Minho
jop@di.uminho.pt

Abstract. View synchronous group communication is a mature technology that greatly eases the development of reliable distributed applications by enforcing precise message delivery semantics, especially in face of faults. It is therefore found at the core of multiple widely deployed and used middleware products. Although the implementation of a group communication system is a complex task, application developers may benefit from the fact that multiple group communication toolkits are currently available and supported.

Unfortunately, each communication toolkit has a different interface, that differs from every other interface in subtle syntactic and semantic aspects. This hinders the design, implementation and maintenance of applications using group communication and forces developers to commit beforehand to a single toolkit, thus imposing a significant hurdle to portability.

In this paper we propose jGCS, a generic group communication service for Java, that specifies an interface as well as minimum semantics that allow application portability. This interface accommodates existing group communication services, enabling implementation independence. Furthermore, it provides support for the latest state-of-art mechanisms that have been proposed to improve the performance of group-based applications. To support our claims, we present and experimentally evaluate implementations of jGCS for several major group communication systems, namely, Appia, Spread/FlushSpread and JGroups, and describe the port of a large middleware product to jGCS.

1 Introduction

View synchronous group communication is a coordination paradigm that eases the development of multi-participant applications, ranging from replicated servers, cooperative caches, multi-user cooperative application, just to name a few. The set of protocols that implement group communication services are typically bundled in a package called a *group communication toolkit*. After the pioneer work initiated two decades ago with Isis [1], many other toolkits have been developed. Appia [2], Spread [3], and JGroups [4] are, among others, some of the group communication toolkits in use today. Therefore, group communication is, today, a mature technology that, when correctly used, greatly eases the development of reliable distributed applications. At the same time, group communication is still a hot research topic, as performance improvements and wider applicability are sought [5,6,7,8,9].

* This work was partially supported by the IST project GORDA (FP6-IST2-004758).

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1485–1502, 2006.
© Springer-Verlag Berlin Heidelberg 2006

For self-containment, we provide a very brief introduction to group communication. A group communication toolkit integrates two complementary services: membership and multicast communication. Informally, the role of the membership service is to provide, to each participant in a distributed computation, information about who is active (or reachable) and who is failed (or unreachable). Such information is called a *view* of the group of participants. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties. Membership and multicast need to be integrated because reliability guarantees are usually defined in the context of the current group view. For instance, if the membership service indicates that participants A, B and C are active, the reliable multicast service will deliver all multicast messages to these three participants unless one of them fails meanwhile (in which case, a new view is installed).

From the description above it is clear that group communication is much more than just "yet another reliable multicast protocol" given that, the added value, is the precise semantics that are enforced among the communication and membership services, namely in the presence of faults [10]. By ensuring that the same messages are delivered to multiple destinations, ordered among themselves and with group membership change notifications, each message may be handled by the application in a predictable and globally consistent context.

Naturally, by enforcing strong semantics, group communication is more expensive that other weaker forms of multicast, such as best-effort multicast. Therefore, it should not be used when the application has weak consistency requirements. On the other hand, group communication excels when the application is required to maintain global invariants on distributed state. Complex distributed state maintenance problems are then greatly simplified [11]. Group communication is therefore found at the core of multiple widely deployed and used middleware products, namely, the iBus//MessageBus scalable publish/subscribe toolkit, the JBoss and Zope application servers, the Sequoia (formerly Objectweb C-JDBC) database cluster, and the CORBA FT high-availability service. Group communication is also a cornerstone of innovative research prototypes such as Postgres-R [12] and Eternal [13].

Unfortunately, each communication toolkit has a different interface, that differs from every other interface in subtle syntactic and semantic aspects. This hinders the design, implementation and maintenance of applications using group communication and forces developers to commit beforehand to a single toolkit, thus imposing a significant hurdle to portability. Such commitment is undesirable because group communication toolkits are often optimized for specific execution environments. The ability to replace one toolkit by another has the advantage of allowing the application to use the most appropriate toolkit for each deployment scenario. If the application code is tightly coupled with a particular toolkit, changing the implementation of group communication requires a costly refactoring. This also prevents emerging loosely coupled service oriented architectures from taking full benefit of view synchronous group communication.

In this paper we tackle the problem of defining a generic interface that may be used to wrap multiple toolkits. The interface, called *Group Communication Service* for Java, or simply jCGS, has been designed for the Java programming language and leverages

on several design patterns that have recently become common ground of Java-based middleware. The interface specifies not only the API but also the (minimum) semantics that allow application portability. jGCS owns a number of novel features that makes it quite distinct from previous attempts to define standard group communication interfaces, namely:

- jGCS aggregates the service in several complementary interfaces, namely a *configuration interface*, a *message passing interface*, and a set of *membership interfaces*. The configuration interface specifies several opaque configuration objects that encapsulate specifications of message delivery guarantees. These are to be constructed in an implementation dependent manner to match application requirements and then supplied using some dependency injection technique. The message passing interface exposes a straightforward interface to sending and receiving byte sequences, although concerned with high throughput, low latency and sustainable concurrency models in large scale applications. Finally, a set of membership interfaces expose different membership management concepts as different interfaces, that the application might support or need.
- jGCS provides support for recent research results that improve the performance of group communication systems, namely, *semantic annotations* [9,8,7] and *early delivery* [14,15,6,16].
- open source implementations of jGCS for several major group communication systems have been already developed, namely, Appia [2], Spread [3] (including the FlushSpread variant), and JGroups [4]. Besides making jGCS outright useful in practice, these validate that the interface is indeed generic.
- the interface introduces negligible overhead, even when the jGCS is implemented as wrapper layer and is not supported natively by the underlying toolkit.
- the interface has been already integrated in existing middleware products. In particular, we also describe here the port to jGCS of the Sequoia database clustering middleware (formerly Objectweb C-JDBC), as an illustration of the expressiveness and performance of jGCS.

As we have already noted, there were previous attempts to define generic group communication interfaces. As it will become clear after we present our work, those attempts have approached the problem from quite different, and often orthogonal angles. We will postpone comparison with related work until Section 6.

The rest of this paper is structured as follows. In Section 2 we briefly describe the goals that we propose to achieve with the presented service and the pitfalls that need to be avoided in the design of such a service. Section 3 introduces the jGCS interface. Section 4 describes existing jGCS bindings and Section 5 offer a performance evaluation of the jGCS. Section 6 compares jGCS with related work and Section 7 concludes the paper.

2 jGCS Design Goals and Pitfalls

The design of the jGCS is shaped by a number of goals that determine also a number of tradeoffs. In this section we enumerate and describe each of these goals. Furthermore,

the design of the jGCS is also shaped by the need to avoid a number of pitfalls: features that would at first sight seem desirable but which in fact are not and would force unbearable compromises. In this section we discuss not only the goals that we aimed with the design of the jGCS but also the pitfalls we have avoided in the process.

2.1 Goals to Achieve

Goal 1: *No changes to payload required.* The first goal is that when the jGCS is implemented on top of an existing toolkit, no assumptions or changes are made on message payload. This means that implementing jGCS does not require specific data formats, additional message headers or additional messages exchanged. Naturally, toolkits that adopt jGCS as their native interface are free to implement jGCS-specific optimizations. As a result, applications using a specific protocol through jGCS are interoperable with legacy versions using native interfaces. Furthermore, no Java specific constructs or data formats are forced on the application, most notably, no Java serialization is required. This makes it possible to easily translate the proposed API to languages in the same family such as C++ or C#.

Goal 2: *Support service locator and dependency injection patterns.* Complex applications of group communication can make use of multiple groups and even multiple service guarantees selected independently for each message. On the other hand, different sets of guarantees might be available on each implementation of jGCS. It is therefore required that all details regarding protocol configuration and service selection are encapsulated in objects that can be supplied to the application by a third party (i.e. the configurator) using a service locator[1] [17] or the dependency injection patterns[2]. As an example, this allows substitution by a stronger service, when the exact service required by the application is not available in the target environment.

Goal 3: *Support multiple group-based programming paradigms.* jGCS is flexible enough to support different flavors of multicast communication based on process groups. Most notably, jGCS supports both open groups (where any process can send messages to the group) and closed groups (where only group members can send messages to the group). Open groups are useful in client/server applications while closed groups are useful for coordination and synchronization among servers.

Additionally, besides the more common multicast group paradigm, in which messages are targeted to all group members, jGCS supports peer groups, in which messages are target to specific members of the group. As an example, a multicast group is useful for data replication while a peer group is useful in a load balancing application. Note that both flavors require precise knowledge of current membership to function properly.

Goal 4: *Export a flexible subsetable interface.* Supporting existing toolkits requires an extensive interface. This includes facilities for sending and receiving messages accord-

[1] http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html
[2] http://www.martinfowler.com/articles/injection.html

ing to each group communication paradigm, as well as to receive membership notifications with various degrees of detail.

In some scenarios, there may be an overlap among the services provided by jGCS and other services already in-use in the target system, such as best-effort reliable multicast protocols and cluster management infrastructure. Therefore, it may be useful to deploy just parts of the jGCS to avoid redundancy. Due to this reason, jGCS has been designed to be subsetable, in the sense that parts can be independently reused, without carrying along with partially implemented interfaces and runtime exceptions.

As we will see, the subsetable property is also useful to accomodate multiple view synchrony variants (however, see Pitfall 2 below).

Goal 5: *Non-blocking input/output and container-managed concurrency.* The jGCS supports an event-driven interface. The application registers a number of callback listener interfaces to be notified of messages arriving and changes to group composition. The rationale for this is twofold. First, avoiding the requirement to have threads blocked on input/output improves scalability and allows application containers to manage threading under a single integrated policy. Second, ordering guarantees directly translate into synchronization requirements, thus allowing jGCS implementations to cooperate with application containers to optimize the number of concurrent threads.

Goal 6: *Accomodate latest research results.* Finally, it is a goal of the jGCS to allow recent research results, such as support semantic annotations [9,8,7] and early delivery [14,15,6,16], to be easily accomodated. In fact, the goal is to foster programming idioms that naturaly take advantage of such results as they become available. Section 3 will address this topic in detail.

2.2 Pitfalls to Avoid

Pitfall 1: *Specify a common set of service guarantees.* By assuming that protocol configuration and selection of service guarantees are hard-coded within the application, portability to multiple protocol implementations can only be achieved by standardizing on a limited set of guarantees that must exist everywhere. Such agreed set is either very small, and thus of limited use, or large and not portable to multiple implementations. jGCS avoids this pitfall by assuming a configuration step as described in Goal 2, that matches available service guarantees to application requirements.

Pitfall 2: *Exclusively reuse existing standard interfaces.* The semantics of view synchronous group communication are so different from other message passing middleware, such as JMS, that any attempt to map these semantics into other paradigms introduces substantial obstacles to all goals enumerated in the previous section. Furthermore, given the semantic mismatch, is also likely that no significant portability advantages result from the exercise. A better option is to provide a syntatically incompatible interface that embodies similar structure and the same patterns such that programmers can easily make the transition.

Pitfall 3: *Provide interfaces for protocol composition.* A lot of research effort has been invested in composing group communication protocols from fine-grained components by using uniform interfaces [18,19,2] or even standard ad-hoc interfaces [20]. The main problem is that the mapping of an existing implementation to a component interface is not straighforward and thus the approach is not general. Furthermore, interfaces that allow efficient assembly of fine-grained protocol components are likely to impose a specific runtime that is not acceptable as a general purpose application programming interface.

3 A Generic Group Communication Service

This Section describes jGCS, a generic group communication service for Java. We provide a specification of the API and of the minimum semantics that support application portability. The service is organized in four complementary interfaces, namely: the configuration interface, the common interface, the data interface, and the control interface. Each of these interfaces is decribed below.

3.1 Configuration Interface

The configuration interface decouples the application code from specific implementations by requiring that a third party, the configurator, matches available services with application requirements. Besides the obvious portability advantages, this also fulfills Goal 2. This interface shown in Figure 1 is composed by opaque objects as follows:

ProtocolFactory. The protocol factory serves as the interface entry point and triggers the initialization of runtime instances of a protocol implementation. At the semantic level, it encapsulates an implicit service guarantee specification which is enforced for all sessions.

GroupConfiguration. A group configuration encapsulates the address of a group that can be used to open a session that subsequently allows messages to be sent or received, or the membership to be observed. As the ProtocolFactory, at the semantic level it also encapsulates an implicit service guarantee specification which is enforced for all messages exchanged. This object may be used as a key in hashtables.

Service. A service encapsulates a specification of the guarantees to be enforced on a particular message. Upon encountering a service specification that is unknown or incompatible with group or protocol configuration, the implementation must throw an exception. A partial order is defined on guarantees provided by services by extending the Comparable interface (i.e., some services may be stronger than, and subsume, other services). Therefore, the application can use the service interface to discover if a service guarantee is subsumed by some other.

Annotation. An annotation is an optional field that encapsulates semantic knowledge about a message that can be used by the protocol to optimize performance. The contents of the annotation are therefore implementation specific and protocols should silently ignore unknown annotations without erroneous or unpredictable behavior.

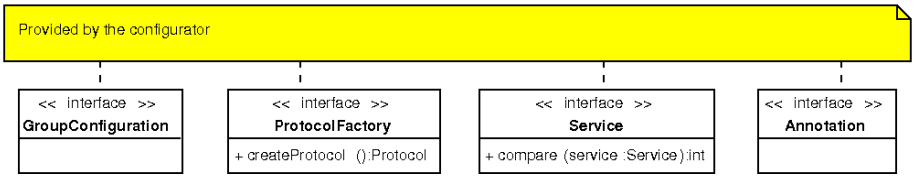


Fig. 1. Configuration interfaces

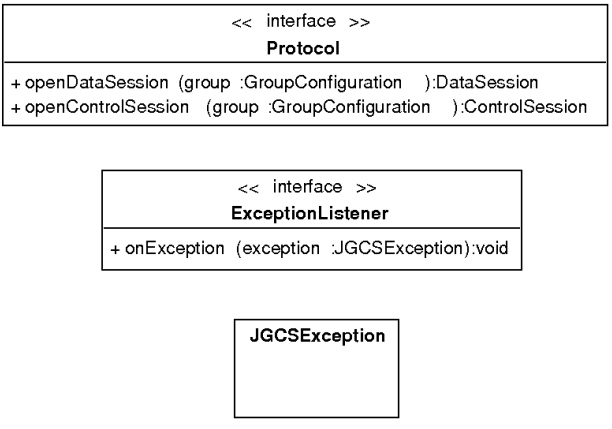


Fig. 2. Common interfaces

Configuration objects should be easily stored and retrived in configuration files and directory services. It is therefore advisable that implementations provide configuration objects with one or more of the following properties: are serializable, can be constructed from properties files, and export parameters according to JavaBean conventions. For the same reason, these objects should not be used to keep session state at runtime.

3.2 Common Interface

A protocol session is represented by a Protocol instance, obtained from the configuration stored in a ProtocolFactory. Using a Protocol instance it is possible to obtain, for a specific GroupConfiguration, a *data* and a *control* session. All further operation are invoked through one of these two interfaces, as depicted in Figure 2.

Both data and control sessions identify group members using java.net.SocketAddress objects. This directly allows a large number of protocols to be supported without any form of address conversion. Protocols that use different address formats, can easily be wrapped. Examples of the use of both native and wrapped member identifiers are described in Section 4.

Finally, exceptions thrown by the jGCS extends the JGCSException class, which itself extends the java.io.IOException interface. Usually, a nested implementation dependent exception can be obtained by using the standard getCause method. Exceptions

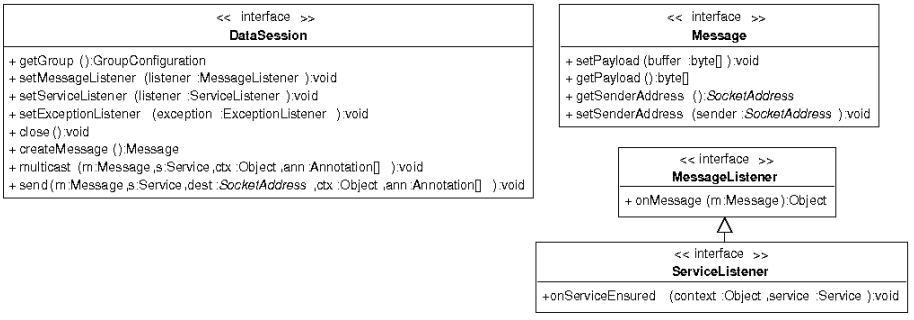


Fig. 3. Message passing interfaces

thrown asynchronously within the protocol implementation are delivered to the application using the ExceptionListener interface. This can be registered using either session object.

3.3 Data Interface

The data interface provides the methods for messages to be sent and received. Whenever the application multicasts a message there is always a specific quality of service, i.e. a specific set of guarantees, associated with the request. The guarantees can be implicitly derived from the group or protocol configuration or explictly set using a Service parameter. The data interface shown in Figure 3 is as follows:

DataSession. The data session provides methods for sending messages in both multicast and peer groups. It also allows registering listeners for the various events.

Message This interface wraps payload and sender address. According to Goal 1, the only payload supported is a byte array. As instances can only be created by the DataSession, it can be implemented as a thin layer on implementation specific objects to avoid having to perform additional buffer copy operations.

MessageListener. Handles delivery of message payload. This is the main entry point for incoming data. When no separate ServiceListener is being used, implicitly does service notification.

ServiceListener. Handles delivery of service notification events. As described below, applications that do not need to be optimized for concurrency can ignore this interface.

The data interface exposes one of the key features of view synchronous group communication: messages are delayed by the protocol implementation to be delivered to the application only after global guarantees have been ensured, namely, regarding order and stability. For instance, when providing uniform agreed (or safe) guarantees [10], the implementation must collect a number of acknowledgements from differente members before issuing the message delivery.

However, recent work on group communication [5,6] has shown that it is useful to deliver the payload to the application as soon as it is received and then later notify

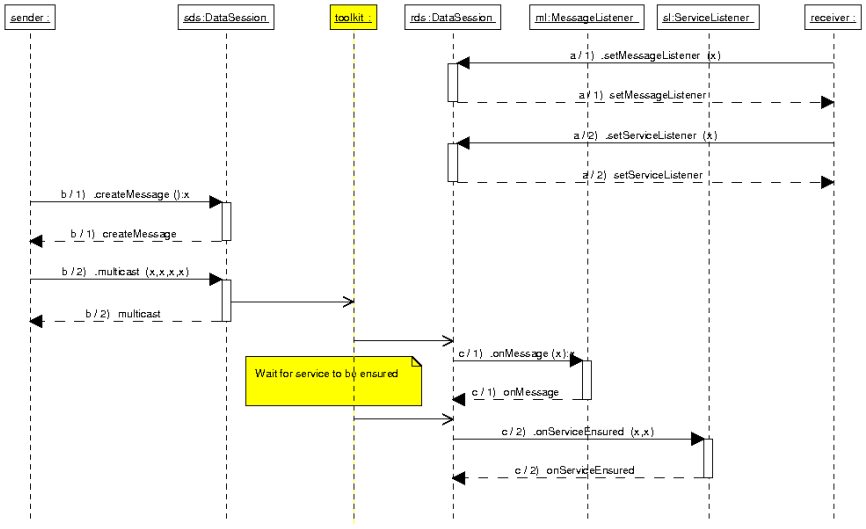


Fig. 4. Data session usage

the application that the requested service has been ensured. This allows increased concurrency and masking of latency, by allowing the application to start processing the message earlier, at least, by deserializing the message in parallel with the execution of the remaining of the protocol. Examples of this strategy can be found in systems based on optimistic atomic multicast protocols [21].

jGCS supports this optimization as described in Figure 4. The application registers a ServiceListener with the DataSession. The protocol is allowed to deliver payload without ensuring services. Upon handling the message, the application chooses how to proceed:

- Returns a context reference (any POJO) which the protocol associates with the message. Typically, the context contains a pre-processed message. When service is ensured, the protocol calls back into the application providing references to both the context object and the service object that has been achieved. The application then resumes processing the message.
- Returns a null reference. This informs the protocol that no further notifications or service guarantees are required for this message and no further callbacks happen.

Notice that even if the protocol implementation does not natively support this interface, the binding can trivially support it by performing both callbacks only after the final delivery. Currently, the only toolkit that natively supports this interface is Appia [22].

On the sender side, the jGCS also provides mechanisms to prevent the aplication from being blocked when invoking the interface. For instance, a specific protocol implementation may not accept requests until some service is ensured. Also, an implementation may perform end-to-end flow control, thus throttling the sender in a similar fashion. The non-blocking interface works as follows. Upon sending a message, an

application might also specify a context. This means that multicast does not block and the application gets notified using the service listener callback.

An additional advantage of jGCS is that it does not impose artificial limits to the application concurrency, namely in the processing of incoming messages. In fact, jGCS allows for concurrent message delivery notifications whenever the requested service does not impose ordering on messages. Therefore, only total ordering constraints prevent concurrent deliveries. Also, this applies both to payload deliveries, when no service listener has been registered, as well as to service callbacks. Notice that in the later, payload deliveries can always be performed concurrently, up to an optimal concurrency degree, that can be coordinated with application containers.

Finally, the jGCS provides support for the use of semantic knowledge to improve system performance. This is achieved by letting application annotate messages with control information that can be used by the group communication toolkit to selectively relax reliability, order and view synchrony guarantees [8,9,7]. For that purpose, the application should obtain one or more annotation objects in an implementation specific fashion. These are then handed to the protocol as parameters in the multicast operation. Although these objects are implementation specific, this interface does not pose a threat to portability as, by definition, a semantic annotation can be safely ignored.

3.4 Control Interface

The control interface provides a flexible subsetable interface for a wide range of membership management protocols. The most simple interface is suitable only for best-effort multicast protocols and is shown on Figure 5:

ControlSession. Provides methods for entering and leaving a group, as well as for registering a listener for control events.

ControlListener. Allows a simple notification of members entering and leaving the group. Precise semantics of these events, namely regarding concurrency with message deliveries, depends on the implementation.

This interface can be used separatly for failure detection or cluster management infrastructure, which are not directly related to group communication. Notice also that implementations can choose to distinguish members that have left the group volutarily and in a controlled fashion from members that have failed and thus been forcibly excluded. The former allows recovery from a known state and thus is more efficient.

Support for view synchronous group communication differs wether the underlying implementation provides sending view delivery [10], and thus blocks applications briefly before installing new views. This reflects in the following interfaces (Figure 5):

Membership. Describes a view of the group. This can be used to obtain a ranked list of all members, whose sort order depends on the implementation but which should be the same everywhere. It can also be used to obtain information on the event leading to the view change, namely, which processes have just been included and excluded and why.

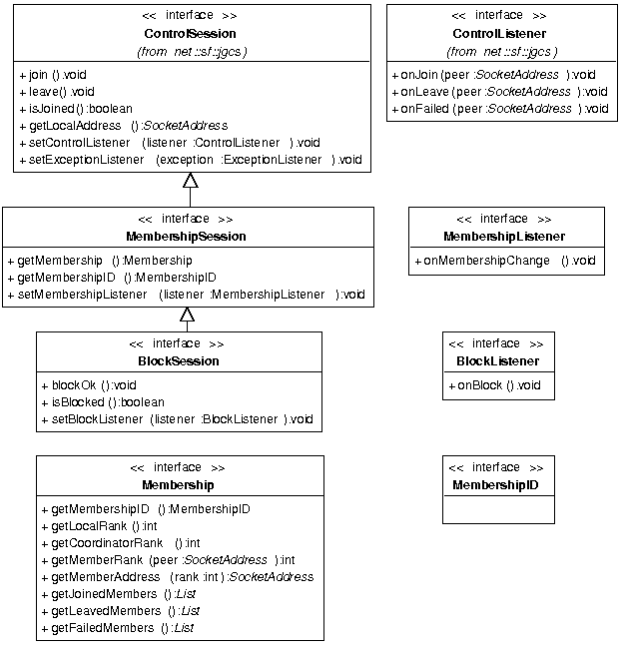


Fig. 5. Interface extensions for view synchrony

MembershipID. Provides an opaque unique identifier of the view, suitable for being exchanged and stored persistently. This can be obtained from the currently installed Membership object.

MembershipSession. Provides methods to obtain the current membership and register the callback for view change events.

MembershipListener. Handles notifications of view change.

BlockSession. Used only by implementations enforcing sending view delivery, providing methods for signaling that the application has blocked and that view change can proceed.

BlockListener. Handles requests by the protocol for the application to block.

Support for view synchronous group communication requires that membership notifications are coordinated with message and service notifications performed by the corresponding data session. In detail, the implementation must ensure that the view change notification is mutually exclusive with any other view dependent event, namely, message delivery and service ensured callbacks. This means that no other notification is issued concurrently with the view change. Although protocol implementations might allow this restriction to be lifted, this should be possible only by explicitly selecting a configuration option. On the other hand, block notifications can be issued without any concurrency restrictions. This means that it is up to the application to synchronize with any other active threads.

4 jGCS Bindings

jGCS was implemented in several group communication toolkits and primitives: Appia [2], JGroups [4] and Spread [3]. To validate the generality of the service, the jGCS was also implemented using IP Multicast and NeEM [9]. All these bindings are open source and available on SourceForge.net[3]. These toolkits and the implementations are described in the following paragraphs.

Appia Binding

Appia [2] is a layered communication support framework that was implemented in the University of Lisbon. It is implemented in Java and aims at high flexibility to build communication channels that fit exactly in the user needs. The QoS offered by a channel can be statically configured by an XML file or dynamically assembled by the application at run time. The application can create several channels with different QoSs and send messages to different channels, depending on the QoS required by each message. In contrast with traditional layered protocols, components of Appia channels can be shared and thus offer multiple related Qualities of Service (QoS). This makes it easy, for instance, that several channels can be bound to the same group membership.

Although Appia is protocol independent, in the sense that it can be used to compose any protocol as long as it respects the predefined interface, it includes an extensive layer library targeted at view synchronous group communication. Namely, it has protocols that implement virtual synchrony, causal order, and several implementations of total order algorithms.

The implementation of jGCS is build directly on Appia's protocol composition interfaces as an additional layer. jGCS configuration objects thus define the micro-protocols that will be used in the communication channels. Each Service identifies an Appia channel and messages are sent through the channel that fits the requested service. As Appia supports early delivery in totally ordered multicast, this is exposed in the jGCS binding using the ServiceListener interface. Appia implements all extensions of the ControlSession, depending on the channel configuration.

JGroups Binding

JGroups [4] is a group communication toolkit modelled on Ensemble [18] and implemented in Java. It provides a stack architecture that allows users to put together custom stacks for different view synchronous multicast guarantees as well as supporting peer groups. It provides an extensive library of ordering and reliability protocols, as well as support for encryption and multiple transport options. It is currently used by several large middleware platforms such as JBoss and JOnAS.

The JGroups implementation of jGCS also uses the configuration interface to define the micro-protocols that will be used in the communication channel. JGroups can provide only one service by the applications, since configurations only support one JGroups channel per group communication instance. JGroups implements all extensions of the ControlSession.

[3] jGCS and its bindings are available in `http://jgcs.sf.net`

Spread Binding

Spread/FlushSpread [3] is a toolkit implemented by researchers of the Johns Hopkins University. It is based on an overlay network that provides a messaging service resilient to faults across local and wide-area networks. It provides services ranging from reliable message passing to fully ordered messages with delivery guarantees. The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services. Although there are interfaces for Spread in multiple languages, these do not support the FlushSpread extension, which provides additional guarantees with a different interface.

The Spread and FlushSpread implementations of jGCS use the configuration interface to define the location of the daemon and the group name. The implementation to use (FlushSpread or just Spread) is also defined at configuration time. In Spread, the quality of service is explicitly requested for each message, being thus encapsulated in Service configuration objects.

Other Bindings

To prove the generality of jGCS, we provide also two implementations, based on the well known IP Multicast and on the Network-friendly Epidemic Protocol (NeEM) [9]. The NeEM protocol is an epidemic multicast protocol (also called probabilistic or gossip-based) in wide-area networks that uses multiple TCP/IP connections in a non-blocking fashion. The resulting overlay network is automatically managed by the protocol. The implementations of jGCS that use IP Multicast and NeEM allow peers to join and leave the multicast group, and send and receive messages to/from other peers. One application that uses only these functionalities can easily be ported to other implementations.

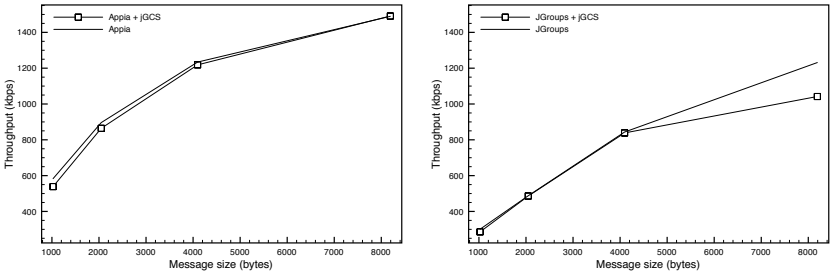
5 Performance Evaluation

We have done a number of experiments to assess the overhead imposed by the use of jGCS to wrap different group communication toolkits. Namely, we want to assess the impact of the extra level of indirection between the application and the toolkit introduced by jGCS.

For this purpose we have made two different sets of tests. In the first set we have made standalone throughput measurements for two different toolkits, both with and without the jGCS. In the second set of tests we have integrated jGCS in a production environment, namely in the Sequoia database clustering middleware.

5.1 jGCS for Appia and JGroups

To measure the impact of the jGCS on the maximum throughput of existing group communication toolkits we have selected Appia and JGroups. To run the experiments, we



(a) Appia throughput. (b) JGroups throughput.

Fig. 6. Throughput of the toolkits with and without the jGCS

have implemented three different versions of a test application that transmits a number of messages of a configurable payload size to the group. One version uses the Appia native interface, other uses the JGroups native interface, and the last version uses jGCS. This allowed us to run four different configurations: *(i)* the test application with Appia; *(ii)* the test application with JGroups, *(iii)* the test application with jGCS, configured to use Appia and *(iv)* the test application with jGCS, configured to use JGroups.

Measurements were obtained with the following environment. The JGroups and the Appia protocol stack were created using similar configurations. All tests used a virtual synchrony protocol stack and a token based total order protocol. All tests were made with a group of three members, each member sending 10 000 totally ordered messages to the group. Each member of the group runs in a Pentium IV/2.8GHz server with 1Gb of memory. The three machines are connected through 100Mbps ethernet switch. Each test was made with different message sizes.

The Figure 6 shows the throughput of the two group communication toolkits, using directly the interface provided by the toolkit and using jGCS. As we can see in 6(a), the Appia implementation of jGCS does not cause a significant overhead and this overhead is increasingly less noticeable as the message size grows. In the case of JGroups, in 6(b), the overhead caused by the jGCS is also very small but it grows as the message size increases. This is explained as follows: For improved performance, JGroups delivers messages in a buffer that can be reused later by the protocol, forcing the application to locally copy data during delivery. The native JGroups test application does not do this copying, and thus has better performance. On the other hand, the current jGCS binding does this copying in order to provide the same service as other bindings and thus incurs in additional overhead. In the future, this decision should probably be left to the configurator, thus making it possible to achieve the same performance as with the native interface.

5.2 jGCS in Sequoia

The second set of tests measure the overhead of having jGCS in a real application. To do these tests we used Sequoia, a middleware database replication system that exports a JDBC interface to applications and routes client requests to a set of databases. Sequoia

Table 1. Latency of client requests of TPC-W (ms)

Implementation	Mean	Std. Dev.	Samples
Native	39.96	41.10	3846
With jGCS	40.26	52.97	3832

is composed by a JDBC driver, that is used by applications that want to access the databases and a controller that receives the client requests and forward them to a set of databases. For availability and fault tolerance, the Sequoia controller can (and should) be replicated. Each controller manages a set of databases. In a system with more than one controller, the application can use any controller to make the requests. The controllers exchange their requests using view synchronous total order, to execute the same set of requests in the same order in all databases.

The implementation of primitives that make use of group communication is distributed as a separate package, Hedera (formerly ObjectWeb Tribe). In detail, it provides access to an application specific subset of group communication and additional functionality for explicitly acknowledged messages, multiplexing and dispatching. Hedera has been previously implemented twice, using JGroups and Appia. We thus ported Hedera to jGCS which allowed us to use Sequoia with any jGCS implementation that supports the required service guarantees.

Performance figures were obtained in a system configured as follows. The clients are a Java implementation of the TPC-W[4] that use the Jakarta Tomcat to make requests to a database. The requests are made to a sequoia controller that replicates the requests. Sequoia is configured to use three controllers, each one controls one MySQL database. The emulated browsers of the benchmark used and Tomcat run in one machine. The other three machines have one instance of the Sequoia controller and one instance of the MySQL database each. All four machines are connected by a 100Mbps ethernet switch and have the same memory and processing power of the machines used in the previous tests. In these tests, the benchmark was configured to have allways 20 clients (emulated browsers) making requests to the database, in the Ordering Mix (50% of write operations).

Table 1 shows that latency results of test system using the Appia toolkit, either through the native interface or through jGCS, are in practice the same. In fact, the difference is not statistically relevant, even with a very low confidence level, as confidence intervals overlap significantly. This shows that the use of jGCS is negligible in the overall performance of a complex system.

6 Related Work

Although there have been multiple attempts to ease the development of applications based on group communication by standardising their interfaces and their semantics, most of these efforts have quite different goals and, therefore, can be seen as complementary (instead of competing) efforts. We discuss four non mutually exclusive approaches.

[4] `http://www.ece.wisc.edu/~pharm/tpcw.shtml`

The first approach attempts to hide the complexity of group communication by wrapping it in higher level abstractions. The rationale for this line of work is that there is a category of users that would like to benefit from the advantages provided by group communication (namely, easy maintenance of consistent global states) but that do not want to invest in understanding the semantics of view synchronous communication. A particularly popular approach is to wrap view synchronous communication in RPC-like interface, such as RMI [23]. While the above may be true, using such interfaces often introduce a significant performance overhead that is unacceptable for programmers attempting to build high-performance applications such as replicated database middleware. In our view, the widespread use of the Java programming language and the broad adoption of several design patterns used in jGCS allow to conciliate the programmer familiarity with the satisfaction of performance constraints.

A second line of work attempts to fit the view synchronous interface into widely adopted interfaces such as, for instance, SNMP messages [20] or JMS [24,25]. The rationale for this line of work is that a view synchronous services can be easily adopted if exported using an interface that is familiar to most programmers. We see this category as a complementary line of work, given that the simplicity is achieved at the cost of loosing some of the benefits of both paradigms. Namely, many JMS applications rely of persistence or transactional services that are not provided directly by group communication toolkits. On the other hand, the notion of explicit membership, a keystone of view synchronous communication, conflicts with the goal of decoupling publishers from subscribers. Therefore, group communication can be a valuable tool to increase the reliability of messaging services but, certainly, one service cannot simply replace the other.

A third alternative aims at standard semantics for view synchronous communication [26]. The rationale for this type of work is that it would be easier to see a wider adoption of view synchronous communication toolkits if all provide the same guarantees. Although this is a valid goal, practice has shown that it is hard to make the community converge on such an common semantics, given that each toolkit exploits a different angle to provide better performance for some distinct target application areas. Instead of trying to define unique semantics, jGCS only defines minimum common semantics and provides the necessary hooks to support specific features using a flexible interface.

A final approach is to make the service provided by the group communication toolkit highly reconfigurable, usually through the composition of micro-protocols [18,2]. The rationale for this type of work is that applications are better served if the toolkit can be tailored exactly to their specific needs. Toolkit reconfiguration is an important topic. Our own work with the Appia toolkit has addressed this facet extensively. However, existing protocol composition mechanisms are typically tied to concrete language or run-time constructs from which the application should be shielded. The configuration interface of the jGCS decouples the way the application expresses its requirements from the mechanisms used to configure the underlying toolkit.

The work with jGCS is unique in the sense that it provides a low-level interface to view-synchronous communication, that allows the implementation of high-performance higher level abstractions while, at the same time, promotes a level of decoupling

between the application and the underlying toolkit, that allows the application to be portable. Higher level primitives, such as state transfer, multiplexing and dispatching, or explicit aknowldgement, commonly found in group communication wrappers as Maestro [27] or Hedera should be built on jGCS services.

7 Conclusions

This paper describes a generic interface for group communication to be used as a service to the applications – the Group Communication Service, or simply jGCS. The paper presents the goals to achieve and features to avoid and presents the interfaces and how should it work in order to achieve the desired goals.

Results show that the jGCS interfaces can be implemented using most of the state of the art group communication toolkits. It is also shown that the overhead caused by the jGCS service is negligible and do not affect real applications, improving modularity and configurability. This service was implemented in Java and is hosted at SourceForge.net (`http://jgcs.sf.net`).

References

1. Birman, K., van Renesse, R., eds.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press (1993)
2. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, IEEE (2001) 707–710
3. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: IEEE International Conference on Dependable Systems and Networks. (2000)
4. Ban, B.: Design and implementation of a reliable group communication toolkit for java (1998)
5. Vicente, P., Rodrigues, L.: An indulgent uniform total order algorithm with optimistic delivery. In: Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS'02), Osaka, Japan (2002) 92–101
6. Sousa, A., Pereira, J., Moura, F., Oliveira, R.: Optimistic total order in wide area networks. In: Proc. 21st IEEE Symposium on Reliable Distributed Systems, IEEE CS (2002) 190–199
7. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing Journal (2002) 97–107
8. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically reliable multicast: Definition, implementation and performance evaluation. IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems **52** (2003) 150–165
9. Pereira, J., Rodrigues, L., Monteiro, M.J., Oliveira, R., Kermarrec, A.M.: NeEM: Network-friendly epidemic multicast. In: Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03), Florence,Italy (2003) 15–24
10. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33** (2001) 427–469
11. Guerraoui, R., Schiper, A.: Software-based replication for fault-tolerance. IEEE Computer **30** (1997) 68–74

12. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: The VLDB Journal. (2000) 134–143
13. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Eternal - a component-based framework for transparent fault-tolerant corba. Software Practice and Experience (2002)
14. Pedone, F., Schiper, A.: Optimistic atomic broadcast. In: Proceedings of the 12th International Symposium on Distributed Computing (DISC'98). (1998)
15. Sussman, J.B., Keidar, I., Marzullo, K.: Optimistic virtual synchrony. In: Symposium on Reliability in Distributed Software. (2000) 42–51
16. Rodrigues, L., Mocito, J., Carvalho, N.: From spontaneous total order to uniform total order: different degrees of optimistic delivery. In: Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Dijon, France, ACM (2006)
17. Alur, D., Crupi, J., Malk, D.: Core J2EE Patterns: Best Practices and Design Strategie. Prentice Hall / Sun Microsystems Pres (2001)
18. Hayden, M.: The Ensemble System. PhD thesis, Cornell University, Computer Science Department (1998)
19. Hiltunen, M., Schlichting, R., Wong, G.: Implementing integrated fine-grain customizable QoS using Cactus. In: Fast Abstracts, The 29th International Symposium on Fault-Tolerant Computing Systems, Madison, Wisconsin, USA (1999) 59–60
20. Wiesmann, M., Défago, X., Schiper, A.: Group communication based on standard interfaces. In: Proc. 2nd Int'l Symp. on Network Computing and Applications (NCA-03), Cambridge, MA, USA, IEEE (2003) 140–147
21. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proceedings of 19th International Conference on Distributed Computing Systems (ICDCS'99). (1999)
22. Mocito, J., Respício, A., Rodrigues, L.: On statistically estimated optimistic delivery in wide-area total order protocols. Technical Report DI/TR/2006, University of Lisbon (2006)
23. Montresor, A., Davoli, R., Babaoglu, O.: Group-enhanced remote method invocations (1999)
24. Amir, Y., Munjal, A.: (Jms4spread: http://www.spread.org/jms4spread/)
25. SoftWired: (ibus/messagebus: http://www.softwired.ch/)
26. Kupsys, A., Pleisch, S., Schiper, S., Wiesmann, M.: Towards JMS compliant group communication - a semantic mapping. In: NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium on (NCA'04), Washington, DC, USA, IEEE Computer Society (2004) 131–140
27. Birman, K., Friedman, R., Hayden, M.: The Maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Ithaca, NY, USA (1997)

Optimizing Pub/Sub Systems by Advertisement Pruning

Sven Bittner* and Annika Hinze

Department of Computer Science
The University of Waikato, New Zealand
{s.bittner, a.hinze}@cs.waikato.ac.nz

Abstract. Supporting advertisements in large publish/subscribe services has an improving influence on the system scalability. The additional application of event routing optimizations based on subscriptions further increases the event throughput and decreases the memory usage of publish/subscribe systems. However, also advertisement-based optimizations can improve these system properties but have been widely neglected so far. In particular, no current optimizations are applicable to advertisements defined as arbitrary Boolean filter expressions.

In this paper, we outline the final milestone on the way to practically support the arbitrary Boolean publish/subscribe model: We firstly develop a novel optimization approach, advertisement pruning, that is tailored to arbitrary Boolean advertisements. Secondly, we present a detailed evaluation of the optimization potential of this advertisement-based solution. We finally analyze the effects of additionally optimizing subscriptions on the advertisement pruning optimization.

1 Introduction

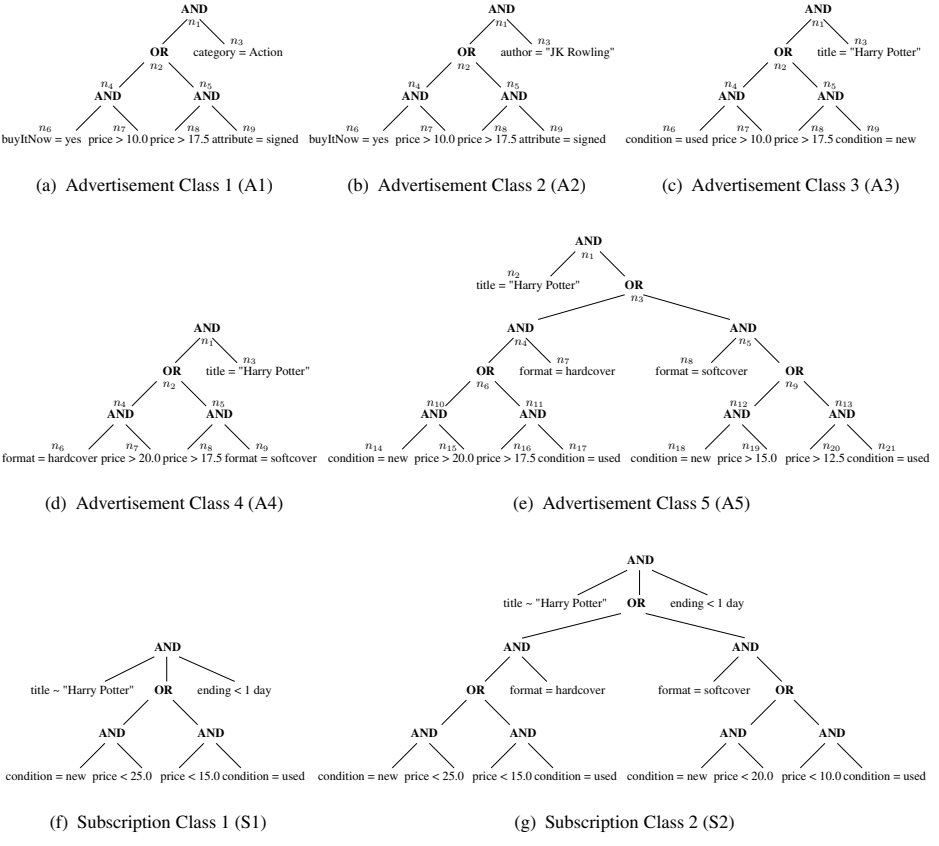
Content-based publish/subscribe (pub/sub) systems aim at an efficient and scalable dissemination of incoming information to all interested clients (we refer to [6] for an overview of the various facets of pub/sub systems in general). One way to increase the efficiency and scalability of these systems is the utilization of routing optimizations based on subscriptions. Another, orthogonal approach is the support of advertisements to decrease the amount of subscriptions forwarded in the network. The combination of these two methods eventually leads to optimizations based on advertisements.

In this paper, we present the first optimization method for content-based pub/sub systems that is applicable to arbitrary Boolean advertisements, advertisement pruning. We also show the results of a detailed analysis we have undertaken to investigate the influence of this optimization on various system parameters. For this evaluation, we define meaningful subscriptions and advertisements that are based on an analysis of real-world online book auctions [2]. Typical attributes of event messages in this scenario include the *Title*, the *Author*, the *Condition*, and the *Price* of book copies. We have given the structure of some example advertisements and subscriptions from our online book auction scenario in Fig. 1 (we refer to Sect. 5.1 for their detailed descriptions).

Our work is motivated by a recently proposed filtering and optimization approach for arbitrary Boolean subscriptions [3]. Symmetrically supporting arbitrary Boolean advertisements is the next step in providing the means for the arbitrary Boolean pub/sub

* This author has been partially funded by the NZ government under the NZIDRS scheme.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1503–1521, 2006.
© Springer-Verlag Berlin Heidelberg 2006



(a) Advertisement Class 1 (A1) (b) Advertisement Class 2 (A2) (c) Advertisement Class 3 (A3)

(d) Advertisement Class 4 (A4) (e) Advertisement Class 5 (A5)

(f) Subscription Class 1 (S1) (g) Subscription Class 2 (S2)

Fig. 1. Examples of Advertisement Classes 1 to 5 (A1–5) and Subscription Classes 1 to 2 (S1–2)

model [1]. The final milestone is now to introduce an optimization for these advertisements. Our novel solution is tailored to be applied to advertisements and has thus a higher optimization potential than previous optimizations for (restricted conjunctive) advertisements [4,9]. This stems from the fact that recent approaches have been tailored to subscriptions and are just symmetrically applied to advertisements.

We structured the rest of this paper as follows: After showing the inapplicability of related optimization approaches in Sect. 2, we briefly review the works that are building the foundations of our later optimization proposal in Sect. 3. Section 4 then successively develops the advertisement pruning optimization. Our experimental evaluation is presented in Sect. 5; we finally conclude and present future work in Sect. 6.

2 Related Work: Current Optimizations for Advertisements

In the literature, one can hardly find any optimizations that are based on advertisements. Instead, routing optimizations for subscriptions are suggested to be applied to advertisements as well. These optimizations are subscription covering [4] and merging [9]. Both

covering and merging have, however, strong assumptions on the registered subscriptions, and require similarities and relationships among these subscriptions to lead to an optimization (we refer to [3] for an analysis and description of these drawbacks).

Imperfect merging does not show this strong dependency on subscriptions/advertisements and may have a higher optimization potential than perfect merging [13]. One can further improve imperfect subscription merging by incorporating knowledge from advertisements [8]. However, there are no approaches tailored to advertisements.

These facts describe the general problem of existing advertisement optimizations: They are either independent of their application, i.e., they do not exploit whether they are applied to subscriptions or advertisements. Or, the optimizations have been specifically developed for subscriptions and cannot be successfully applied to advertisements.

As a result, meaningful evaluations of advertisement optimizations can hardly be found in existing literature: SIENA [4] supports subscription and advertisement covering in its routing protocols. However, this work does not answer the question of the influence of advertisement covering on any system parameter. The same does hold for HERMES [11] that, however, only supports little expressive type-based advertisements.

Some other analyses consider the existence of advertisements and evaluate the effect of subscription-based optimizations on the routing load: REBECA [9] only analyzes the application of subscription covering and merging in combination with advertisement-based routing. Also PADRES [8] does not consider the optimization of advertisements. We are generally not aware of direct analyses of advertisement-based optimizations.

3 Subscription Pruning and the Boolean Pub/Sub Model

In this section, we review the existing subscription pruning routing optimization in Sect. 3.1. Subscription pruning is the foundation of the later proposed advertisement optimization. We also present the basics of calculating the overlapping relationships among arbitrary Boolean subscriptions and advertisements in Sect. 3.2.

We outline our optimization solution on the basis of acyclic network structures (or the created minimum spanning trees for cyclic networks), as assumed by most prototypes targeting at efficiency aspects, e.g., [4,5,8,9]. In this paper, we do thus not aim at, e.g., pervasive environments [12] or automatic topology reconfigurations [10]. Instead, we assume the well-studied subscription forwarding scheme [4] in combination with the attribute/value pair pub/sub model. Extensions of our approach to P2P settings, e.g., used in [11], and other data models, e.g., XML-based [5,12] ones, remain future work.

3.1 Review of Subscription Pruning

The subscription pruning optimization has been introduced in [3]. Subscription pruning basically removes parts of subscription trees and is suitable for all kinds of Boolean subscriptions. The decisions about the removals of subtrees are based on heuristics.

One objective of subscription pruning is to decrease the memory usage for storing subscriptions by pruning parts of subscription trees. In doing so, the event filtering process also experiences efficiency gains due to the reduced subscription complexity. Subscription pruning, however, creates more general subscriptions and thus introduces

false positives, increasing the internal network load. These false positives are, although, not forwarded to subscribers because local brokers do not perform pruning for their local subscriptions. Thus, subscription pruning does not affect correct event delivery.

The decisions about pruning operations are based on a heuristic that is estimating their effects on the network load. This estimation takes into account the increases in selectivity induced by pruning operations, referred to as selectivity degradation. Ordering the possible pruning operations by these degradations then allows the system to perform pruning operations in ascending order of their effect on the network load.

The estimation of selectivities is based on incoming event messages. The selectivity of predicates is directly known by counting the number of matching messages. For newly registered predicates, one can apply estimation methods similar to [7]. For Boolean subscriptions, the selectivities are estimated based on the used operators [3].

3.2 Overlapping Among Arbitrary Boolean Subscriptions and Advertisements

Symmetrically to subscriptions, advertisements should be defined as arbitrary Boolean expressions. An important relationship among subscriptions and advertisements is the concept of overlappings: A subscription S overlaps an advertisement A (and vice versa) if and only if A describes at least one message that fulfills S. In [1], we have introduced an algorithm to efficiently determine these overlappings. We briefly sketch it here because it is required to understand the later proposed advertisement optimization.

The algorithm for the determination of the overlappings among Boolean subscriptions and advertisements uses the concept of violating predicates. The violating predicates $P_{vio}(n_l)$ of a leaf node n_l (containing predicate p_A) of an advertisement A include every predicate p_S used in a subscription S that (i) refers to the same attribute as p_A, and (ii) there exists no attribute value leading to *true* when evaluating both p_A and p_S. One can easily calculate these violating predicates for every predicate by looking up the one-dimensional indexes required for event filtering. One can then recursively calculate the violating predicates based on the utilized operators (we refer to [1] for details).

Based on these violating predicates, one can determine all candidate overlapping subscriptions/advertisements by counting the violating predicates and comparing this number to a structure-dependent parameter of subscriptions/advertisements (minimally required number of fulfilled predicates [1]). For all candidates, one finally just needs to evaluate whether the tree structure can still result in *true* when all violating predicates evaluate to *false*. If this is the case, one has found an overlapping.

4 Advertisement Pruning: An Optimization for Advertisements

The general idea of advertisement pruning is as follows: The pruning algorithm is trying to decrease the complexity of advertisements, in respect to the space complexity for storage and the time complexity for processing, without, in case of advertisements, strongly increasing the amount of overlappings. In this section, we describe how to achieve these goals. We show the inapplicability of the existing subscription pruning heuristics for advertisements in Sect. 4.1. In the remaining subsections, we then propose how to tackle the advertisement pruning problem (Sect. 4.2 to 4.4) and how to find the most suitable pruning operation among all advertisements (Sect. 4.5).

4.1 Inapplicability of the Subscription Pruning Heuristics

One could image that the heuristic to decide on the next subscription pruning can be directly mapped onto the advertisement pruning problem. However, these optimizations follow different goals, and the existing heuristic is thus not the applicable:

The main focus of subscription pruning is to decrease the selectivity of subscriptions as little as possible, i.e., each pruning should generalize subscriptions to the least extend. The determination of these selectivities is based on the incoming messages. When applying advertisement pruning, we want to increase the overlappings as little as possible. This is, at a first glance, independent of the incoming event messages. Although, we can partially utilize the selectivities, as shown in the following subsections.

What is required to prune advertisements is rather a correlation between subscriptions and advertisements because their overlapping relationships are altered by the pruning process (optimally, as minimal as possible). Additionally, we do not need to evaluate event messages against advertisements. Even the sole consideration of the predicates of advertisements while filtering would unnecessarily decrease the event throughput. Thus, we cannot determine the selectivities of the predicates of advertisements and, consequently, not of advertisements themselves. But also when assuming these selectivities to be given, simply minimizing their degradation (cf. Sect. 3.1) when pruning has no relation to the registered subscriptions and thus to the overlapping relationships.

Hence, advertisement pruning does require a completely different heuristic estimation approach than subscription pruning. We successively propose a possible heuristic in the following subsections. In Sect. 5, we then evaluate it in detail.

4.2 Discovering the Influences on the Overlappings

In this section, we subsequently identify the factors affecting the overlappings. We then incorporate these factors into a heuristic estimating these influences in Sect. 4.3.

The algorithm to determine the overlappings uses the concept of violating predicates (cf. Sect. 3.2). The number of these predicates is then used to compute candidate subscriptions. The less candidates exist, the more efficient the overlapping calculation.

Hence, an advertisement pruning should increase the amount of violating predicates. Such an increase is, however, impossible because a pruning [3] always removes some predicates and thus the corresponding violating predicates. Thus, a pruning needs to target at removing as little violating predicates as possible to, in turn, enlarge the number of candidates as little as possible (Influence 1). Although, if only considering the number of candidates, we could still strongly increase the overlappings. In worst case, before the pruning no candidate represents an overlapping, but afterwards all candidates overlap. Hence, it is also important what violating predicates are removed (Influence 2).

Ideally, the removed violating predicates do not influence the fact whether a subscription and an advertisement overlap. Or, to put it the other way round, the remaining violating predicates should still disqualify subscriptions and advertisements from overlapping. Hence, pruning operations must not remove predicates from advertisements that have violating predicates preventing subscriptions from overlapping. Making rational assumptions about the usage of predicates, we can base this decision on the selectivities of predicates. We elaborate on this proposal in the next subsection.

4.3 Characterizing an Arbitrary Boolean Advertisement

Having identified the influences of prunings on overlappings, we now develop a heuristic incorporating these factors. The first step is to quantify an *overlapping rank* for advertisements. By the help of this rank, we can then estimate the effects of prunings (Sect. 4.4). This, finally, allows for the determination of the best pruning (Sect. 4.5).

The overlapping rank combines the number of violating predicates of an advertisement (*quantitative overlapping rank*) with the influence of these predicates on the overlappings (*qualitative overlapping rank*). That is, it incorporates both of the previously identified influences (Sect. 4.2). The overlapping rank can be successively calculated for an advertisement tree based on the utilized operators, as shown in the following.

Leaf nodes. For a leaf node n_l of an advertisement tree, the quantitative overlapping rank includes the number of violating predicates of the predicate stored in n_l and the number of subscriptions using these predicates. For the qualitative overlapping rank, we have to make assumptions about the usage of predicates in subscriptions: Generally, subscriptions contain a number of predicates having different selectivities. For example, predicates on *Title* or *Author* show a high selectivity (restrictive predicates), whereas predicates on *Condition* or a generally low price have a low selectivity (general predicates). In practice, there is the tendency that highly selective predicates stronger determine whether a subscription tree might be fulfilled (in respect to both matching messages and overlapping advertisements). Thus, to state it the other way round, the more selective a predicate in subscriptions, the more important its state of fulfilment.

Putting together these observations with the importance of violating predicates when evaluating candidates, we should try to remove little selective violating predicates rather than highly selective ones. The qualitative overlapping rank should thus incorporate the selectivity of violating predicates: The higher selective a violating predicate, the higher the corresponding qualitative overlapping rank. We define the overlapping rank $rank(n_l)$ of a leaf node n_l as follows. It is always between 0 and 1:

$$rank(n_l) = \frac{1}{|predSubAssoc|} \sum_{p_i \in P_{vio}(n_l)} \frac{predSubAssoc(p_i)}{\sqrt{matches(p_i)+1}} \ .$$

The expression in the right multiplication factor sums up the overlapping rank for all violating predicates p_i of the leaf node n_l. The elements in the sum contain the quantitative part $predSubAssoc(p_i)$, describing the number of predicate subscription associations of a violating predicate p_i, i.e., how many subscriptions contain p_i. The qualitative part is given by the denominator, stating the selectivity of p_i. The number of matchings per predicate, $matches(p_i)$, is known from the selectivity estimation for subscriptions[1].

The left coefficient of $rank(n_l)$ ensures that the overlapping rank is always between 0 and 1. The value $|predSubAssoc|$ describes the total number of predicate subscription associations. The case $rank(n_l) = 1$ occurs if a leaf node n_l has all registered predicates (from subscriptions) as violating predicates with a selectivity of zero each.

This definition of the overlapping rank assigns higher values to nodes with a large number of violating predicates. These violating predicates are weighed by their selectivity, i.e., their importance to disqualify a candidate from being an overlapping.

[1] It can also be estimated for newly registered subscriptions and predicates, as outlined in [3].

For the Boolean operators in advertisement trees, we only estimate the overlapping rank, as shown in the following paragraphs. For each tree node n_i, this estimation contains three values, the *minimal possible overlapping rank* ($rank^{min}(n_i)$), the *average overlapping rank* ($rank^{avg}(n_i)$), and the *maximal possible overlapping rank* ($rank^{max}(n_i)$). For a leaf node, these three estimations are the same.

Conjunctive nodes. For a conjunctive node n_c, we only estimate the overlapping rank. This is required due to the general lack of information about the relationships among violating predicates. This approach, more importantly, allows for a time and space efficient calculation of the required overlapping rank. This estimated overlapping includes both concepts of a rank, the qualitative and the quantitative part.

The minimal possible overlapping rank $rank^{min}(n_c)$ occurs if all violating predicates are shared among the children of n_c. It is thus their maximal rank. The average rank $rank^{avg}(n_c)$ estimates a mean value for the ranks of the children. It describes the expected mean if assuming independent child nodes and an equiprobable distribution of their violating predicates. Finally, the maximal possible rank $rank^{max}(n_c)$ occurs if the violating predicates of child nodes exclude each other. It is thus at most the sum of the ranks of all children but further restricted to 1. This leads to the following equations for the three components. The calculations are always based on the estimation of the child nodes. For brevity, here we only consider the case of two children, n_1 and n_2:

$$rank^{min}(n_c) = \max(rank^{min}(n_1), rank^{min}(n_2)) \ .$$
$$rank^{avg}(n_c) = rank^{avg}(n_1) + rank^{avg}(n_2) - (rank^{avg}(n_1) \times rank^{avg}(n_2)) \ .$$
$$rank^{max}(n_c) = \min(1.0, rank^{max}(n_1) + rank^{max}(n_2)) \ .$$

Disjunctive nodes. For the other kind of inner nodes, disjunctions n_d, we also apply an estimation approach to determine the overlapping rank. It again contains the qualitative and the quantitative part because it is based on the combined rank of the child nodes.

The minimal possible rank is described by the smallest rank of the child nodes of n_d. The reason is that at least this rank will occur, regardless of which part of the disjunction will lead to the overlapping. The average rank of n_d again treats child nodes independently of each other and assumes an equiprobable distribution of violating predicates. The maximal overlapping rank describes the situation that all children of n_d hold simultaneously and that their violating predicates exclude each other. This leads to the following equations. Again, we only consider the case of two children, n_1 and n_2, here:

$$rank^{min}(n_d) = \min(rank^{min}(n_1), rank^{min}(n_2)) \ .$$
$$rank^{avg}(n_c) = rank^{avg}(n_1) + rank^{avg}(n_2) - (rank^{avg}(n_1) \times rank^{avg}(n_2)) \ .$$
$$rank^{max}(n_d) = \min(1.0, rank^{max}(n_1) + rank^{max}(n_2)) \ .$$

Having the means to estimate the overlapping rank, we can now determine the effect of a pruning operation, as shown in the following subsection.

4.4 Estimating the Influences of a Pruning on the Overlappings

In the last subsection, we have defined the overlapping rank, heuristically estimating a measure for the overlappings among advertisements and subscriptions. By the help of

this measure, we can now estimate the influence of a pruning operation. The question to be answered is as follows: Given a set of advertisements, what is the order of pruning operations to perform. That is, we firstly have to determine the best pruning of each advertisement. Secondly, we need to compare prunings of different advertisements.

As identified previously, each performed pruning operation should minimally influence the amount of overlapping subscriptions for the pruned advertisement. Because our overlapping rank estimates this relationship among advertisements and subscriptions, a pruning should minimally change, i.e., decrease, the overlapping rank.

To describe the influence of prunings, we should use a relative rather than an absolute measure. This helps to weigh the same change higher for a small overlapping rank than for a large one. That is, if there is only a small amount of violating predicates, the influence of removing some of them on the overlappings is higher than for removing the same number of predicates from an overall large number of violating predicates.

We refer to the influence of a pruning of advertisement A_i to A_j as *overlapping rank degradation*, $\Delta rank(A_i, A_j)$. It is defined as follows (the rank of an advertisement A_k always equals the rank of its root node n_k, e.g., $rank^{min}(A_k) = rank^{min}(n_k)$):

$$\Delta rank(A_i, A_j) = \max(\frac{rank^{min}(A_i) - rank^{min}(A_j)}{rank^{min}(A_i)}, \frac{rank^{avg}(A_i) - rank^{avg}(A_j)}{rank^{avg}(A_i)}, \frac{rank^{max}(A_i) - rank^{max}(A_j)}{rank^{max}(A_i)}) \ .$$

This definition weighs the change in the overlapping rank relatively to the existing rank before pruning and thus fulfils our requirements. The final step is now just to relate all prunings with each other to determine the pruning order, as shown in the following.

4.5 Determining the Best Pruning and Pruning in Practice

Our measure for the overlapping rank degradation allows us to find the best pruning for each advertisement: We just need to calculate all possible prunings and compare their degradations. The pruning that is leading to the least degradation should be performed.

Analogous to subscription pruning, the only valid pruning is to remove a child of a conjunction in an advertisement tree because it creates a more general advertisement, potentially increasing the overlappings. Removing the child of a disjunction, conversely, results in a more restrictive advertisement. This pruning would thus reduce the amount of overlappings and introduce false negatives, opposing the pruning idea. For the last option, removing the root node, the semantics is that all subscriptions would overlap the completely removed advertisement, opposing the overall application of advertisements.

Practically, whenever an advertisement A_i is registered we calculate the best possible pruning operation leading to A_j, i.e., the pruning that is resulting in the least degradation $\Delta rank(A_i, A_j)$. We then insert the tuple $(\Delta rank(A_i, A_j), A_i)$ in the *advertisement degradation queue*. This queue implements a priority queue that is ordered in an ascending way by the first element of the tuple, the overlapping rank degradation.

To perform a pruning, we just need to extract the top element of the advertisement degradation queue. This operation works in a constant time due to our decision about the data structure. We then perform the best pruning of the corresponding advertisement A_i, resulting in A_j, and re-insert the new best pruning of A_j into the queue. According to the goal of the optimization, e.g., to decrease the memory usage or to increase the time efficiency, we successively perform the prunings that are stated by the top elements of the advertisement degradation queue. The pruning stops when the required optimization has been achieved, e.g., the memory usage has been reduced by a certain percentage.

The most important question emerging from our advertisement pruning proposal is the influence of prunings on the memory usage, the efficiency to calculate overlappings, and the number of overlappings. We now extensively evaluate these properties.

5 Experimental Analysis

In this section, we describe our evaluation of an extensive set of experiments. In Sect. 5.1, we characterize our experimental setup to allow for the classification and repeatability of our experiments. The influence of advertisement pruning on time efficiency, space efficiency, and overlappings is analyzed in Sect. 5.2. Section 5.3 finally evaluates these three parameters when combining advertisement and subscription pruning.

5.1 Experimental Setup

We have identified eight classes of advertisements (A1 to A8) and three classes of subscriptions (S1 to S3) for our experiments. These classes of subscriptions and advertisements describe reasonable interests (subscriptions) and potential event messages (advertisements) in our online book auction scenario [2]. These classes are as follows:

A1. A publisher offers books of a certain category. These books are Buy-It-Now items that are further specified by a minimal price, or signed book copies that are also stating a minimal price. We have illustrated an example advertisement of A1 in Fig. 1(a). It specifies category "Action", and prices of NZ$10.0 and NZ$17.5.

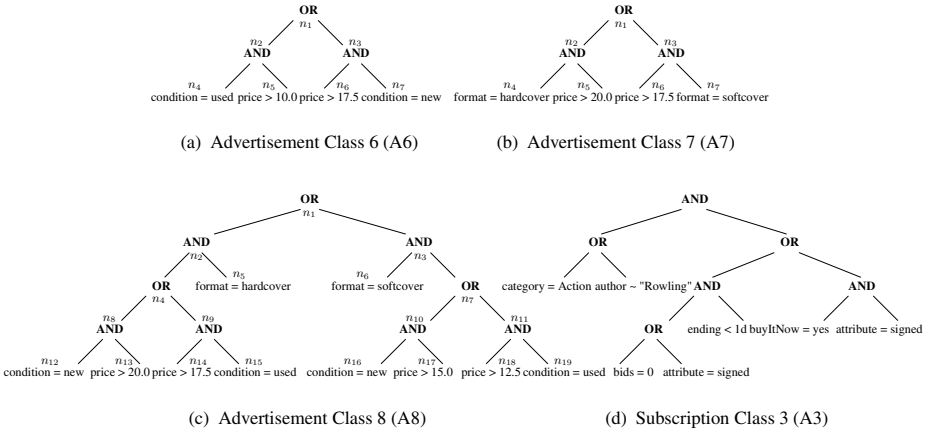
A2. This class is similar to A1 but specifies authors instead of categories. The example in Fig. 1(b) states author "JK Rowling", and prices of NZ$10.0 and NZ$17.5.

A3. A publishers offers books of a particular book title. There are different minimal prices for used and new copies of the book. In the example in Fig. 1(c), we have chosen "Harry Potter" as title and, again, prices of at least NZ$10.0 and NZ$17.5.

A4. This class is similar to A3 but specifies different prices for hardcover and softcover copies of the book. Figure 1(d) shows an example of this class stating prices of NZ$20.0 and NZ$17.5 for books entitled "Harry Potter".

A5. This class combines A3 and A4. That is, the publishers offers different prices for the four combinations of hardcover and softcover, as well as new and used book copies. Our example in Fig. 1(e) again refers to the title "Harry Potter" and specifies prices of at least NZ$12.5, NZ$15.0, NZ$17.5, and NZ$20.0.

A6. The publisher offers used and new books, stating minimal prices (similar to A3 except of the title). In the example in Fig. 2(a), these prices are NZ$10.0 and NZ$17.5.



(a) Advertisement Class 6 (A6) (b) Advertisement Class 7 (A7)

(c) Advertisement Class 8 (A8) (d) Subscription Class 3 (A3)

Fig. 2. Examples of Advertisement Classes 6–8 (A6–8) and Subscription Class 3 (S3)

A7. This class describes a publisher offering softcover and hardcover books of different minimal prices (similar to A4 except of the title). In Fig. 2(b), these prices have been chosen with NZ$17.5 and NZ$20.0.

A8. Similar to A5, the publishers specifies different minimal prices for the four combinations of used and new book copies, as well as hardcover and softcover books. There are no further restrictions on the books. The example in Fig. 2(c) specifies prices of at least NZ$12.5, NZ$15.0, NZ$17.5, and NZ$20.0.

S1. A subscriber is interested in a particular book title and specifies different maximal prices for used and new copies of books. Notifications should only be sent if the auction ends in less than one day. We have illustrated an example subscription of this class in Fig. 1(f). It specifies that the book title should contain the phrase "Harry Potter", and that the book copy should cost less than NZ$15.0 or NZ$20.0.

S2. In addition to S1, the subscriber specifies different prices for the four combinations of used and new books, as well as hardcovers and softcovers. Our example in Fig. 1(g) again expresses interest in books entitled "Harry Potter" that are offered for at most NZ$10.0, NZ$15.0, NZ$20.0, or NZ$25.0.

S3. A collector is interested in books from a certain author but also in books of a particular category. The books should either be signed and Buy-It-Now items, or end within one day if they are signed or have got zero bids so far. The example in Fig. 2(d) states the category with "Action" and the author with "JK Rowling".

We consider these advertisement and subscription classes, expressing a wide range of interests and potential event messages, as representative for our application scenario. In our experiments, the values of all predicates are chosen randomly using a uniform distribution of reasonable values (derived from the analysis in [2]). We have named all nodes n_i in the example advertisement trees (Fig. 1 and 2).

Our experiments have been performed on machines equipped with a 1.8 GHz CPU and 512 MB of main memory. All results shown in the following describe the mean values when registering 100,000 subscriptions and 50,000 advertisements. These large numbers of subscriptions and advertisements have led to stable mean values.

5.2 Analysis of Advertisement Pruning

We have analyzed the influence of advertisement pruning on all advertisement classes individually and on a combination of these classes to understand the effects of pruning in a wide range of settings. We graphically present the results in Fig. 3 and 4. We have always registered a combination of subscriptions of all classes in our experiments.

We show the proportion of pruning operations on the abscissae of all figures. It ranges from 0, describing the situation without pruning, to 1, stating that all possible pruning operations have been performed. The left ordinate shows the time efficiency, i.e., the average time to calculate the overlappings per advertisement. We show two curves, one describing the time to determine all overlapping subscriptions (Scenario All) and one testing whether at least one overlapping exists (Scenario First). The right ordinate states the proportion of overlappings, of removed predicate advertisement associations (our measure for memory usage), and of candidate subscriptions for a given amount of prunings (abscissa). These properties are represented by individual curves.

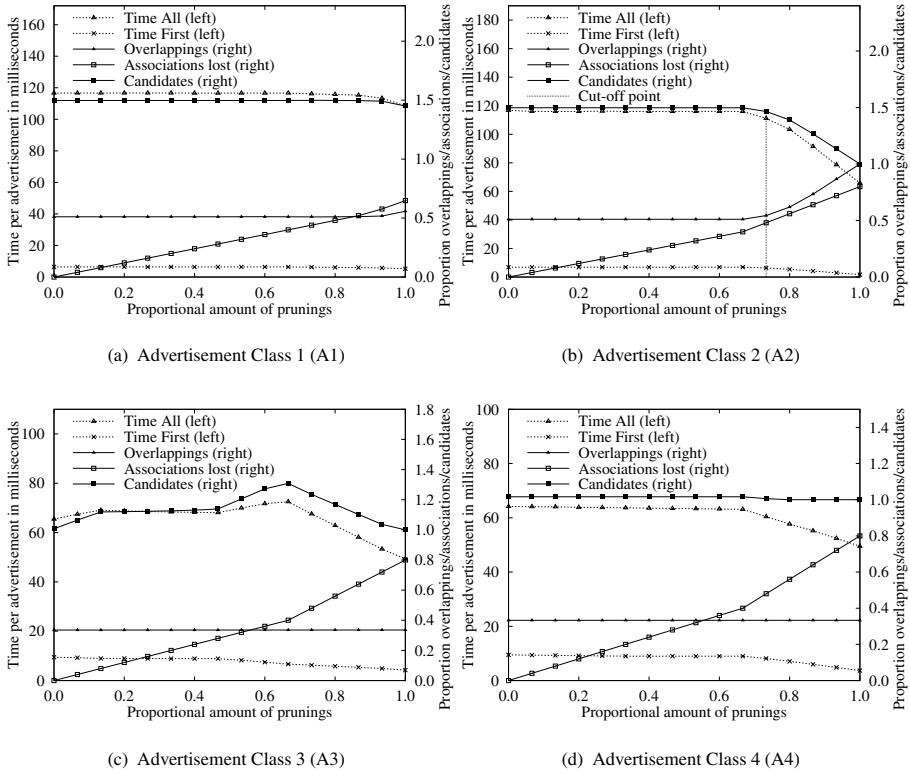
In some of our settings, we experience a cut-off point. At this point, the behavior of advertisement pruning changes from a worthwhile optimization to less valuable one, i.e, the overlappings increase strongly. We have given these cut-off points in the figures representing these settings. Our evaluation of advertisement pruning is as follows.

Advertisement Class 1. For A1 (Fig. 3(a)), the pruning optimization decreases the memory requirements by 65% after performing all prunings (removal of proportionally 0.65 predicate advertisement associations, shown at the right ordinate). Simultaneously, determining whether any overlapping exists (Scenario First) is performed 18% faster (time reduction from 6.4 to 5.2 msec, illustrated at the left ordinate); the calculation of all overlappings (Scenario All) works 7% more efficient (decrease from 117 to 109 msec per advertisement, shown at the left ordinate). These beneficial results are obtained by only increasing the overlappings from 51% to 56% (right ordinate) of all subscriptions.

The reason for this advantageous behavior is the initial pruning of predicates on *Buy It Now* (Node n_6, cf. Fig. 1(a)) and *Special Attribute* (Node n_9), having merely significance for S3. But even for S3, these prunings do not change the overlappings. The predicates on *Category* (Node n_3) are pruned later on, followed by the disjunctions (Node n_2), which only have predicates on *Price* (Nodes n_7 and n_8) as children anymore. This pruning of predicates on *Price* then creates the little increase in overlappings.

Advertisement Class 2. There exists a cut-off point at approx. 73% of all prunings operations for A2 (cf. Fig. 3(b)). At this point, the pruning algorithm has decreased the memory requirements for advertisements by 48%. The time efficiency has increased by 5% for Scenario All and by 8% for Scenario First. The total amount of overlappings has grown from 51% to 54% of all registered subscriptions.

The reason for this behavior is the advertisement and subscription structures: Up to the cut-off point, the algorithm only prunes on *Buy It Now* (Node n_6, cf. Fig. 1(b)) and *Special Attribute* (Node n_9), which does not influence the overlappings (cf. A1). All of these predicates have been pruned at the cut-off point. Later on, the algorithm prunes the disjunctive nodes (Node n_2). Hence, the only remaining predicate is on *Author* (Node n_3). All registered subscriptions are then overlapping due to their structure.



(a) Advertisement Class 1 (A1)

(b) Advertisement Class 2 (A2)

(c) Advertisement Class 3 (A3)

(d) Advertisement Class 4 (A4)

Fig. 3. Influence of advertisement pruning on Advertisement Classes 1 to 4 (A1 to A4)

Advertisement Class 3. We can decrease the memory requirements for advertisements by 80% when performing all possible pruning operations for A3 (Fig. 3(c)). Additionally, the efficiency properties improve strongly: The algorithm works 55% faster for Scenario All; the computations for Scenario First are 24% more efficient. Also the amount of overlappings develops promising and increases by only 1%.

The reason for the stronger improvement in Scenario First compared to Scenario All is the development of candidate subscriptions: The number of candidates increases when pruning. All of these candidates need to be evaluated to determine all overlappings. Hence, the reduction in advertisement complexity (due to prunings) does not have the same strong effect as in case of Scenario First. After removing the disjunctive node (Node n_2, cf. Fig. 1(c)), the amount of candidates decreases again. This is due to the definition of violating predicates for disjunctive nodes [1], determining the candidate subscriptions. Also the memory requirements decrease more strongly from this point onwards because the algorithm removes two associations per pruning operation.

The reason for the advantageous slight increase in the overlappings is again found in the advertisement and subscription structure: S3 is not influenced by any prunings because it specifies predicates on other attributes only. S1 and S2 use the same attributes as

A3. Hence, pruning should affect the overlappings but only does so insignificantly: The most selective indicator for overlappings is the *Title* attribute (Node n_3). Subscriptions and advertisements have to specify the same title for an overlapping. The overlapping further depends on the prices (Nodes n_7 and n_8) according to the condition (Nodes n_6 and n_9). For most subscriptions either the prices for used or new books lead to an overlapping. Thus, pruning any of these predicates does hardly change the overlappings.

Advertisement Class 4. A4 (Fig. 3(d)) has a similar structure as A3. Thus, we experience a similar behavior when pruning: The memory requirements decrease by 80% when performing all prunings. This strong improvement creates an additional amount of overlappings of less than 1% of all registered subscriptions (always around 33%). The efficiency in Scenario All is increased by 23%; Scenario First is improved by 61%.

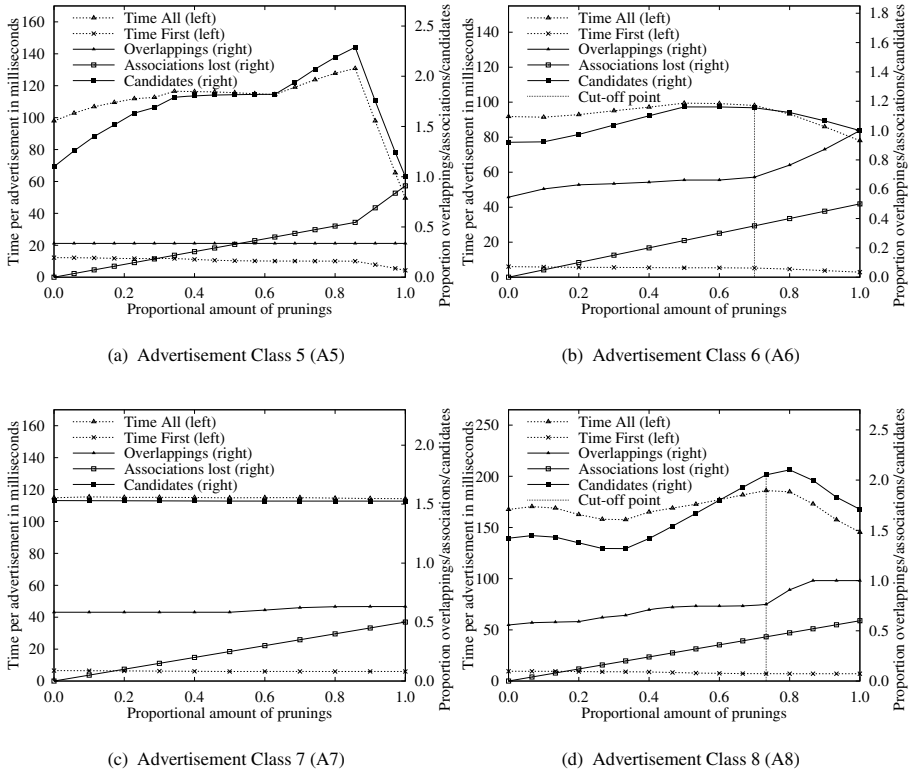
The reason for this beneficial behavior is again found in the subscription and advertisement structure: Pruning on *Format* (Nodes n_6 and n_9, cf. Fig. 1(d)) does not affect the amount of overlappings to a large extend. This is due to the *Title* attribute (Node n_3), mandatorily requiring to specify the same value in an advertisement and a subscription to lead to an overlapping. If this is given, in most cases either the respective prices for hardcover or softcover items overlap (S2), or one of these prices overlaps any of the specifications in subscriptions (S1). When pruning the disjunctive node (Node n_2), we again experience a stronger decrease in memory usage than when pruning leaf nodes (bend in curve). The efficiency also increases more strongly from this point onwards.

Advertisement Class 5. A5 (Fig. 4(a)) leads to the best results when applying advertisement pruning: The memory requirements have been decreased by 91% after performing all possible pruning operations. At this point, the computations in Scenario All can be performed 49% faster. For Scenario First, the improvement is even 65%. The pruning algorithm achieves these advances with increasing the overlappings by only 1%.

However, pruning firstly degrades the efficiency in Scenario All. The reason is again found in the number of candidates, showing a similar increase as the overall time. This increase in candidates is due to the performed prunings: In the beginning, the algorithm chooses to prune on *Format* (Nodes n_7 and n_9, cf. Fig. 1(e)). This results in more candidates of S2. Then, mainly predicates specifying low prices are pruned (also some predicates on new books). This results in an increase in candidates of S1 and S2. However, at some point the candidates only increase slowly anymore because mainly predicates on new books (Nodes n_{14} and n_{18}) are pruned (decreasing the violating predicates much less than pruning on *Price*). If most of these predicates on *Condition* have been pruned, the algorithm again decides to prune on *Price*. This leads to the newly developing larger gradient of the curve in Fig. 4(a). Finally, the disjunctive nodes (Node n_3) are pruned, resulting in a dropping amount of candidates (only one predicate remains (Node n_2)).

Advertisement Class 6. For A6 (Fig. 4(b)), we again experience a cut-off point. This time, it occurs after having performed 70% of all pruning operations. At this point, the memory requirements could be reduced by 35%; the overlappings are increased from 55% to 68%. In Scenario First, the algorithm works 13% more efficiently at the cut-off point. However, Scenario All requires a 6% higher computation time.

When performing all prunings, each subscription overlaps every advertisement, but the computation works faster than without pruning. The reason is that after more than



(a) Advertisement Class 5 (A5)

(b) Advertisement Class 6 (A6)

(c) Advertisement Class 7 (A7)

(d) Advertisement Class 8 (A8)

Fig. 4. Influence of advertisement pruning on Advertisement Classes 5 to 8 (A5 to A8)

70% of all prunings, mostly one predicate on *Condition* and one on *Price* remain. Because S1 and S2 always specify the interest in both conditions, all subscriptions finally overlap. The overlappings with S3 are not influenced by any prunings. The first increase in the computation time in Scenario All is again due to the increase in candidates. After the cut-off point, the amount of candidates decreases and thus the computation time. Altogether, more overlappings exist for A6 than for A3: A3 additionally restricts titles.

Advertisement Class 7. There are again strong benefits of advertisement pruning when using A7 (Fig. 4(c)): After performing all pruning operations, the memory usage is half of the original one, and the overall increase in overlappings is only 5%. Additionally, the efficiency in Scenario First is increased by 8% and in Scenario All by 1%.

The amount of overlappings is nearly unaffected by pruning operations for A8 because only predicates on *Format* (Nodes n_4 and n_7, cf. Fig. 2(b)) are considered to be pruned (similarly to A4). Firstly, the algorithm prunes hardcover specifications (Node n_4), which does not affect the overlappings (hardcover books are generally more expensive than softcover ones). When pruning softcover predicates (Node n_7), the amount of overlappings increases slightly. In fact, it only increases if the hardcover part of an advertisement does not overlap the subscription but the lower price from the softcover.

Advertisement Class 8. We experience a cut-off point for A8 (Fig. 4(d)). It occurs at approx. 73% of all prunings. The pruning algorithm decreases the memory usage at this point by 44%. The total number of overlappings is increased by 21%. The computations in Scenario First work 25% faster; however, in Scenario All they degrade by 11%.

This efficiency degradation in Scenario All is again caused by the increase in candidate subscriptions. Similar to A5, the pruning algorithm firstly removes predicates on *Format* (Nodes n_5 and n_6, cf. Fig. 2(c)), increasing the candidates of S2. However, in contrast to A5 there is no predicate on *Title* in A8. This property also increases the amount of overlappings and finally leads to the situation of all advertisements overlapping all subscriptions (when pruning more than specified by the cut-off point).

Advertisement Classes 1–8. The effect of advertisement pruning in this setting is not merely the mean of the results for the individual classes. It rather describes the quality of our heuristic when applied to different advertisement structures simultaneously.

For this general setting, the cut-off point occurs at approx. 77% of all prunings. At this point, the algorithm has decreased the memory usage by 49%; the overall amount of overlappings has increased by only 5%. The efficiency properties in Scenario First are improved by 24%; Scenario All, however, leads to a decrease by 3%. We do not show the detailed developing of the measured parameters here due to space restrictions.

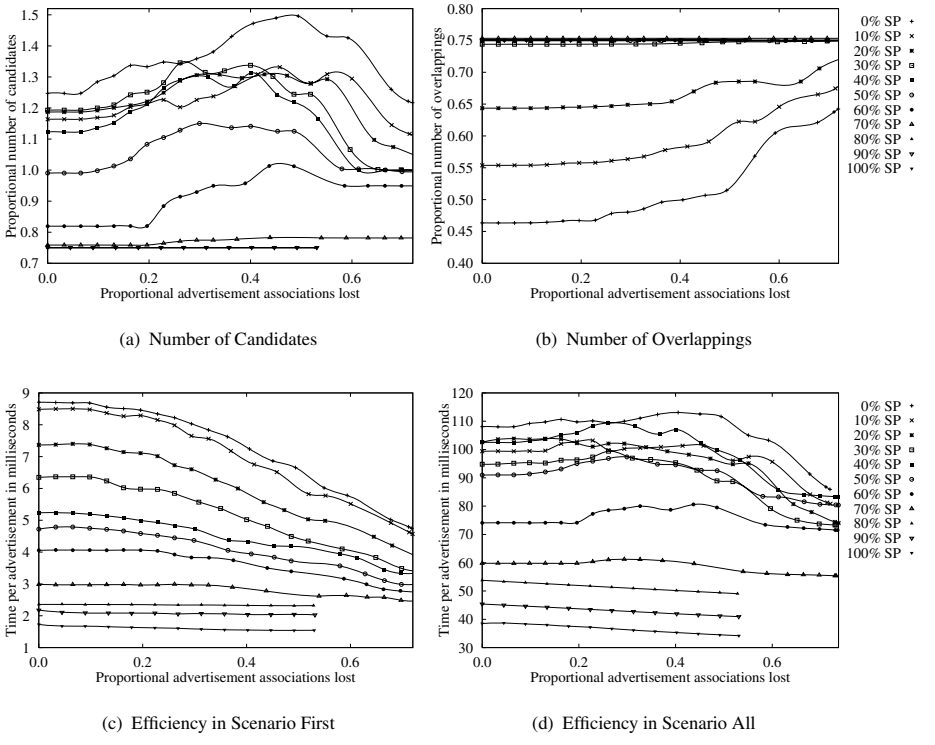
Our combined setting shows that our pruning heuristic does also work if registering advertisements involving different structures. The promising results at the cut-off point—and in particular its position—show that our heuristic chooses pruning operations for advertisements of different classes and correctly judges their influence on the overlappings. In fact, the algorithm firstly performs those pruning operations that do not strongly increase the overlapping properties. The pruning operations increasing the overlappings are only performed if no other suitable pruning opportunities exist anymore, which is leading to the cut-off point in the combined setting.

5.3 Combining Advertisement and Subscription Pruning

Our evaluation in this paper has only considered the influence of advertisement pruning so far. Subscription pruning has been analyzed in [3]. These independent evaluations have shown that the two approaches are valuable optimizations if applied individually. What is still missing is to analyze the effects of utilizing these subscription-based and advertisement-based optimizations at the same time. We will do so in this section.

For our evaluation, we consider the general setting containing subscriptions and advertisements of all classes. We analyze the influence of advertisement pruning after having performed different amounts of subscription pruning. This varies from no subscription pruning (0% SP) to all possible subscription prunings (100% SP). In contrast to our previous figures, the abscissae now state the proportion of reduced predicate advertisement associations, i.e., memory usage (also shown as a curve before).

In Fig. 5(a), we have illustrated the proportional number of candidate subscriptions, strongly influencing the efficiency in Scenario All. Generally, the more subscription prunings one performs (represented by the different curves), the less candidates exist. After only some subscription prunings, this behavior is not that stable yet and still depends on the advertisement prunings as well. But after approx. 50% of all subscription



(a) Number of Candidates

(b) Number of Overlappings

(c) Efficiency in Scenario First

(d) Efficiency in Scenario All

Fig. 5. Candidates, overlappings, and efficiency when applying both optimizations

prunings, the amount of candidates clearly decreases more and more. From 80% of all subscription prunings onwards, the candidates remains stable at 75%.

The reason for this decrease in candidates is the removal of predicates in subscriptions, leading to more general subscriptions. Obviously, more general subscriptions do overlap more advertisements than more restrictive subscriptions. Hence, the amount of candidates decreases because subscriptions are mostly evaluated only once.

The increase in overlappings when performing subscription pruning is illustrated in Fig. 5(b). Without subscription pruning, the proportion of overlappings is 46%. Having performed all subscription prunings, 75% of all subscriptions overlap. This is a rather strong increase, showing that subscription pruning does not incorporate the influence on the overlappings. Advertisement pruning, however, only slightly increases the overlappings: When reducing the amount of predicate advertisement associations by 72%, the overlappings increase by 18%. But reducing the amount of predicate subscription associations by only 30% results in an increase in overlappings of even 29%. A sound property of subscription in combination with advertisement pruning is, however, the stabilization of the overlappings at 30% of all subscription prunings. Then, neither advertisement nor subscription pruning influences the overlappings anymore. Thus, one could perform all remaining prunings without further increasing the overlappings.

The influence of subscription pruning on the time efficiency in Scenario First is shown in Fig. 5(c). Here, we can clearly observe the improving influence of subscription pruning on the efficiency properties: The more subscription prunings one performs, the more efficient the determination whether overlappings exist. Additionally, the more subscription prunings have been executed, the less influence has the application of advertisement pruning: Without any subscription pruning, advertisement pruning improves the efficiency in Scenario First by 46%. Having performed all possible subscription prunings, the improvement is 11%. Altogether, these results show that both subscription and advertisement pruning improve the efficiency to determine overlappings.

The time efficiency to calculate all overlappings is finally shown in Fig. 5(d). Similar to the development of the candidates (Fig. 5(a)), subscription pruning generally improves the efficiency properties after a stabilization phase: Having performed more than 50% of all subscription prunings, Fig. 5(d) clearly shows a steadily improving efficiency. Without any advertisement pruning, the time to calculate all overlappings has decreased from approx. 108 to 38 msec. Generally, this time develops similarly as the candidate subscriptions due the interdependence of these two parameters. So, up to 70% of all subscription prunings, advertisement pruning shows an uneven development of the calculation times (similar to the development of the candidates). When performing more than 70% of all subscription prunings, the time efficiency when additionally applying advertisement pruning improves linearly. This is due to the stable amount of candidates (cf. Fig. 5(a)) in combination with the decreasing advertisement complexity.

Summarizing these results, we can state that subscription pruning leads to an optimization when applied in combination with advertisement pruning. However, subscription pruning does not focus on the influence of prunings on the overlappings (it focusses on minimally increasing the network load for event routing). Thus, it increases the amount of overlappings stronger than advertisement pruning when reducing the memory usage to the same amount. The increase in overlappings stops after having performed a certain number of subscription pruning operations (approx. 30% of all possible ones). After this point, subscription pruning still increases the efficiency significantly and, simultaneously, reduces the memory requirements to index and store subscriptions.

6 Conclusions and Future Work

Current pub/sub systems apply optimizations for subscriptions to decrease the system load. Although one can partially utilize some of these optimizations for advertisements, no directly advertisement-based optimization approaches exist. Additionally, and even more severe, all current optimizations are only applicable to conjunctive advertisements, which prevents their utilization in the arbitrary Boolean pub/sub model.

In this paper, we presented the first optimization approach that is tailored to (Boolean) advertisements. Advertisement pruning is based on the subscription pruning routing optimization but directly considers the influences of pruning decisions on the overlapping properties among subscriptions and advertisements. We have incorporated these influences into the measure of an overlapping rank, allowing us to estimate the influences of prunings. This, finally, gives us the means to determine the best advertisement pruning to perform, i.e., the pruning that increases the overlappings the least.

The remainder of this paper has been an extensive evaluation of the advertisement pruning optimization. For our experiments, we have defined characteristic online auction test settings. We could show that advertisement pruning is a valuable optimization in all of our test settings that, at the same time, is (i) strongly reducing the memory requirements to store advertisements, (ii) moderately increasing the efficiency to determine whether overlappings exist, and (iii) only reasonably increasing the amount of overlappings. We could reduce the memory usage by 49% and improve the efficiency by 24% with only increasing the overlappings by 5% in the most general setting.

The last part of our evaluation has focused on the effects of simultaneously applying both subscription and advertisement pruning. We could show that subscription pruning also leads to an optimization if it is applied in addition to advertisement pruning. However, subscription pruning increases the amount of overlappings more strongly than advertisement pruning. This behavior is clearly due to the different focus of subscription pruning and shows that our estimation of the influences of advertisement prunings has been effectively encapsulated in the introduced, advertisement-tailored optimization.

Our results prove that optimizations for advertisements significantly reduce the memory usage of pub/sub systems while increasing their efficiency. In the future, we plan to analyze advertisement pruning in combination with other application scenarios to show its general applicability. Furthermore, we plan to fully support this optimization in our pub/sub prototype and directly analyze its effects in the distributed system.

References

1. S. Bittner and A. Hinze. Arbitrary Boolean Advertisements: The Final Step in Supporting the Boolean Publish/Subscribe Model. Technical Report 06/2006, Computer Science Department, The University of Waikato, June 2006.
2. S. Bittner and A. Hinze. Event Distributions in Online Book Auctions. Technical Report 03/2006, Computer Science Department, The University of Waikato, February 2006.
3. S. Bittner and A. Hinze. Subscription Tree Pruning: A Structure-Independent Routing Optimization for General-Purpose Publish/Subscribe Systems. Technical Report 01/2006, Computer Science Department, The University of Waikato, January 2006.
4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
5. R. Chand and P. A. Felber. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA 2003)*, pages 123–130, Cambridge, USA, April 16–18 2003.
6. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
7. M. Guimarães and L. Rodrigues. A Genetic Algorithm for Multicast Mapping in Publish-Subscribe Systems. In *Proc. of the 2nd IEEE International Symposium on Network Computing and Applications (NCA 2003)*, pages 67–74, Cambridge, USA, April 16–18 2003.
8. G. Li, S. Hou, and H.-A. Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, pages 447–457, Columbus, USA, June 6–10 2005.
9. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, September 2002.

10. G. P. Picco, G. Cugola, and A. L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *Proc. of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 234–243, Rhode Island, USA, May 2003.
11. P. R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambrigde, Queens' College, February 2004.
12. T. Sivaharan, G. Blair, and G. Coulson. GREEN: A Configurable and Re-configurable Publish-Subscribe Middleware for Pervasive Computing. In *Proceedings of the 7th International Symposium on Distributed Objects and Application (DOA 2005)*, pages 732–749, Agia Napa, Cyprus, October 31–November 4 2005.
13. Y.-M. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, and P. Larson. Summary-based Routing for Content-based Event Distribution Networks. *ACM SIGCOMM Computer Communication Review*, 34(5):59–74, 2004.

A Specification-to-Deployment Architecture for Overlay Networks

Stefan Behnel[1], Alejandro Buchmann[1],
Paul Grace[2], Barry Porter[2], and Geoff Coulson[2]

[1] Databases and Distributed Systems Group,
Darmstadt University of Technology (TUD), Germany
{behnel, buchmann}@dvs1.informatik.tu-darmstadt.de
[2] Computing Department, Lancaster University, UK
{gracep, porterbf, geoff}@comp.lancs.ac.uk

Abstract. Implementing overlay software is non-trivial and time-consuming. Current projects build overlays or intermediate frameworks on top of low-level networking abstractions. This leads to far reaching incompatibilities between overlay implementations, tight coupling to frameworks and limited adaptability to different deployment environments.

We present a new approach to rapid overlay implementation that combines a modelling framework for overlay design with a dynamic component architecture for run-time adaptation. It is the first architecture in the overlay area that tackles the complete design process from modelling, through code generation and implementation down to adaptive deployment. To demonstrate the effectiveness of this architecture, we describe the step-by-step procedure of designing an overlay and deploying it within an adaptive middleware framework.

1 Introduction

Recent years have seen a large body of research in decentralised, self-maintaining overlay networks like Chord [1], ODRI [2] or Gia [3]. They are commonly regarded as building blocks for Internet-scale distributed applications. Current overlay implementations are built with incompatible, language specific frameworks on top of low level networking abstractions. This complicates their design and hinders the comparison and integration of different topologies. Apart from a recently proposed API for the specific use case of structured overlay networks [4], there is little standardisation effort in the rest of the overlay area. And a common API by no means simplifies the design of the overlay implementation itself.

Currently, programmers who want to use overlays for their applications must decide in advance, at a very early design phase, which of the distinct overlay implementations they want to use and must invest time to understand its specific usage. This effectively prohibits testing the final product with different topologies or delivering versions with specialised overlays. There is no support for deployment time or run time adaptation to changing environments or quality-of-service requirements. Consequently, the actual usefulness of overlays for application design is currently very limited.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1522–1540, 2006.
© Springer-Verlag Berlin Heidelberg 2006

This exposes a clear requirement for the high-level specification, development and automated deployment of overlay networks. In this paper, we investigate a specific case of this. Gridkit [5] is a middleware framework that underpins different middleware services: event notification, group communication, media streaming, etc. with an array of overlay network types; this is because the deployment of these services can be across highly diverse environments, and hence they require different properties from an overlay network, e.g. in terms of QoS attributes like robustness, scalability, ability to handle particular traffic patterns, etc. Hence, Gridkit developers need to rapidly generate and evaluate overlays that will best support particular middleware configurations; for example, which overlay will best support multicast in an ad-hoc network; and which will best support large-scale data storage in the Internet.

Therefore, this paper investigates a high-level specification-to-deployment architecture for overlay networks. It aims to provide developers with clean models for overlay design, implementation support using a model driven approach, and run-time deployment and dynamic adaptation using a component-based infrastructure. To evaluate our solution, we document an example case study which illustrates how a complex overlay implementation for the Gridkit middleware is specified, developed and deployed by our architecture. We chose to implement the Scribe [6] multicast tree atop the Chord [1] key-based routing overlay.

In the remainder of this paper, we first examine in section 2 the highly complementary work effected by two research groups at Lancaster University and the University of Darmstadt; namely, the Gridkit middleware and the OverML overlay specification tools respectively. In section 3, we then present OverGrid, an architecture that completes the edit-compile-deploy loop for developing overlays that can underpin middleware services. Then in section 4, we exemplify the development of a particular overlay. Finally, in section 5, we describe related work in this area, and draw conclusions in section 6.

2 Background on Gridkit and OverML

2.1 Gridkit

Motivation for Overlay-Based Middleware. Gridkit[1] [5] is a reflective middleware whose key aim is to support application development for pervasive computing in the face of increasing diversity at both the infrastructure level and the "middleware service" level. There is diversity in the types of devices (workstations, mobile devices, sensors, and clusters) and in the types of networks employed (e.g high-speed LAN, infrastructure-based wireless networks, and ad-hoc wireless networks). Furthermore, the requirements of middleware have exploded in the range of "middleware service types" that must be available to application developers; beginning with basic point-to-point interactions (e.g. RPC and

[1] `https://sourceforge.net/projects/gridkit/`

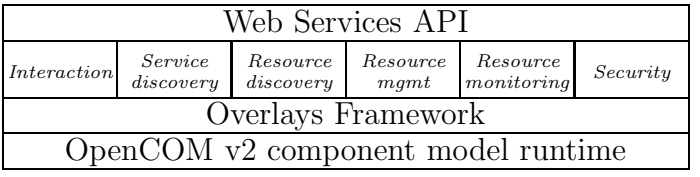


Fig. 1. The per-node Gridkit software framework

SOAP messaging), the range of interaction paradigms is expanding to include (amongst others): reliable and unreliable multicast; work flow; media streaming; publish-subscribe; tuple-space/ generative communication; and peer-to-peer based resource location or file sharing.

Gridkit's key approach to tackle these challenges is to deploy an extensive and extensible set of middleware services over an infrastructure of overlay networks that best support particular interaction types. For example, a publish-subscribe service can be built atop a multicast network service e.g. an application-level multicast overlay; and a tuple-space middleware can by layered atop a DHT key-based routing overlay e.g. Chord. Notably, one of the key research questions posed by Gridkit is how to select appropriate overlay networks to support individual middleware services.

Software Architecture. In terms of software architecture, the Gridkit framework (illustrated in figure 1) is deployed on each participating node of a middleware service. In this paper, we concentrate on the bottom two parts: the OpenCOM runtime and the overlays framework; for further information about the higher level services, see [5].

At the bottom level, Gridkit employs a minimal runtime for the loading and binding of lightweight software components, known as OpenCOM components [7]. Components are language-independent encapsulated units of functionality and deployment that interact with other components exclusively through interfaces and receptacles. Fundamentally, OpenCOM supports both reflection and component frameworks. Reflection is used to reason about component configurations and to dynamically alter configurations at runtime. Component frameworks are scoped compositions of components that accept plug-in components that are validated according to component framework specific constraint rules (cf. the overlay framework described later). A further key feature of the OpenCOM model is the support for component *caplets*. Essentially, these are separate address spaces into which heterogeneous component types can be loaded. The component runtime then manages the bindings between components in different caplets. For example, applications and systems software can easily be composed of Java, C++, python, and other OpenCOM components.

The layer above (the overlays framework) is a distributed framework for the deployment of multiple overlay networks. In practice, this amounts to hosting, in a set of distributed overlay framework instances, a set of per-overlay plug-in

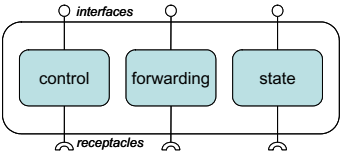


Fig. 2. The overlay framework control-state-forward specification

overlay components (see figure 2). Each of these represents a single overlay network node and consists of a *control* element that cooperates with its peers on other hosts to build and maintain some virtual network topology, a *forwarding* element that appropriately routes messages over its virtual topology, and a *state* element that contains per-overlay-node state such as a next-neighbours list.

The state component is specific to the overlay, and communicates through an overlay-specific state interface. Whereas, the control component and forwarding component implement the common interfaces *IControl* and *IForward* listed below. This is because overlays are plugged vertically into the framework using just the control and forward interfaces; state communication is horizontal. The control operations are: *Create* a specified overlay, *Join* an overlay or *Leave* an overlay. The forward operations are: *Send* forwards messages to an identified destination, *Receive* blocks awaiting messages from the overlay, and *EventReceive* does the same in a non-blocking style.

```
1  interface IControl {
2    ResultCode Create(String netId, Object params);
3    ResultCode Join(String netId, Object params);
4    ResultCode Leave(String netId);
5  }
6  interface IForward {
7    public byte[] Send(String destID, byte[] msg, int param);
8    public byte[] Receive(String netId);
9    public void EventReceive(String netId, IDeliver evHandler);
10 }
```

Cost of Developing Gridkit Overlays. Gridkit requires overlays that best support a middleware service in the current environmental conditions; hence, a key question to be answered is: which overlay meets the requirement set for this environment? To best answer this, multiple overlay types should be developed and tested to determine the right choice. Gridkit represents a considerable improvement on the basic approach of implementing overlays without a software framework. So far, we have implemented 7 types of overlay networks; this work has been carried out by 8 different developers. However, developing overlays in the Gridkit style is still a relatively expensive, time-consuming task. The following table illustrates both the length of code produced, and the time taken for each overlay to be developed. It can be seen that a new individual implementation will require a significant development expense.

Overlay Type	Code Lines	Man Days
Chord Key-based Routing [1]	790	42
Chord-based Distributed Hash Table [1]	570	35
Scribe[6]	880	42
Tree-Building Control Protocol [8]	7406	35
SCAMP [9]	1000	35
Minimum Spanning Tree	1300	20
Gossip-based Failure Monitor [10]	450	28

We further analysed the development process by examining the produced code, and questioning the developers. This exercise discovered these key points:

- Developers easily followed the framework structure e.g. the IForward and IControl interfaces i.e. the code behaved correctly for the create, join, leave, send, and receive operations. Hence, third-party overlay implementations were directly plugged in and out of the Gridkit framework.
- Developers sometimes had difficulty separating the overlay behaviour for control, forward and state into distinct components. It was noticeable that the code for these were often interleaved across the three components. Hence, this hinders the ability for a third party to effectively perform fine-grained dynamic reconfiguration e.g. replacement of the routing algorithm.
- The implementations of all the overlays are in Java. However, certain devices e.g. sensor motes cannot run a Java virtual machine. Hence, overlays would need to be reimplemented in order to operate on such devices. OpenCOM supports this level of portability, however, at present this requires overlay reimplementation and hence significant duplicated effort.

We therefore argue that the high-level development of portable overlay code is needed to overcome the identified problems, and provide the following benefits:

- Reduced development time, will aid experimentation of component configurations, leading to better supported middleware services.
- Generated overlay code will better follow the control-forward-state split, avoiding the high level of programmer skill needed for this task. The resultant implementation will then be easily reconfigurable by third parties.
- Portable overlay implementations can be generated for heterogeneous settings, reducing the duplication of overlay implementations.

2.2 OverML

The Overlay Modelling Language OverML [11] is an integrated set of domain specific XML specification languages for node attribute schemas, messages, view definitions, routing decisions and event graphs. There are obviously many different ways of specifying data schemas, messages and graphs. The main advantage of OverML is the clear focus on the domain of overlay design and the integration between the languages. This allows OverML models to provide a rich set of semantics for generative overlay implementation.

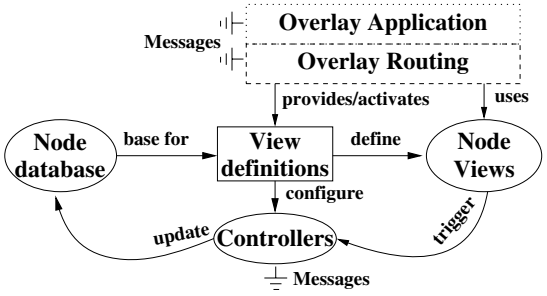


Fig. 3. Components of the OverML System Model

The architectural idea behind OverML is illustrated in figure 3. It clearly borrows from the Model-View-Controller Pattern for software design. The locally available knowledge about remote nodes (node attributes) and about the system state is stored in a local node database. View definitions select nodes from it according to topological rules and overlay adaptation strategies. The resulting database views present these nodes to software components. Note that the local database may well be incomplete or globally inconsistent. No guarantees are made on the data storage and representation side of the architecture.

Routers (or message forwarders) need these views for their forwarding decisions, controllers use them for their maintenance decisions. Controllers are event-triggered maintenance components that are integrated into the system at deployment time or run time. OverML event graphs are used to specify the connection between the events generated from views and messages, and the triggered controller actions that update the database or request new data from remote nodes.

The overlap between Gridkit and OverML should be clear from the figure. Just like Gridkit, OverML targets a clean separation of control, forwarding and state. However, its modelling approach allows it to impose further structure on these components and to leave a major part of their implementation to code generators and generic infrastructure components.

The OverML languages are briefly described here and in more detail in [11]. For brevity, **Nala** and **Himdel** , which specify node attributes and overlay messages respectively, are restricted to an exemplary description later on.

Slosl, the SQL-Like Overlay Specification Language. The view definitions for topology rules and adaptation are expressed in Slosl. We present it here using a simple example, an advanced implementation of the Chord graph [1].

```
1 CREATE VIEW chord_fingertable
2 AS SELECT node.id, node.ring_dist, bdist=node.ring_dist−2^i
3 FROM node_db
4 WITH log_k = log(|ℵ|), backups = 1
5 WHERE node.supports_chord = true AND node.alive = true
6 HAVING node.ring_dist in [2^i, 2^(i+1))
7 FOREACH i IN (0, log_k)
8 RANKED lowest(backups+i, node.msec_latency / node.ring_dist)
```

The first three clauses behave as in SQL. The WHERE clause specifically selects nodes based on their attributes. Note that SLOSL is not concerned with the source of the information that node attributes contain. This is left entirely to the controllers. SLOSL only constrains and categorises the presentation of locally available data. The remaining clauses do the following:

WITH This clause defines variables or options of this view that can be set at instantiation time and changed at run-time. Here, *log_k* is a global constant for the life-time of an overlay, while *backups* allows adding neighbour redundancy at runtime.

HAVING–FOREACH This pair of clauses aggregates the selected nodes into buckets (or equivalence classes) to categorise them. In the example, the (constant) node attribute *ring_dist* refers to the logical distance between the local node and the remote node. The HAVING expression states that it must lie within the given half-open interval (excluding the highest value) that depends on the bucket variable i. The FOREACH part defines the available node buckets by declaring this bucket variable over a range (or a list, database table, ...) of values. It defines either a single bucket of nodes, or a list, matrix, cube, etc. of buckets. The structure is imposed by the occurrence of zero or more FOREACH clauses, where each clause adds a dimension. Nodes are selected into these buckets by the (optional) HAVING expression. In the example, a Chord node sorts remote nodes into ring intervals of increasing size and distance that designate equivalent neighbours: itself $\rightarrow 2^1 \rightarrow 2^2 \rightarrow 2^3 \ldots$

RANKED To support topological adaptation and optimisation, the nodes in the `chord_fingertable` view are chosen by the ranking function *lowest* as the $(backups + i)$ top node(s) of each bucket that provide the lowest value for the given expression. Rankings are often based on the network latency, but any arithmetic expression based on node attributes can be used. The expression in the example implements a simple tradeoff between the network latency and the distance travelled in the identifier space. Other overlays may require more complex expressions or user defined functions in the ranking expression. The original Chord implementation selects exactly one node based only on its ring distance.

EDGAR, Extensible Decision Graphs for Adaptive Routing. EDGAR is a small graph language that models routing decisions. As opposed to other routing languages, it is not concerned with topology decisions. This is entirely left to SLOSL views. EDGAR only combines views and conditions into a decision graph. A general graph is shown in figure 4.

EDSL, the Event Driven State-Machine Language. EDSL is the language that connects the OverML specification with external controller components. It defines the event flow in the overlay implementation, including the protocol and maintenance strategies. EDSL is a special event-driven state machine language in that it knows about the events specified by the other OverML languages. States

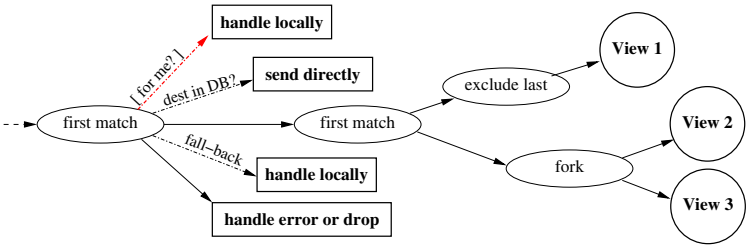


Fig. 4. General graph for EDGAR routing decisions

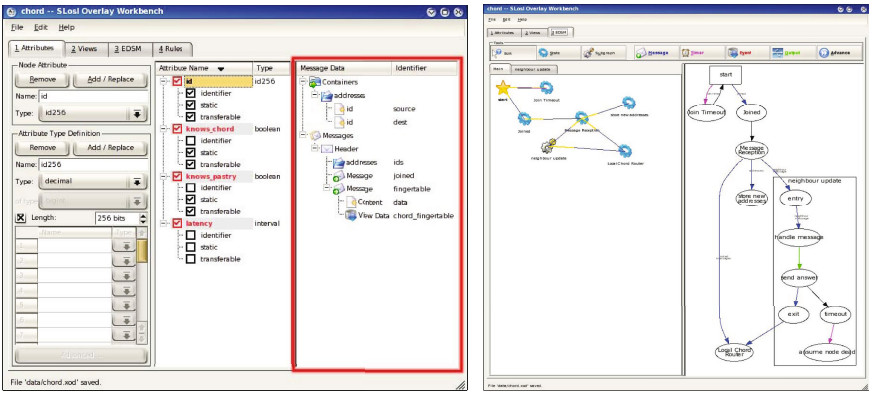
are defined using a number of generic interfaces that handle incoming messages or that respond to view events. Event flows are visually described as automata, as shown in figure 5. An important feature of EDSL is the support for subgraphs. This allows the creation of EDSL implemented complex components for database update strategies and maintenance, and their integration in different designs.

The SLOSL Overlay Workbench. Major work went into the implementation of a graphical overlay design tool, named the SLOSL Overlay Workbench. The screenshots in figure 5 give an idea about the interfaces for the different languages of OverML [11]. They support the semantically integrated specification of node attributes, messages, SLOSL view statements and component event flow graphs. EDGAR routing support is currently worked on. The workbench exports platform independent XML descriptions of the designed overlay, which can be passed into translators and source code generators. The development of the workbench is now continued as an Open Source project at the Berlios developer site:
`http://developer.berlios.de/projects/slow/`

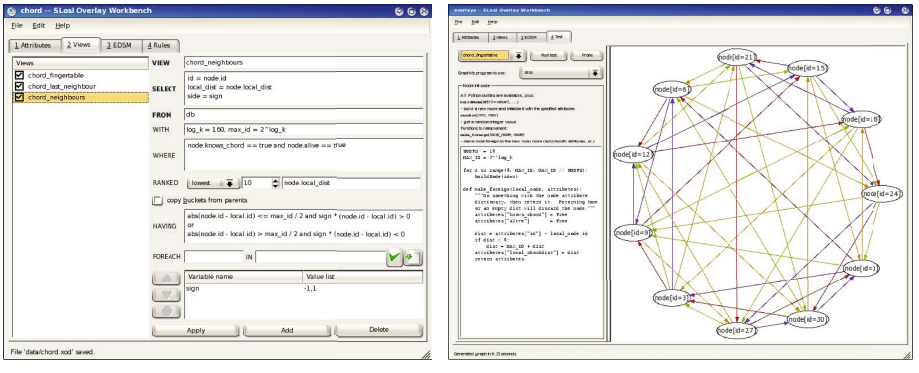
3 OverGrid – The Best of Both Worlds

OverML and Gridkit are very complementary approaches. OverML focuses on the design aspects of overlay implementations and tries to provide platform-independent models for model driven code generation. Gridkit, on the other hand, provides a dynamic component environment and a layered software architecture for run-time adaptation of overlay systems. Where OverML lacks the support for modelling and designing the controllers in source-level detail, Gridkit provides a dynamic software component infrastructure and generic, layered networking interfaces. Where Gridkit lacks the support for rapid overlay development and code portability, OverML excels with its platform-independent models and the generation of efficient, platform-specific source code.

The resulting architecture is presented in figure 6. The vertical separation follows Gridkit's interface layers. The horizontal separation represents the OverML implemented overlay architecture. The main idea is to use OverML for the platform-independent design of overlay topologies, routing strategies etc., and then generate very specialised Gridkit components and glue code from the model.



Attributes and Messages Event flows



SLOSL view definitions SLOSL visualisation

Fig. 5. A Glimpse at the SLOSL Overlay Workbench

We will now briefly overview major parts of the infrastructure that OverML requires in Gridkit: database and views, control and forwarding components, and event handling.

3.1 Database and SLOSL Views

OverGrid replaces individual overlay state components in Gridkit overlays with a database exposing views to the control and forwarder components. Hence, the number of executing components is reduced, and state is more easily shared across overlay implementations. This obviously requires a platform-specific SLOSL infrastructure in Gridkit, namely a database and a view evaluator. However, both of them are partially generated from OverML. The node attribute language (NALA) describes the data schema and SLOSL describes the evaluation of nodes and node attributes into views (or sets of nodes). In its simplest incarnation, the database can even be a hash table with node objects that are

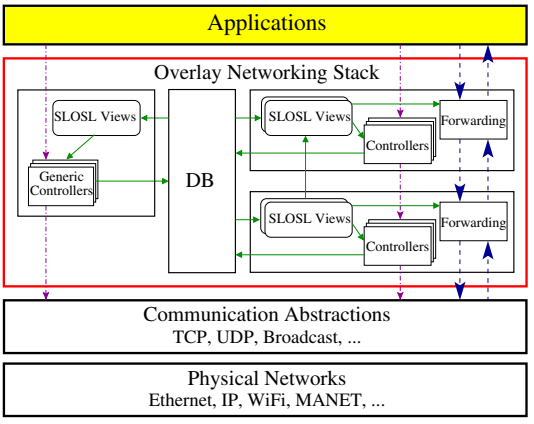


Fig. 6. The OverGrid Architecture

linearly scanned during evaluation. View updates can then be triggered by updates to the node objects. The trigger paths are entirely known through the dependencies of the SLOSL statements, which allows efficient view update code to be generated.

The tradeoff between the complexity of the code generation process and that of static components is up to the specific requirements of the target environment. Gridkit can easily provide different implementations and select between them at deployment time.

3.2 Control

Control components are generated mainly from the SLOSL and EDSL models. The resulting code is wrapped in an OpenCOM component that implements the IControl interface and interacts with the database through the modelled views. Currently, Gridkit provides general, re-usable overlay control components (termed generic controllers here), that provide repair and backup strategies for overlays [12]. These can be remodelled using EDSL to make their internal structure visible and subject to adaptation in OverGrid.

3.3 Forwarding

Forwarding is entirely driven by code generation. EDGAR is easily mapped to conditions in source code. SLOSL views, however, require the decision code to be executed in a SLOSL infrastructure or by code generated for SLOSL. This is mainly identical to normal SLOSL evaluation and thus integrates with the database implementation. OverGrid wraps the generated code within an individual OpenCOM component that implements the IForward interface, and binds to the state database.

3.4 Event Infrastructure

EDSL describes the component interaction in terms of events. Implementations can follow two possible paths:

EDSM. The implementation uses a generic event-driven state machine implementation. This can either interpret the EDSL graph directly or can be specialised by OverML code generators. Such a specialisation can be the generation of unique event IDs that speed up the dispatching process. It can also mean the generation of specific event objects that are forwarded between states, or of event-specific handler code. The exact implementation depends on the constrains of the target environment and the effort required for the specialisation of the code generators.

Interface code. Implementations can choose to generate glue code for components that directly interconnects them according to the event flow graph. The extreme incarnation is direct interaction of components through their generic interfaces. This requires run time setup code for OpenCOM that connects the correct components. Note that this approach allows for threaded implementations, such as one thread per event path.

3.5 Network Layers

The FROM clause in SLOSL describes a layering of topologies that maps directly to layers in Gridkit as in figure 6. In combination with EDSL event flow graphs (and its component subgraphs), this nicely describes the event flow through the network layers and the dependencies of local decisions on lower layers.

4 Case Study: Scribe over Chord

This section describes a case study of designing a complex overlay implementation with OverGrid. We specify the overlay using the SLOSL Overlay Workbench and then map the resulting OverML specification to Gridkit components. This simple walk-through does not honour the fact that design usually evolves incrementally. The real-world design process would normally follow edit-compile-test and edit-compile-deploy cycles. Due to space constraints, we only present the design steps in summaries. Note, however, that the support for incremental design is a major advantage of our high-level architecture.

4.1 Node Attributes for Scribe/Chord

The first step in the design process is to provide a database schema, expressed as node attributes in NALA. Chord nodes require logical IDs, which are essentially large integers. We define them as having 256 bits. We will see later on that the SLOSL views for Chord and Scribe require the following additional attributes:

ring_dist is a dependent attribute based on the id attribute. It contains the locally calculated ring distance towards a node based on the Chord metric.

supports_chord is a boolean flag that states whether a node is known to support the Chord protocol.

alive is another boolean flag that is only true for live nodes.

triggered is a boolean attribute. It is set to true when the node did not respond to a message and is considered to be in a state between dead and alive. Another way of specifying this would be a bounded counter.

subscribed_groups is a set of group identifiers (256 bit IDs) that a node is known to be subscribed to in Scribe.

4.2 SLOSL Implemented Chord Topology

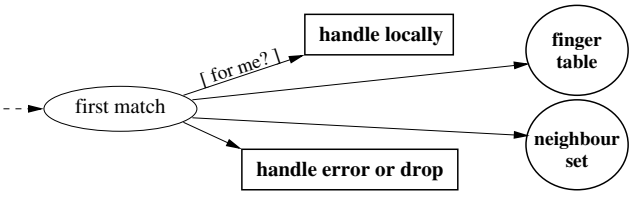
Chord [1] deploys two different views: the *finger table* defines the major characteristics of its topology and the *neighbour set* contains the predecessor and successor along the ring. SLOSL implements the finger table as follows.

```
1  CREATE VIEW chord_fingertable
2  AS SELECT node.id, node.ring_dist, chord_bucket=i
3  FROM node_db
4  WITH log_k = log(|𝒦|)
5  WHERE node.supports_chord = true AND node.alive = true
6  HAVING node.ring_dist in [2^i, 2^(i+1))
7  FOREACH i IN [0, log_k)
8  RANKED lowest(1, node.ring_dist)
```

The neighbour set implementation contains the node with the lowest node ID further along the ring (the successor) and the node with the highest node ID backwards on the ring (the predecessor). For resilience reasons, the view specified below stores a larger number of nodes, as encouraged by the original Chord paper.

```
1  CREATE VIEW circle_neighbours
2  AS SELECT node.id, side=sign
3  FROM node_db
4  WITH ncount=10, max_id=|𝒦| − 1
5  WHERE node.alive = true
6  HAVING abs(node.id − local.id) <= max_id / 2
7         AND sign*(node.id − local.id) < 0
8      OR abs(node.id − local.id) > max_id / 2
9         AND sign*(node.id − local.id) > 0
10 FOREACH sign IN {−1,1}
11 RANKED lowest(ncount, node.ring_dist)
```

4.3 Chord Routing Through SLOSL Views



The routing decision graph for Chord is shown in the figure. Messages are pushed through the tree from the left. The outcome is either some kind of local

handling (boxes) or message forwarding through views (circles). At the *first match* of a target amongst the sequence of child branches (top-to-bottom order), the execution is terminated and the message is handed to the target. One of the transitions has a predicate associated with it. Its implementation is provided by an external component.

If the target is a view, the message is forwarded via its equivalence classes. The receiver node is locally instantiated as a possibly incomplete node, for which the HAVING–FOREACH and RANKED clauses are evaluated. This yields a number of nodes to which the message is then broadcasted. If the view evaluation fails, the tree traversal continues.

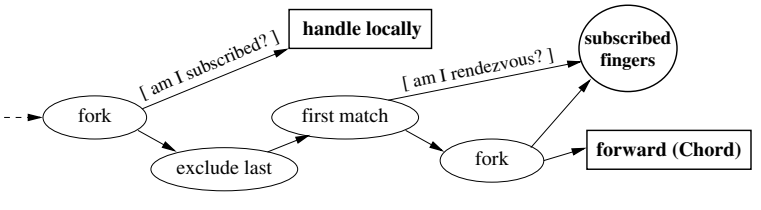
4.4 SLOSL Implemented Scribe on Chord

The Scribe [6] multicast scheme requires an additional view for routing publications. It forwards them towards the rendezvous node of the respective group and at each hop along the path broadcasts it to all subscribed children. Forwarding towards the rendezvous simply deploys Chord routing, but the broadcast requires an additional view on top of the Chord topology that selects only subscribed neighbours.

The selection is based on the active subscriptions of nodes in the finger table. All locally active subscriptions are given by the well defined set containing the subscriptions of the local node or its children towards the parents. It includes the group IDs for which the local node is the rendezvous node and for which children or the local node are subscribed. The Scribe implementation requires a view for each of the locally active groups as follows.

```
1 CREATE VIEW scribe_subscribed_children
2 AS SELECT node.id
3 FROM chord_fingertable
4 WITH sub
5 WHERE sub in node.subscribed_groups
```

4.5 Multicast Routing Through SLOSL Views



The routing decision graph for Scribe publications on Chord is shown in the figure. Along the path, the execution is forked into different branches, which treat the message independently. The "exclude last" property is a common helper that prevents the last hop of a received message from appearing amongst the selection of next hops further down the tree.

4.6 Message Specifications in HIMDEL

Instead of presenting the XML representation of HIMDEL, we will stick to a shorter form here. The following messages are used in our implementation. The name in brackets is used for accessing fields and structure of messages. Note that the programmatic interface of messages also defines a *last_hop* field (if it is known) and a *next_hop* field (if it was determined by a router).

- Header [chord]
  - Container [ids]
    - id [sender]
    - id [receiver]
  - Message [chord_joined]
  - Message [chord_find_successor]
  - Message [chord_find_predecessor]
  - Message [chord_notify]
  - Message [chord_update_fingertable]
    - View-Data [finger_table_bucket] → chord_fingertable/bucket
  - Header [scribe]
    - Message [scribe_create_rendezvous]
    - Message [scribe_join]
    - Message [scribe_leave]
    - Message [scribe_publish]
      - Data [event]

4.7 Event Handling in EDSL

This is the main part where Gridkit integrates with the code generation process. Gridkit components implement the controllers that are connected by the EDSL graph specification. Due to space constraints, we cannot present the entire protocol graph of Scribe on Chord that implements the IControl component. We must therefore content ourselves with an example event processing cycle that shows how the system responds to events in a non-trivial way. We chose the leave process in Scribe for this purpose, i.e. the propagation of unsubscriptions from groups. The complete process is presented in figure 7. The vertices are EDSL states (Gridkit implemented controllers), solid lines represent EDSL transitions and dashed lines represent programmatic actions of controllers, that are not covered by EDSL.

There are three cases in Scribe that trigger a leave. The first one is the local leave that unsubscribes the local node. The second one is the explicit leave where a neighbour sends an unsubscribe message for a group. The third one is the implicit leave where the local node loses the connection to a subscribed neighbour. The first two cases are mainly identical in handling, the implicit one requires additional logic to decide that a leave must be triggered.

In this case, any component that sends messages and expects some kind of acknowledgement for them has two transitions coming out of it: one for receiving

the ACK and one for a timeout. Only one of them will ever be triggered, so whatever comes first will determine the further execution path.

If the ACK is received first, all is fine. If, however, the timeout comes first, it triggers the destination state of the timeout, which is a generic ACK handler controller. This controller looks up the *next_hop* in the respective message and switches its *triggered* attribute through the database API. If it becomes true, the controller triggers the node by sending it a ping and then terminates. The same controller will handle the timeout of this ping. If, however, the attribute was true already and becomes false now, the controller sets the *alive* attribute of that node to false. These changes will trigger events from the database. In our example, the update event of the *triggered* attribute is not used, but all SLOSL views must be updated for the *alive* event.

For simplicity, we will assume that the node was only contained in the finger table view and has open subscriptions in the Scribe subscriptions view. Containment can be decided in different ways depending on the database and view implementations. If materialised views are used (which may be the simplest implementation anyway), it is easy to check if a node is visible in a view. Otherwise, the view has to be evaluated for the node, once with the original attributes and once with the modified attributes, to determine a change. If the node update did not impact the view content, no view update is needed and no events are generated. Note that the SLOSL statements in an OverML specification provide all semantics necessary to determine efficient evaluation plans at compilation time.

In our example, we assumed that the node was contained in the finger table. It will therefore disappear from the view after the *alive* update. This triggers the event that the node left the finger table view. The same will happen for the scribe_subscribed_children views that inherit from the finger table. Our current Chord implementation can ignore these events, as its routers only use the consistently updated views and do not require any notifications about updates. Similarly, Scribe can ignore them as long as there are nodes left in all subscription views. Therefore, in the simplest case, event handling ends just here.

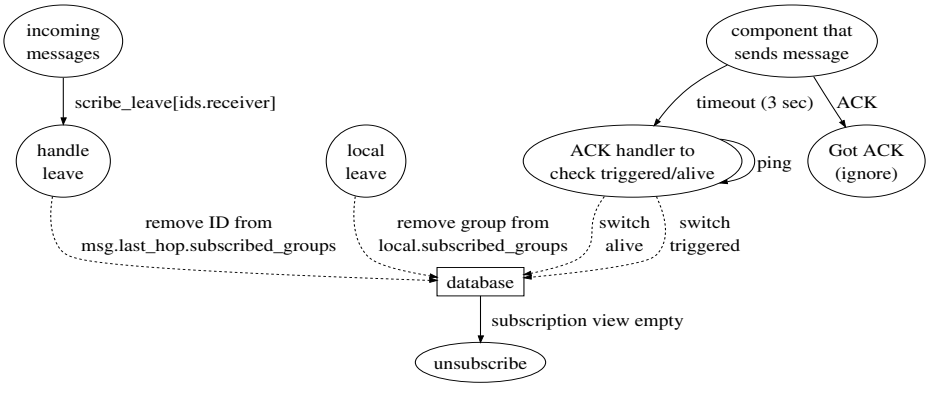


Fig. 7. The event processing cycle for explicit and implicit leaves in Scribe

In the case where the failed node was the only subscriber for a group, however, Scribe has to unsubscribe from that group. This is done through the *view empty* event that is triggered whenever a view update leaves the view empty. Note that the same event is triggered when a controller receives an explicit unsubscription from the last subscribed neighbour in a group and deletes the group ID from its *subscribed_groups* attribute. This will delete the last node from the subscription view of that group and thus trigger the event.

The *view empty* event of subscription views is therefore connected to an *unsubscribe* controller in the Scribe implementation. For each event, it sends out an unsubscribe message towards the rendezvous node of the respective group by using the underlying chord router.

4.8 From Specification to Deployment

Once the major characteristics of the overlay stack are defined, we can make the system runnable step by step. We start by running tests within the workbench. The SLOSL visualiser in figure 5 allows the developer to play with simple scenarios. Based on the evaluation of SLOSL views against a global database, it supports per-node restrictions on the local view to simulate global differences and inconsistencies in the local state of each node. This helps in finding topological problems in the specification before starting to work on the controllers.

Overlay specifications are then used to generate OpenCOM components that fit the Overlay framework. Since the router components are generated completely based on SLOSL and EDGAR, the main portion that remains to be implemented is the logic behind the IControl interface. It is internally structured by the EDSL graph. At the beginning of the coding step, it is helpful to use dummies for controllers that are not yet implemented. Their interfaces are defined by their EDSL interaction, so this is simple to do in code generators. Also, as figure 7 suggests, controllers generally tend to be very simple and small, which allows for pre-implemented components and quick-and-dirty stub implementations to get the system working. From the EDSL graph, it is immediately clear which components are required to make a specific portion of the protocols work.

The following steps obviously depend on the available tool support. Environments for testing, debugging and simulating overlay implementations are not yet available for the OverGrid architecture. Our main achievements in this area, however, are the rich semantics that become available to debuggers and the possibility to write these tools for generic OverML models and the generic Gridkit/OpenCOM component architecture. Once available, these developer tools will not require any adaptation to the specific overlay implementations. Current overlay simulators are very much focused on specific requirements of the specific overlays they were written for. OverGrid implemented overlays, on the other hand, will seamlessly move between different OverGrid compatible simulators, visualisers, debuggers and deployment environments without major redesign or rewrites. Throughout the testing phase, the designer will be free to go back to the design phase, modify the models and regenerate the overlay implementation.

5 Related Work

There are middlewares and application toolkits that provide principled support for p2p application development. *JXTA* (`http://www.jxta.org`) is a framework where p2p applications are developed atop a resource search abstraction; this supports grouping and contacting nodes. This abstraction can be implemented using a number of overlay topologies. Hence, like Gridkit the developer must follow the implementation specification.

Furthermore, like OverML there are high-level languages and tools for reducing the development effort of overlay networks. *iOverlay* [13] provides a message switch abstraction for the design of the local routing algorithm. The neighbours of a node are instantiated as local I/O queues between which the user provided implementation switches messages. This simplifies the design of overlay algorithms by hiding the lower networking levels. However, unlike OverML there is no further support for topology rules, maintenance or adaptation.

Macedon [14] is a state machine compiler for overlay protocol design and forms the most interesting approach so far. Event-driven state machines (EDSMs) have been used over decades for protocol design and specification. Macedon extends this approach to an overlay specific, C++ based language from which it generates source code for overlay maintenance and routing. In a number of different proof-of-concept overlay implementations, this was shown to be very useful for implementing and testing algorithms for routing and maintenance.

However, overlays must automatically configure themselves and adapt to a changes; it is not just a matter of routing protocol design. Each node in an overlay needs to take local decisions. The sum of these decisions is the distributed algorithm that maintains the overlay. iOverlay bases these decisions on the currently available connections. It does not provide means for selecting the "right" connections or categorising them, neither does it support ranking connection candidates for adaptation and fall-back mechanisms. Similarly, Macedon does not support candidate nodes or adaptability of topologies. Modelling adaptivity in state machines is likely to be complex and lead to state explosion. Neither framework supports a fine-grained componentisation of the implementation. Consequently, in these incompatible and language dependent frameworks, the designer models local decisions in framework specific source code.

In the P2/Overlog project [15], applications use a declarative logic language to specify their requirements of the overlay network. This is combined with a data flow approach, as opposed to a finite state machine approach, to maintain the overlay at runtime. Like Macedon, this simplifies the development process of overlays in specific cases. Apart from the implementation of overlay protocols, however, P2 offers no support for overlay applications or their deployment.

In this section, we have investigated solutions from both the middleware and overlay development domains. However, as far as we are aware, there has been no work similar to OverGrid that has investigated the combining of the two approaches; whereby the best of both worlds allows the creation of highly tailorable overlays to better support actual deployed applications and middleware services.

6 Conclusions

OverGrid is not yet a ready-to-run overlay suite. It is the main purpose of this paper to outline the possibilities of an integrated specification-to-deployment approach to overlay implementation. The combination of OverML models with the Gridkit infrastructure shows extremely promising results for the simplified design, automated generation and adaptive deployment of overlay applications.

A major advantage of OverGrid is the clean separation into forward, state and control components, that is otherwise hard to achieve by hand. Even more, this separation is done by OverML in a completely platform-independent way. Radical moves to new deployment environments become possible through the reduced dependency on platform-specific source code of overlay implementations. A generic, platform-specific OverGrid implementation can host a broad variety of overlays and topologies. Configuration and adaptation is moved into code generators and deployment- or run-time component architectures.

To back our promises for OverGrid, we have presented specifications of complex overlays. The simplicity of the abstract models that shines from the examples translates into a better understanding of the system and faster development cycles through early edit-compile-run feedback. However, a problem of OverGrid in this early project phase is the lack of tool support. Besides the SLOSL Overlay Workbench prototype there are no simulators or debugging environments that could support developers in their work. However, we expect that these tools can be integrated into the Gridkit/OpenCOM component infrastructure; especially debuggers will benefit from the semantically rich execution environment.

To our knowledge, OverGrid is the first in the area of overlay middleware and frameworks, to tackle the entire design process from the specification in platform-independent models to the adaptable deployment in changing environments.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. of the 2001 ACM SIGCOMM Conference, San Diego, CA, USA (2001)
2. Loguinov, D., Kumar, A., Rai, V., Ganesh, S.: Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. [16]
3. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making gnutella-like p2p systems scalable. [16]
4. Dabek, F., Zhao, B., Druschel, P., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03), Berkeley, CA, USA (2003)
5. Grace, P., Coulson, G., Blair, G., Porter, B.: Deep middleware for the divergent grid. In: Proc. of the Int. Middleware Conference (Middleware2005), Grenoble, France (2005)
6. Rowstron, A., Kermarrec, A.M., Castro, M., Druschel, P.: SCRIBE: The design of a large-scale event notification infrastructure. In: Proc. of the 3rd Int. Workshop on Networked Group Communications (NGC'01), London, UK (2001)

7. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A Component Model for Building Systems Software. In: Proc. of the IASTED Conference on Software Engineering and Applications (SEA'04), Cambridge, MA, USA (2004)
8. Mathy, L., Canonico, R., Hutchinson, D.: An Overlay Tree Building Control Protocol. In: Proc. of the 3rd Int. COST264 Workshop on Networked Group Communication, London, UK (2001) 76–87
9. Ganesh, A.J., Kermarrec, A.M., Massoulie, L.: SCAMP: Peer-to-peer lightweight membership service for large-scale group communication. In: Proc. of the 3rd Int. Workshop on Networked Group Communication, London, UK (2001) 44–55
10. van Renesse, R., Minsky, Y., Hayden, M.: A Gossip-Based Failure Detection Service. In: Proc. of the 1st IFIP International Conference on Middleware, Lake District, UK (1998) 55–70
11. Behnel, S., Buchmann, A.: Models and Languages for Overlay Networks. In: Proc. of the 3rd Int. VLDB Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2005), Trondheim, Norway (2005)
12. Porter, B., Coulson, G.: Intelligent Dependability Services for Overlay Networks. In: Proc. of the 6th IFIP WG International Conference on Distributed Applications and Interoperable Systems, Bologna, Italy (2006)
13. Li, B., Guo, J., Wan, M.: iOverlay: A lightweight middleware infrastructure for overlay application implementations. In: Proc. of the Int. Middleware Conference (Middleware2004), Toronto, Canada (2004)
14. Rodriguez, A., Killian, C., Bhat, S., Kostić, D., Vahdat, A.: MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proc. of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI2004), San Francisco, CA, USA (2004)
15. Loo, B.T., Condie, T., Hellerstein, J.M., Maniatis, P., Roscoe, T., Stoica, I.: Implementing declarative overlays. SIGOPS Operating Systems Review (2005)
16. The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM). (2003)

Distributed Lookup in Structured Peer-to-Peer Ad-Hoc Networks

Raphaël Kummer, Peter Kropf, and Pascal Felber

Computer Science Department, University of Neuchâtel,
Emile-Argand 11, CP 158,CH-2009 Neuchâtel, Switzerland
{raphael.kummer, peter.kropf, pascal.felber}@unine.ch
http://www.unine.ch/iiun

Abstract. Various peer-to-peer (P2P) architectures for ad-hoc networks have been proposed over the last few years. Most of them are unstructured and use some form of flooding to locate content, because the physical constraints of the underlying network make the construction of arbitrary application-layer overlays impractical.

In this paper, we study the problem of applying distributed hash tables (DHT) to ad-hoc networks. Our approach to efficiently lookup content in such networks exploits physical proximity of peers when establishing and maintaining the DHT based routing tables. The efficiency of our method is demonstrated by simulation of large networks.

1 Introduction

Peer-to-peer (P2P) systems, in which peer nodes form a cooperative network and share their resources (storage, CPU, bandwidth), have attracted a lot of interest lately. Roughly speaking, P2P networks can be classified as either structured or unstructured. Unstructured P2P networks (e.g., the Gnutella [1] and KaZaA [2] file sharing systems) have no precise control over the file placement and generally use "flooding" search protocols, which typically generate a large number of query messages. In contrast, structured P2P networks (e.g., Chord [3], CAN [4], Pastry [5], P-Grid [6]) use specialized placement algorithms to assign responsibility for each file to specific peers, as well as "directed search" protocols to efficiently locate files.

Structured P2P networks using deterministic algorithms are based on the distributed hash table (DHT) paradigm. Each data item is identified by a key (data identifier). The DHT maps keys to the nodes of an overlay network and provides facilities for locating the current peer node responsible for a given key. DHT designs differ mostly by the way they maintain the network and perform lookups: there is a fundamental trade-off between the degree of the network (the number of neighbors per node, i.e., the size of the routing tables) and its diameter (the average number of hops required per lookup) [7].

Directed search protocols are particularly efficient, because they accurately route queries toward the peers responsible for a given file. They require few communication stages generating only little traffic, and do not produce false

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1541–1554, 2006.
© Springer-Verlag Berlin Heidelberg 2006

negatives (i.e., the search fails only if there is no matching file in the system). Therefore, they are particularly attractive for systems with limited resources, where the cost of sending a message is non-negligible (e.g., in terms of energy) and flooding is considered prohibitive.

Ad-hoc networks could greatly benefit from the efficiency of a directed search protocol to locate content. They do, however, suffer from limitations that make the maintenance of a DHT impractical. Most notably, a node in an ad-hoc network can communicate directly only with its physical neighbors, i.e., with the nodes that are within its (radio) communication range, and communication between remote nodes must traverse multiple intermediate nodes. DHTs assume that the peers form a fully connected network and that the neighborhood of a node can be chosen according to its location in a "logical" overlay, rather than to "physical" considerations. A simple mapping of a DHT design to an ad-hoc network would be unrealistic, as every node would need to keep track and communicate with a set of "logical" neighbors many steps away in the physical network. There is a fundamental gap between the logical proximity defined by the DHT overlay and the physical proximity imposed by the underlying ad-hoc infrastructure.

In this paper, we study the feasibility of DHT lookup in ad-hoc networks. We propose a DHT design that combines a minimalist logical overlay structure together with adaptive routing mechanisms to quickly locate content. Nodes are organized in a logical ring, similarly to Chord [3] or Pastry [5], but we do not use logical long-range neighbors whose maintenance costs would be prohibitive. Instead, we rely on the physical neighbors of the nodes traversed by requests routed through the physical network to quickly converge toward the destination in the logical overlay. We propose additional extensions, in which we consider an extra level of visibility in the physical neighborhood (neighbors of neighbors) and maintain a history of previous requests to dynamically identify and exploit possible shortcuts.

We have extensively studied our algorithm by the means of simulations in static deployment scenarios. We did not take into account mobility nor churn as we are mostly interested in evaluating the feasibility of DHT lookup in ad-hoc networks. Results demonstrate that our approach is indeed efficient and scales well to large peer populations. It represents a promising alternative to existing solutions based on unstructured P2P networks.

The remainder of this paper is organized as follows. Section 2 discusses related work. We detail our algorithm in Section 3 and present results from experimental evaluation in Section 4. Finally, Section 5 concludes.

2 Related Work

Peer-to-peer overlays have emerged from file sharing applications on top of the Internet, leveraging from its routing infrastructure and the intrinsic peer-to-peer property of the IP protocol. As already discussed in Section 1, the situation is different in the case an ad-hoc networks. In particular, the DHT paradigm with

its notion of regular topology (often a ring) and the shortcuts (fingers) introduced at the overlay layer, makes a direct mapping to ad-hoc networks difficult. Various approaches are known from literature to achieve such mappings.

While GRACE (Global Replication And Consistency) [8] has not been specifically designed with ad-hoc networks in mind, it enables for mobile collaboration by combining DHT properties with a layered architecture. GRACE supports mobility in wide-area networks. The different layers or consistency levels are interconnected through "consistency neighbors" that are logically close to each other. Requests are routed along these neighbors. The lookup algorithm of the system is based on Pastry [5]. This approach still relies on the standard Internet infrastructure.

Pucha et al. [9] implement Pastry on top of the Manet routing protocol DSR (Dynamic Source Routing)[10]. Three modifications are proposed as compared to the implementation on the Internet: (1) the node join procedure is modified by expanding the ring search for locating distinguished bootstrap nodes in charge of arrivals; (2) the Pastry ping metric is replaced by a distance metric to reduce network load; and (3) the DSR protocol is modified to inquire about the proximity used in the adapted Pastry routing.

Ekta [11] and MADPastry [12,13] integrate the DHT paradigm with ad-hoc network routing. Both approaches introduce the necessary functions at the network routing layer. The principal idea of Ekta is to move the DHT protocol from the overlay level to the network layer of the Manet : a one-to-one mapping between IP-addresses and logical (DHT) node IDs is applied. MADPastry is built on top of the AODV protocol (Ad-hoc on-demand vector routing) [14]. This protocol aims to avoid full broadcasts as much as possible because these are costly in ad-hoc networks when targeted to the entire network. MADPastry creates clusters composed of physically close nodes sharing, at the same time, a common overlay prefix. The nodes in a cluster are thus physically and logically close. Routing is then based on the logical overlay node IDs.

All the approaches above suffer from the size of the routing tables and the complexity of setting up and managing connections with all the nodes that they contain. In contrast, in our design, we only need to keep track of two remote nodes, as explained in the next section.

Cramer et al. [15] propose the Chord based Proximity Neighbor Selection strategy (PNS-CHORD). Here, the nodes are connected to their logical successors on the ring and through logical shortcuts to further nodes as usual in Chord. These logical long-range neighbors are chosen according to physical proximity in the ad-hoc network and are either one or two steps away.

Because the construction of the routing tables is based on physical proximity, it may happen that the logical path pursued by a request traverses the same node multiple times. This may only be prevented when the nodes keep track of the requests that pass by and subsequently adjust their routing tables in case the same request passes twice, but on different logical shortcuts. Our approach takes into account of this situation and prevents thus such loops through the according learning process.

Cell Hash Routing (CHR) [16] is a specialized ad-hoc DHT. CHR uses position information to construct a DHT of clusters instead of organizing individual nodes in the overlay. This approach groups nodes according to their physical location. The routing between the clusters is done by position-based routing with the GPSR [17] routing algorithm. A major limitation of this approach is that nodes are not individually addressable, but only via the clusters.

Finally, we are aware of a parallel effort to our's by Rowstron *et al.* at Microsoft Research (Cambridge) that similarly aims to combine ad-hoc routing and logical DHT-like addressing (Virtual Ring Routing, VRR). Based on the (yet unpublished) information that we have obtained directly from the authors, their approach differs from ours in several ways. First, they build and proactively maintain bidirectional routes between nodes. In contrast, our algorithm always tries to find the best route at each node and sends the request along this path. It is therefore possible to take into account locally profitable situations. Second, VRR also maintains existing routes in a proactive way. Our solution does not maintain existing routes, but previous routing decisions are kept in a cache and can thus be reactivated if appropriate. This results in more efficient routes to emerge at each step along a path and allows to faster reacting on topological changes. Finally, VRR nodes use information of the physical paths traversing the nodes for improving the routing. Our experiments, however, have shown that such a caching policy results in small improvement, only. Indeed, the cached entries are effective only when a request's path goes through the node having registered them. In our approach, the nodes also capture communications of the nodes in the physical neighborhood. The information acquired this way implicitly includes routing decisions and thus allows avoiding paths to traverse the same physical area twice. It has to be noted that this information is obtained without any further communication overhead, since it may be captured for free due to the characteristics of wireless networks, i.e. radio transmission.

3 The Ad-Hoc Lookup Algorithm

A DHT system maps keys to nodes in a peer-to-peer infrastructure. Any node can use the DHT substrate to determine the current live node that is responsible for a given key. Every node and key has a specific position in a logical identifier space, and the mapping between keys and nodes is determined according to a proximity metric in the logical space. DHT nodes typically rely upon "long-range neighbors" to quickly route messages to remote locations in the identifier space. For instance, Chord [3] organizes the nodes in a logical ring. Each node is connected to its closest neighbors (successor and predecessor) in the identifier space. These connexions are necessary to guarantee successful routing: it is always possible to traverse the whole ring by following successor links, albeit at a very high cost. Each node additionally has a number of long-range neighbors, called fingers, that refer to nodes located at exponentially increasing distance in the logical space. Using these links, a node can quickly reach a remote location: the expected path length (number of hops) of a lookup is in $O(logN)$, where N

is the number of nodes in the system. Long-range neighbors are not necessary for the safety and reliability of the system (successor links are sufficient), but they provide liveness properties that allow for efficient lookup.

In an ad-hoc network, nodes can directly communicate only with their physical neighbors, i.e., nodes that are within their communication range. Messages sent to remote nodes require multiple steps[1]. In the following, we assume that (1) the ad-hoc network forms a connected graph and (2) there is an underlying ad-hoc routing protocol that allows us to route messages between any two nodes. We do not make any assumption on the ad-hoc routing protocol, except that it always succeeds and that it is possible to take actions at intermediate nodes on the routing path (e.g., to change the course of a message). One may note, that when a DHT overlay is implemented over a classical Internet structure, the IP routing also takes care of delivering messages over multiple physical steps for one logical link (hop).

Mapping a DHT design such as Chord, Pastry, or P-Grid on an ad-hoc network would place nodes that are physically close at arbitrary locations in the logical space. Therefore, successor, predecessor, and long-range links would refer to remote nodes that can only be reached by multiple physical steps. Besides being inefficient in terms of physical path length, such an approach is impractical because each node would have to maintain accurate routing information for $O(logN)$ long-range neighbors. This can become a major problem when the size of the network grows or if nodes often become unreachable (due to disconnections or failures). Moreover, mobility and churn, though not considered in this study, make a direct mapping of a DHT design to ad-hoc networks unrealistic.

The intuition underlying our design is to only maintain a minimalist logical overlay without long-range neighbors, that is responsible for safety only. For efficient lookup, we rather rely on the physical neighborhoods of the nodes traversed during lookup to spontaneously find long-range links in the logical space; therefore, we have a probabilistic form of liveness property because long-range links may be encountered on a random basis. Our conjecture is that lookup requests can be routed more efficiently and with much lower management costs than when deterministically maintaining $O(logN)$ long-range neighbors.

Similarly to Chord or Pastry, we organize the nodes in a logical ring, as shown in Figure 1. The node responsible for a given key is the closest one in the identifier space. Each peer n only needs to keep track at all times of its successor $succ(n)$ and predecessor $pred(n)$ on the ring, which imposes limited management overhead, only. Additional robustness can obviously be gained by considering several successors and predecessors, but we do not consider failures in this paper.

Each node is assigned a random identifier in the logical space, e.g., by hashing the node's IP address using a cryptographic hash function such as SHA-1. Therefore, the physical neighbors of a given node are expected to be randomly distributed in the logical space (see Figure 1). This diversity property is important for the efficiency of our lookup algorithm.

[1] In this paper, we denote by *hop* a logical link on the DHT overlay, and by *step* a physical link of the ad-hoc network.

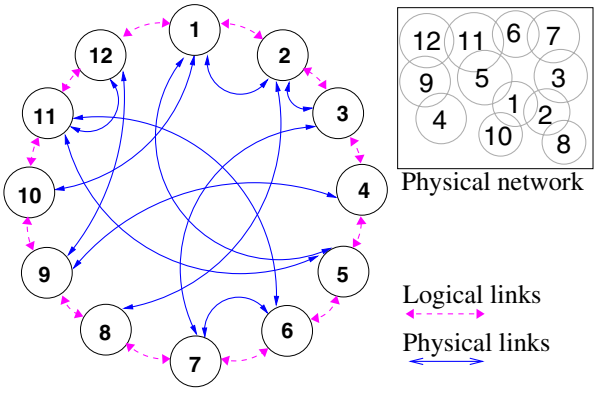


Fig. 1. Illustration of the DHT model for ad-hoc network

3.1 Basic Lookup

We first describe the basic version of our algorithm, before discussing improvements to enhance lookup performance. The pseudo-code is shown in Algorithm 1.

In our approach, DHT lookup is closely integrated with the routing of messages through the ad-hoc network: when a message is sent to a remote node multiple steps away, it might diverge from its path at some intermediate if it finds a shorter way to its final destination.

Every lookup message contains the identifier of the key being searched (denoted by k in the pseudo-code). In addition, when a lookup message is routed to a remote node across the physical network, it contains the identifier of its destination (n_d in the pseudo-code).

Upon receiving a lookup message, a node n_i proceeds as follows. First, n_i searches among its physical and logical neighbors, as well as itself and n_d, which node is closest to the key (line 2). If n_i is closest, then it is responsible for the key and it replies to the originator of the request (line 4).

If the request is on its way to a remote node n_d closer to the key than any of n_i's neighbors, we simply forward the message to the next node on the multi-step path to n_d (lines 6–7).

Otherwise, the node n_j closest to the key is part of n_i's physical or logical neighborhood and will become the new destination. There are two cases to consider: if n_j is a logical neighbor, we issue a multi-step request toward n_j (lines 9–10); otherwise, we directly send the request to that node (line 12). In both cases, we bypass the regular routing process and take a shortcut toward the destination key.

3.2 Convergence and Termination

We shall now study the convergence and termination of our algorithm, under the assumption that the system is in a stable state with no node joining or leaving.

Algorithm 1. Basic DHT lookup algorithm at node n_i for key k

$\delta(k_1, k_2)$: distance between keys k_1 and k_2
$id(n)$: logical identifier (key) of node n
V_i: physical neighbors of n_i
$L_i = \{pred(n_i), succ(n_i)\}$: logical neighbors of n_i

```
1: procedure LOOKUP(k, n_d)
   k : looked up key
   n_d : next logical hop
2:     n_j = arg min_{n ∈ V_i ∪ L_i ∪ {n_i, n_d}} δ(id(n), k)
3:     if n_j = n_i then                          { We are responsible for k }
4:         return n_i
5:     else if n_j = n_d then                     { Continue to n_d }
6:         n_k ← next step on physical path to n_d;
7:         send LOOKUP(k, n_d) to n_k;
8:     else if n_j ∈ L_i then                     { Go to logical neighbor }
9:         n_k ← next step on physical path to n_j;
10:        send LOOKUP(k, n_j) to n_k;
11:    else                                       { Go to physical neighbor }
12:        send LOOKUP(k, n_j) to n_j;
13:    end if
14: end procedure
```

Let us consider node n_i receiving a lookup request for key k, possibly on its way to a remote node n_d. To guarantee convergence, our algorithm tries to always reduce the logical distance to k. To that end, it looks among its neighbors, n_d, and itself, the node n_j closest to the key.

First observe that, if n_i is closest to the key, then it *must* be responsible for the key. Otherwise, either its predecessor or successor would be closer.

There are four cases to consider:

- If n_i is closest to the key, then it is responsible for the key and the lookup ends.
- If a physical neighbor is closest to the key, then it directly receives the request.
- If a logical neighbor is closest to the key, then it becomes the new target n_d of the request.
- If n_d is closest to the key, then the algorithm does not change the destination of the request.

In the first three cases, the distance to the key decreases, either because we reach the node responsible for the key, or because we send the request to a node closer to the key. In the last case, we do not decrease the distance to the key but we proceed toward n_d and, if the request reaches n_j, one of the first three cases will apply. The assumptions on the underlying ad-hoc routing protocol guarantees that a request sent to a node will eventually reach that node; thus,

the last case cannot apply forever. It follows that our algorithm converges and terminates in a finite number of steps.

3.3 Increasing Visibility

The visibility of a node, in terms of physical neighborhood, is limited by the communication range. As nodes can communicate directly with their physical neighbors, a simple improvement is to have nodes exchange information about their neighborhood. That way, visibility gets extended to the "neighbors of neighbors" (NoN).

This extension has a limited cost in terms of message overhead, because NoN lists are sent only upon change and require a single broadcast. Yet, it has the potential to significantly improve lookup efficiency because it increases the probability of finding a suitable long-range link while routing requests across the ad-hoc network.

3.4 Exploiting Request History

Ad-hoc networks are characterized by the fact that every node must participate in request routing and forwarding. Therefore, a node sees traffic for which it is not the target. We can thus exploit information about past requests to acquire knowledge of remote nodes and improve lookup efficiency.

Each node maintains a cache that stores, for each observed request, the searched key k and the destination of the message n_d. Thereafter, the cache entry (k, n_d) can act as a long-range neighbor: when a node receives a request for a key closer to k than the node that would be selected by Algorithm 1, the request is redirected toward n_d instead.

This extension does not require any extra message to be sent. Further, as all the physical neighbors of a node can listen to its messages (radio communications are broadcast), information about past requests can be gathered passively. The cache uses a least-recently used replacement policy and we have limited its size to 256 entries in our implementation.

4 Evaluation

4.1 Experimental Setup

An experimental system has been implemented to analyze the efficiency of the ad-hoc lookup protocol and to demonstrate the feasibility of exploiting the known performance of the DHT lookup paradigm when applied to ad-hoc networks.

The simulated system is an overlay network of a fixed number of nodes arranged in a rectangular area as physical space. Recall that our study neither targets to investigate mobility nor churn. Therefore, the experimental setup considers only static scenarios.

The nodes are randomly distributed in the physical space and are randomly assigned their identifiers in the DHT space. To take into account the limited

radio range of a node in the ad-hoc network, two nodes are only considered to be connected if their separating distance (relative to the dimensions of the rectangular area) is smaller than a maximum range limit. A density parameter allows to control the average number of connections of a node in the system.

To forward a request to the next hop in the logical space, appropriate routing in the physical space needs to be simulated. The implemented simulation routing strategy is similar to ad-hoc routing algorithms such as LAR [18]. The positions of all nodes are known to the simulator, which allows to refrain from implementing routing tables. When a node receives a request to be forwarded to some destination node at the logical level, it chooses as the next step the node in its physical neighborhood that is the closest to the position of the (logical) destination node. The network generation described above allows for degenerated situations where this routing process does converge to the destination (e.g., when reaching a "dead-end"). To cope with this situation and avoid the complexity of simulating a complete ad-hoc routing protocol, a bypass method is used in which the request is directly delivered to the destination node. For such a communication, the current average number of steps for the corresponding logical link is accounted for in the statistical analysis.

The experiments have been conducted for different network sizes and the different variations of the ad-hoc lookup algorithm.

A. Simulation Setup.

Network size : 1,000, 10,000, 100,000.

Connectivity : the average number of physical connections of a node (network density) varies between 13 and 16.

Lookup requests : for each experiment, the paths of at least 2,000 randomly generated requests are statistically evaluated.

Steady state : to evaluate the variants of the ad-hoc lookup algorithm using the caching mechanism, the simulation runs a *warm-up* phase during 2,000 (2K), 50,000 (50K), or 100,000 (100K) requests in order to reach a steady state before the statistical information is collected for the analysis.

B. Simulation Experiments.

Basic : the basic DHT lookup algorithm

Neighbors-of-neighbors (NoN) : the basic algorithm evaluating the physical neighbors and their physical neighbors to choose the next step.

Cache (C) : the NoN algorithm using a cache memorizing previous forwarding choices.

Warm-up (Wup) : the simulation system executes a warm-up phase to reach a steady state prior to issuing the analyzed requests.

The data used for the analysis is collected by the requests themselves along their itinerary through the ad-hoc network. This allows us to also analyze the paths through the physical network for individual requests and hence compute average values.

4.2 Discussion and Results

Each request is evaluated at each node independently of whether the node is an intermediate node of the logical path or an intermediate node for reaching a node on the logical path. In either case, the algorithm tries to forward the request to the node approaching the most the destination. Logical paths may thus be altered by the algorithm. Table 1 shows the percentages of altered paths (or shortcuts taken) determined during the different simulation scenarios. The versions using the cache use shortcuts for more than 50% of the requests. For large networks, however, this percentage decreases. This is not surprising, because the density (average number of physical connections) is the same for all networks. In case of large networks, there are hence less possibilities for profitable shortcuts.

Table 1. Average use of logical shortcuts

Network size	Basic	NoN	C Wup-0	C Wup-2K	C Wup-50K	C Wup-100K
1,000	38%	37%	51%	54%	54%	54%
10,000	39%	41%	49%	56%	59%	59%
100,000	35%	37%	42%	46%	53%	N/A

One observes that the average values for networks of size 1,000 and 10,000 are roughly the same when applying the warm-up phase. This suggests that our lookup algorithm provides a stable lookup time in average after some running time (warm-up phase in the simulation), and up to some network size.

Figure 2 shows the average number of physical steps required to serve a request. The use of the NoN and cache with warm-up strategies reduces the number of physical steps by $\sim 20\%$ respectively $\sim 50\%$. The introduced logical shortcuts can thus be considered as very effective.

The average costs of one logical hop in terms of physical steps are roughly the same for all variants of the algorithm (see Table 2). The performance of the lookup algorithm increases thus when the number of logical hops for a request decreases. Figure 3 demonstrates that our ad-hoc lookup algorithm indeed reduces the number of logical hops to serve a request.

We observe that our algorithm always achieves the lookup in less than $\log N$ hops or even less than $\frac{1}{2}\log N$ hops. This clearly demonstrates the efficiency of the approach and confirms that the introduction of the cache further improves performance. Moreover, the various warm-up phases applied show that the ad-hoc lookup algorithm stabilizes over time: the longer it runs, the better it performs. There exists of course a limit where no further improvements will be possible when running the system longer and longer. To gain insight to the complex question how, whether and under which conditions (network topology) this limit remains a difficult graph theoretic question. Our empirical investigation suggests only that our algorithm exposes a stable behavior over time. Moreover, it must be remembered that the average search costs using the logical links only has a performance of the order of $O(N)$. In comparison, Figure 3 clearly shows the superiority of exploiting the properties of the physical links in the algorithm.

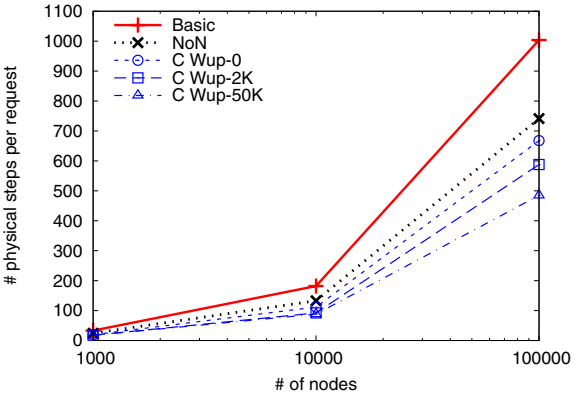


Fig. 2. Average number of physical steps to achieve a request

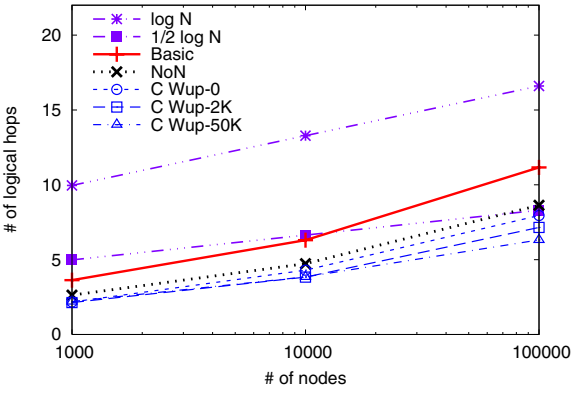


Fig. 3. Average number of logical hops per lookup

Table 2. Average cost of a logical hop in terms of physical steps

	Basic	NoN	C Wup-0	C Wup-50K
1,000	9.07	9.22	8.77	7.84
10,000	28.45	28.06	25.94	24.09
100,000	89.89	85.80	83.95	82.16

The ad-hoc lookup algorithm finds logical shortcuts through the selection of physical nodes closest to the destination and with the help of the information gained from previous requests kept in the cache. Intuitively this suggests that shortcuts determined at the beginning of a logical path should approach the destination the most. Indeed, the probability that some node in the physical

neighborhood is closer to the destination node than the next node on the logical path decreases with the number of logical hops traversed. In other words, the percentage of logical hops that terminate without a shortcut being taken increases when approaching the destination. This characteristic is demonstrated in Figure 4. The figure shows the percentage of terminated logical hops as a function of the distance from the node responsible for the searched key.

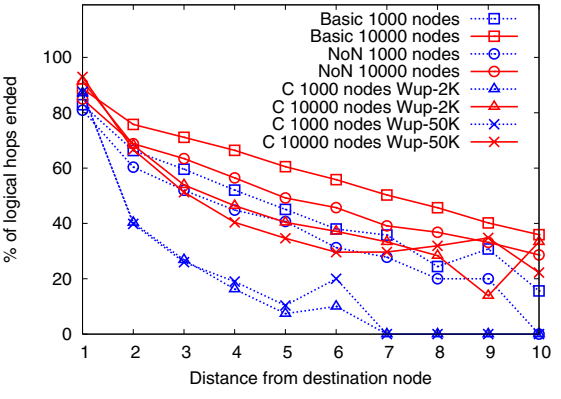


Fig. 4. Percentage of terminated logical hops

Finally, Figure 5 shows the path of the same request in the same network of 10,000 nodes but with different versions of our algorithm. Coordinates represent the geographical position of the nodes. Clearly, the best result is obtained when using the NoN version of the algorithm with caching.

5 Conclusion

Apart from very few known approaches, P2P systems for ad-hoc networks have essentially used unstructured designs and flooding-based search protocols. It is indeed challenging to build a structured overlay with a directed search protocol on top of an ad-hoc network, because nodes have only limited communication capabilities: they can exchange messages only with the nodes that are within their (radio) communication range.

In this paper, we have studied the feasibility of building a DHT for ad-hoc networks. The premise of such a design is to improve lookup performance and reduce the message overhead as compared to flooding-based protocols which yield linear or worse performance. In our approach, we only maintain a minimalist logical overlay structure but rely on the physical neighbors of the nodes traversed during ad-hoc routing to quickly reach the object being looked up. Our simulation results have demonstrated that a sub-linear lookup performance can be preserved when relying on an ad-hoc network infrastructure rather than on the

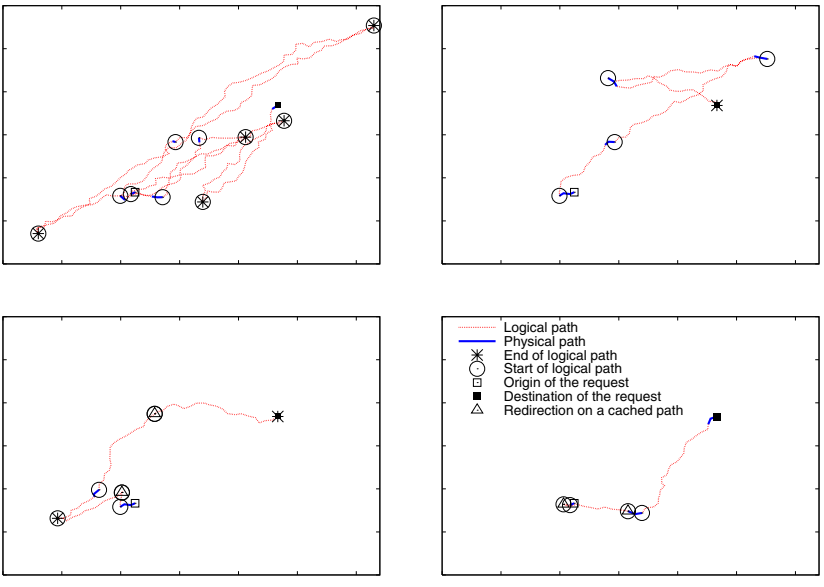


Fig. 5. Path of a request with Basic (top left), NoN (top right), C Wup-2K (bottom left), and C Wup-50K (bottom right) in a two-dimensional space

standard Internet. It must however be noted that, to the best of our knowledge, the observed sub-linear performance still has to be confirmed by an analytical study.

References

1. The Gnutella web site, www.gnutella.com (2006)
2. The KaZaA web site, www.kazaa.com (2006)
3. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, ACM Press (2001) 149–160
4. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, ACM Press (2001) 161–172
5. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware. (2001) 329–350
6. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: a self-organizing structured P2P system. SIGMOD Record **32**(3) (2003) 29–33

7. Xu, J., Kumar, A., Yu, X.: On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. IEEE Journal on Selected Areas in Communications **22**(1) (2004) 151–163
8. Bosneag, A.M., Brockmeyer, M.: GRACE: Enabling collaborations in wide-area distributed systems, 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05) (2005) 72–77
9. Pucha, H., Das, S.M., Hu, Y.C.: How to implement DHTs in mobile ad hoc networks?, the 10th ACM International Conference on Mobile Computing and Network (MobiCom) (2004)
10. Johnson, D.B., Maltz, D.A.: Dynamic source routing in ad hoc wireless networks. In: Mobile Computing. Volume 353., Kluwer Academic (1996)
11. Pucha, H., Das, S.M., Hu, Y.C.: EKTA: An efficient DHT substrate for distributed applications in mobile ad hoc networks., 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004) (2004) 163–173
12. Zahn, T., Schiller, J.H.: MADPastry: A DHT substrate for practicably sized MANETs., 5th Workshop on Applications and Services in Wireless Networks (ASWN2005) (2005)
13. Zahn, T., Schiller, J.H.: DHT-based unicast for mobile ad hoc networks., Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops(PerComW'06) (2006) 179–183
14. Perkins, C.E., Belding-Royer, E.M.: Ad-hoc on-demand distance vector routing. In: WMCSA. (1999) 90–100
15. Cramer, C., Fuhrmann, T.: Proximity neighbor selection for a dht in wireless multi-hop networks. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, Konstanz, Germany (2005) 3–10
16. Araùjo, F., Rodrigues, L., Kaiser, J., Liu, C., Mitidieri, C.: CHR: a distributed hash table for wireless ad hoc networks, 25th IEEE International Conference on Distributed Computing Systems Workshops (2005) 407–413
17. Karp, B., Kung, H.T.: GPSR: greedy perimeter stateless routing for wireless networks. In: MobiCom'00: Proceedings of the 6th annual international conference on Mobile computing and networking, New York, NY, USA, ACM Press (2000) 243–254
18. Ko, Y.B., Vaidya, N.H.: Location-aided routing (LAR) in mobile ad hoc networks. Wirel. Netw. **6**(4) (2000) 307–321

A Document-Centric Component Framework for Document Distributions

Ichiro Satoh

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ichiro@nii.ac.jp

Abstract. This paper presents a framework for building and managing compound documents in distributed systems. It enables an enriched document to be dynamically and nestedly composed of software components corresponding to various types of content, e.g., text, images, and windows. It permits the content of each component and program code to access the content inseparable inside the components so that the components can be viewed or modified without the need for any applications. It enables each component or document to migrate over a network under its own control by using mobile agent technology. Moreover, it introduces components as carriers or forwarders because it enables them to carry or transmit other components as first class objects to other locations. It offers several basic operations for network processing, e.g., forwarding, duplication, and synchronization. Since these operations are still document components, they can be dynamically deployed and customized at local or remote computers through GUI manipulations. It therefore allows an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. This paper describes the framework and its implementation, which currently uses Java as the implementation language as well as a component development language, and then illustrates several interesting applications that demonstrate its utility and flexibility.

1 Introduction

Document manipulation, such as editing, viewing, and distributing documents, is still playing a crucial role in modern information processing. Documents in distributed computing systems are always transmitted passively over a network by external systems, such as electronic mail systems and http servers. As a result, they cannot determine where, when, or how they should go next. However, there have been several applications whose network processing depends on the content of the documents that are transmitted over the network. For example, a workflow management system is required to distribute documents among employees according to the content of the documents and the company's decision-making path. As a result, end-users often want to define the network processing of documents for them to be able to accomplish their application-specific tasks. However, the customization and management of networking processing is too complex and difficult for end-users to achieve.

This paper proposes a compound document framework, called MobiDoc, as a solution to these problems. Like other existing compound document frameworks, it enables

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1555–1575, 2006.
© Springer-Verlag Berlin Heidelberg 2006

an enriched document to be composed of visual components, e.g., text and images. The framework has several unique features, which other frameworks do not have. For example, it introduces the notion of self-contained components in the sense that not only the content of each component but also its codes to view and edit the content are embedded in the component to solve various problems, including content rights management, in existing content-distribution. It also enables network protocols for documents to be implemented by a set of active documents. By using mobile agent technology, documents or components can define their own itineraries and migrate under their own control. Documents or components can transmit other documents or components as first-class objects to their destinations. The framework introduces the components for network processing as document-centric components, so that it allows an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. It provides components with a mechanism for sharing visual content on different computers.

This paper is organized as follows. We first describes the background and related work (Section 2) and outline our compound document framework (Section 3). We then present component runtime systems for executing and migrating document components (Section 4) and present our component model (Section 5). We describe its prototype implementation (Section 6) and illustrate several applications of the framework (Section 7). We conclude by providing a summary and discussing future issues (Section 8).

2 Background

Building systems from software components has already proven useful in the development of large-scale software [24]. Many frameworks for software components have been developed, e.g., COM [16] and JavaBeans [9]. These existing frameworks aim at defining the behaviors of distributed computing, i.e., server-side and client side processing, by combining software components. Therefore, these frameworks are suitable for professional developers but not end-users.

Several frameworks for compound document components have been developed, such as COM/OLE [3], OpenDoc [1], and CommonPoint [15]. They enable one document to be composed of various visible parts, such as text, image, and video, created by different applications. These frameworks assume that content is stored inside the component but it is accessed by external applications. Thus, when users receive a compound document, they cannot view or modify it, if its content needs the support of different applications, if they do not have all the applications. There have been XML-based technologies for office documents, e.g., OpenDocument formats, but these have inherited the problems of compound documents. This is because, when a computer receives XML-based documents, it may not have the viewer/editor programs for them. This becomes a serious limitation in distributing new data formats. Of these, the Bonobo framework for software components and compound documents is being developed by the GNOME project [7]. It provides several mechanisms for creating compound documents from a collection of components, but it is based on an underlying middleware and GUI widget, i.e., CORBA and GTK+, and does not support the distribution of components, including compound documents over a network.

The framework presented in this paper, therefore, has been designed independently of existing component frameworks for distributed computing or compound documents. This is because it permits document-centric components to migrate themselves over a network and process other components as first-class objects [5], e.g., migrating or saving them to other computers or on secondary disks. These features enable our components to have direct access to novel and powerful features that existing components do not have. End-users can also easily customize their network processing of documents through user-friendly manipulations to edit visual components, and they can control their own network processing according to their content. Nevertheless, it is open to existing component frameworks. In fact, it can use typical Java-based components, e.g., Java Beans and Applets, as its components.

Several (non-component-based) attempts have been made to support active documents, e.g., Active Mail [8] and HyperNews [14], but these have aimed at particular application-specific documents, such as electronic mail and newspapers, so that they have not supported varied and complex content. The fuseONE system [26] is composed of GUI-based control panels to control appliances from active documents, i.e., GUI-based buttons and toggle switches. Like other compound document frameworks, these cannot transmit codes for viewing and editing documents. To customize distributed computing, particularly network processing between computers, several researchers have explored active networking technologies [25]. However, these existing technologies have focused on configurations for low-level network processing, e.g., routing and QoS protocols, so that they are not suitable for end-users.

As will be discussed later, this framework uses mobile agent technology [12,6]. However, the technology assumes each mobile agent to be an isolated entity that migrates between computers independently of other agents and it does not support any document-centric approaches. We constructed a mobile agent system called MobileSpaces [17] and designed a compound document framework [18,21] based on the system. The previous framework could not be used to define or customize any network processing, because it was only proposed as an application of the MobileSpaces system. Furthermore, there were several serious mismatches between mobile agent-based components and the requirements of document components.

3 Design Principles

This section briefly outlines the framework presented here.

3.1 Requirements

It needs to satisfy the following requirements.

Composition: Like OpenDoc and JavaBeans, our framework needs to be composed of a document or component of nested components that can display visual parts, e.g., text, images, and windows, and that enables us to edit components in-place without opening a separate window for each component.

Application-absence: Components should be distributed and operated without the need for any applications in their current computers. That is, when a computer

receives a component, it must be able to view or edit it, including its inner components, even when the computer lacks applications.

Autonomy and Mobility: Each document or component is an autonomous programmable entity that can determine which components or computers it will go to according to its program code and content, and that can then migrate to that destination.

Reconfiguration: The network processing of documents, components, or plain data can be easily and naturally defined and customized as a combination of basic components by end-users through document-manipulations just like editing documents.

3.2 Component Model

Let us present the basic ideas behind the framework.

Self-contained component: This framework introduces the notion of a *self-contained* component, where the content of each component and its codes are inseparable. When a component is distributed to other computers, the framework not only transmits the content of each component but also its code to the destinations. To our knowledge, no existing software component frameworks, including compound document frameworks, make the code and state of each component indivisible. This makes documents portable. This is because, when a user receives a document, he/she can view or edit it by using its code instead of any applications deployed at its current computer.

Hierarchical composition: Like OpenDoc and JavaBeans, this framework allows a hierarchy of nested components to correspond to visual parts, e.g., text, images, and windows, and conform to two notions: i) Each component can be contained by at most one component, and ii) it can dynamically migrate to other components along with all its inner components. When a component is contained by another component, the former is still an individual component so that it can be removed from the latter. Each component can move into any other component except itself or its descendant components, which may be at different computers, as long as the destination component accepts it.

Service Provider: Each container component is responsible for automatically offering its own services and resources to its inner components and controlling its inner components. When a component requires a service, it migrates itself to one of the container components that can provide the service. The framework also allows container components to process their inner components as first-class objects. As a result, each container component can migrate, save, and destroy its inner components.

Component interaction: The framework enables a component to control the size and layout of its inner components, and to invoke the service methods explicitly provided by its container (and its neighboring components through its container). In other words, a component cannot access any services supported by components other than its container component. This is important in allowing successful migration to occur. If it were not imposed, then migrating a component could mean that the descendants of that component might suddenly find that they could no longer access services on which they had previously relied.

3.3 Network Processing

This framework provides two approaches to enabling components to customize their own network processing. The first is to make components *mobile* in the sense that they can define their itinerary and travel among multiple computers along this itinerary using mobile agent technology. The second enables components to define network processing for themselves or their inner components. The framework provides three types of components as follows:

- **Visual component** stores its visual content within itself. It displays this content in the estate assigned by its container component by using its own program. When a visual component contains other components, it is responsible for managing the estates of its inner component within its own estate.
- **Carrier component** is transparent and can contain more than one component, which may contain one more component. It can carry its inner components along its own itinerary. Moreover, since it can can treat its inner components as first-class objects, it can explicitly restrict or transform its inner components.
- **Forwarder component** is allocated at a component or computer and can automatically transmit its inner components to its destinations. It can also process its visiting components as first-class objects before it forwards them.

Note that visual components can still travel between other components or computers under their control and network processing components can be assembled and operated through GUI manipulations and embedded into a document as visual components.

4 Component and Runtime System

This framework consists of two parts: runtime systems and components. The former can execute components and migrate them to/from other runtime systems, even when underlying systems, i.e., operating systems and hardware, are heterogeneous, since runtime systems and the latter are constructed with Java language.

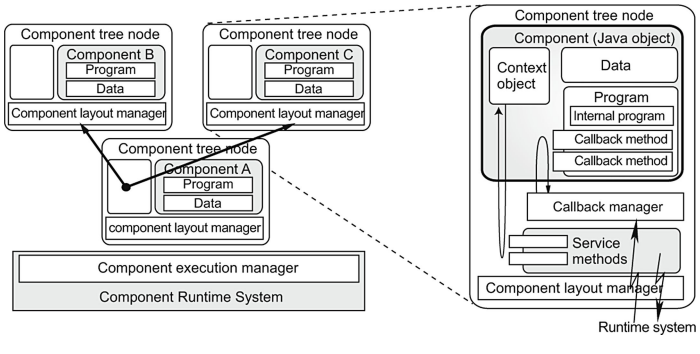


Fig. 1. Component hierarchy and structure of components

4.1 Component

As we can see from Fig. 1, each component is a collection of Java objects wrapped in a component and has its own unique identifier and image data displayed as its icon. All the objects that each component consists of need to implement the `java.io.Serializable` interface, because they must be marshaled using Java's serialization mechanism. Each visual component needs to be defined as a subclass of either the `java.awt.Component` or `java.awt.Container` from which most of Java's visual or GUI objects are derived. To reuse existing software, we implemented an adapter to use typical Java components, e.g., Java Applets and JavaBeans, that are defined as subclasses of the `java.awt.Component` or `java.awt.Container` class within our components[1] We describe a programming model for components in the Appendix.

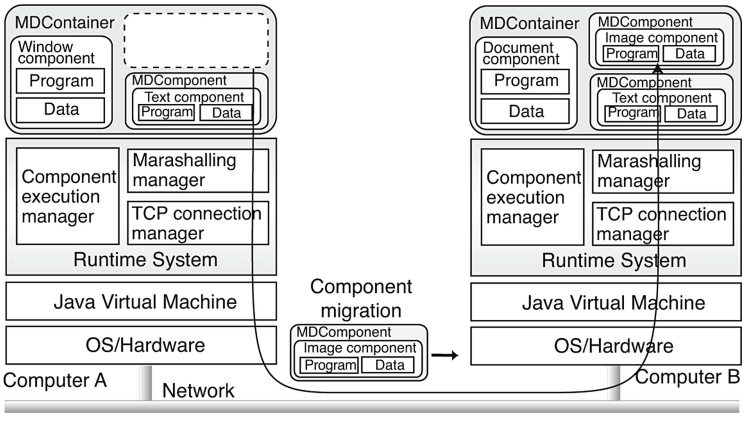


Fig. 2. Component migration between two computers

4.2 Component Runtime System

Fig. 2 outlines the basic structure of a runtime system. Each runtime system governs all the components within it and provides them with APIs for components in addition to Java's classes. It assigns one or more threads to each component and interrupts them before the component migrates, terminates, or is saved. Each component can request its current runtime system to terminate, save, and migrate itself and its inner components to the destination that it wants to migrate to. This framework provides each component with a wrapper, called a *component tree node*. Each node contains its target component, its attributes, and its containment relationship and provides interfaces between its component and the runtime system (Fig. 1). When a component is created in a runtime system, it creates a component tree node for the newly created component. When a component migrates to another location or duplicates itself, the runtime system migrates its node with the component and makes a replica of the whole node.

[1] This is not compatible with all kinds of Applets and JavaBeans, because some of those existing manage their threads and input and output devices depreciatively.

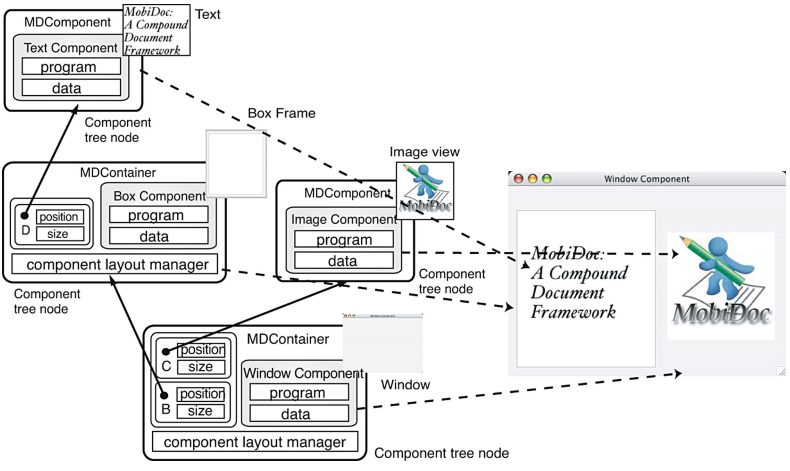


Fig. 3. Component Hierarchy

A hierarchy is maintained in the form of a tree structure of component tree nodes of the components (Fig. 3). Each node is defined as a subclass of `MDContainer` or `MDComponent`, where the first supports components, which can contain more than one component inside them and the second supports components, which cannot contain any components. For example, when a component has two other components inside it, the nodes that contains these two inner components are attached to the node that wraps the container component. Component migration in a tree is only performed as a transformation of the subtree structure of the hierarchy. When a component is moved over a network, on the other hand, the runtime system marshals the node of the component, including the nodes of its children, into a bit-stream and transmits the component and its children, and the marshaled component to the destination.

4.3 Component Manipulation

Each component can display its content within the rectangular estate maintained by its container component. The node of the component, which is defined as a subclass of the `MDContainer` or `MDComponent` class, specifies attributes, e.g., its minimum size and preferable size, and the maximum size of the visible estate of its component in the estate is controlled by the node of its container component. These classes can define their new layout manager as subclasses of the `java.awt.LayoutManager` class.

This framework provides an editing environment for manipulating the components for network processing, as well as for visual components. It also provides in-place editing services similar to those provided by OpenDoc and OLE. It offers several value-added mechanisms for effectively sharing the visual estate of a container among embedded components and for coordinating their use of shared resources, such as keyboards, mice, and windows. Each component tree node can dispatch certain events to its components to notify them when certain actions happen within their surroundings. `MDContainer` and `MDComponent` classes support built-in GUIs for manipulating

components. For example, when we want to place a component on another component, including a document, we move the former component to the latter through GUI manipulations, e.g., drag-and-drop or cut-and-paste. When the boundary of the visible area of a component is clicked, the component is *selected* and displays eight rectangular control points for moving it around and resizing it (Fig. 4 (a)). The user can resize the selected component, move it to another, save it, and terminate it by dragging its handles (Fig. 4 (b)).

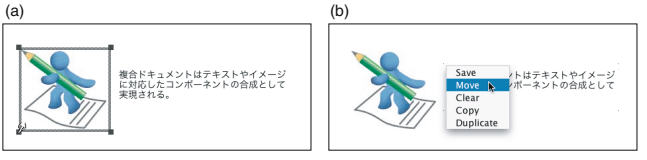


Fig. 4. Editing layout for components and popup-menu for controlling components

We developed various components, e.g., a text viewer/editor component and a JPEG or GIF viewer component (there are several example components in Fig. 9). Note that visual components allow their content to be in arbitrary as well as standard formats, because they have codes for viewing and modifying content. Most Swing and AWT GUI Widgets can be used as our components in the framework without modifications, because they have been derived from the `java.awt.Component` class.

4.4 Component Distribution and Duplication

The runtime system can transmit the marshaled nodes of components to their destinations through an extension of the HTTP protocol.[2] Since the runtime system transmits both the code and state of the components to the destinations, the components can continue processing, even when the destinations are disconnected from the source. Each runtime system also has a built-in mechanism for writing the marshaled component and reading it from the underlying file system, network file system, and database system without losing the component's containment structure or inner components. To duplicate components, the system marshals components into a bit-stream and then duplicates the marshaled component, because Java has no deep-copy mechanisms, which can make replicas of all objects embedded in and referred to from these components. The current implementation treats a component and its clones as independent. Components also have no problems with incompatibility even with different versions, because they contain their codes.

The current implementation uses the Java object serialization package to marshal and duplicate the states of components into a bit-stream. The package does not support the capturing of stack frames of threads. Consequently, our system cannot marshal the execution states of any thread objects. Instead, the runtime system (and the Java

[2] The current implementation can support HTTP tunneling to transfer components outside firewalls.

virtual machine) propagates certain events to components before and after marshaling and unmarshaling them. The current implementation of our system uses the standard JAR file format for passing components that can support digital signatures, allowing for authentication. If inner components embedded in a component share the same codes, the runtime system can detect and remove such redundant codes from the bit-stream corresponding to the marshaled component, including its inner components to reduce the size of the bit-stream.[3]

5 Component-Based Network Processing

Each component for network processing is invisible and has been designed to provide its service to its inner components. A component can directly instruct its inner components to move to another location, and can transform them.

5.1 Carrier Component

Carrier components are transparent, can carry other components to their destinations along their itineraries, and transform their inner components (see Fig. 5).

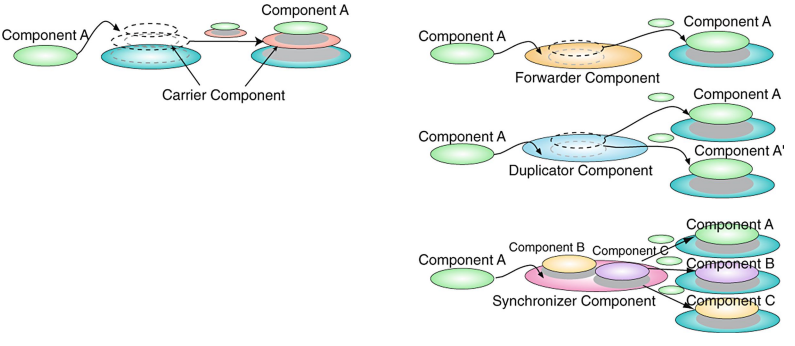


Fig. 5. Carrier component and basic forwarder components for network processing

We developed a language enabling mobile agents to specify their own itineraries from multiple destinations [22]. The carrier components can define their itineraries with the language and migrate to other computers along their itineraries. When the movement of a component deviates from its registered itinerary, the runtime system issues an exception to the component. Moreover, carrier components can encapsulate or restrict their inner components, because they can control them while carrying them, and they can provide a secret-key-based cryptographic procedure to protect these inner components against illegal access or modification.

[3] This optimization mechanism involves a trade-off because its detection of redundant codes is not always lightweight. We intended to disable the mechanism in the evaluations presented in Section 6.

5.2 Forwarder Component

Forwarder components are statically or dynamically deployed at components. When a forwarder component receives a component, it processes the visitor and then forwards it to its target component. A variety of processes for components, e.g., duplication and synchronization, can be defined in derivations from the forwarder component. The current implementation provides basic operations for component migration (Fig. 5). By combining these components, we can easily customize network processing. Since these protocols are given as Java abstract classes, we can easily define further advanced network processing by extending these basic protocols.

- **Forwarder component** can redirect its inner component to other locations. When it receives a component, it automatically transfers the visiting component to its specified destination as long as the destination is within the range that the inner component can migrate to.
- **Duplicator component** can receive another component and then create a copy of its visiting component including all instance variables. The cloned component has the same content as the original.
- **Synchronizer component** can strand its inner components until it can determine whether specified conditions can be satisfied, e.g., the number of inner components, the arrivals of specified components, and time constraints. A typical synchronizer component defines a group of moving components, as a barrier synchronization mechanism for parallel processes. It strands the visiting components inside it, until it receives all the components within the group.

We can define flows of components over a distributed system as a combination of forwarder components like the active routers (or nodes) in active network technology. The components previously described have properties that customize their processing and provide support to GUI editors like those for the property editors developed by Java Beans. The editors allow users to edit the property values of a given type. For example, forwarding components can configure their destinations in their properties and synchronization components can explicitly define their conditions. Moreover, these components can be dynamically deployed at remote hosts through document manipulations because they are still components of compound documents.

Since these components cover most basic functions to implement network protocols, end-users can rapidly and easily implement the protocols they want by combining components. Various types of network processing for components can be implemented as components. Since these protocols are given as abstract classes in the Java language, we can easily define further advanced network processing by extending these basic protocols. Fig. 6 outlines application-specific document-delivery (for workflow management systems) executed by combining basic components.

5.3 Network-Wide Component Manipulation

This framework itself does not support any network-wide component manipulations, e.g., cut-and-paste and drag-and-drop between computers. Instead, such operations can

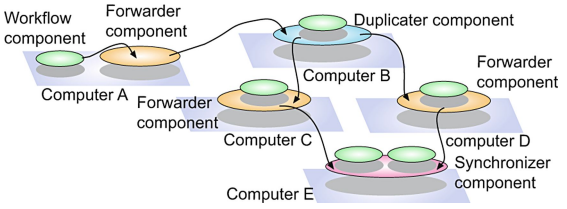


Fig. 6. Combination of basic components for network processing

easily be achieved through the *forwarding* and *duplicator* components. One can also use *forwarding* components as a mechanism to deploy network processing at remote computers. Figure 7 presents a compound document that includes several forwarding components, which automatically transfer other components to specified components at remote nodes. When a user wants to enable new network processing at remote nodes, he/she drags and drops the component that supports the processing to forwarding components corresponding to the nodes. This action transfers that component to the target of the forwarding component. Moreover, forwarding components can have property editors for their target components at the new location in addition to their own editors, allowing us to customize the properties of components deployed at remote computers by using their editors, which can be implemented as plug-in modules. Note that the user interface presented in Fig. 7 has only been implemented as components. Therefore, the interface itself can easily be distributed to other computers. That is, a component can be viewed as the only constituent of the framework. This gives users and programmers a single unified perspective of the system.

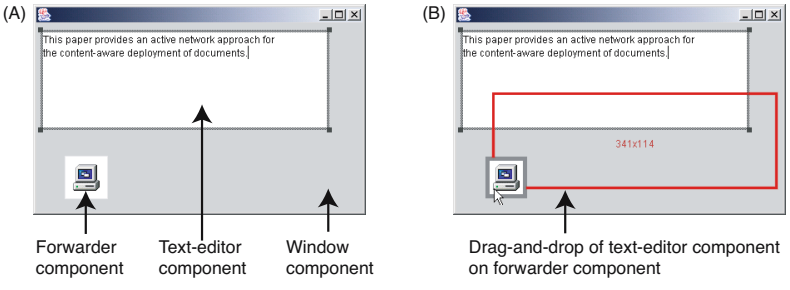


Fig. 7. (A) Screenshot of window component contains text-editor and forwarding components and (B) screenshot of window component when text-editor component is dragged and dropped on forwarding components

Fig. 8 presents a compound document for deploying components for network processing at computers on a network. When three forwarder components contained in the document receive components, they automatically forward their visitors to their target computers. We can easily migrate a component for network processing at each of the forwarders by duplicating the component through our duplicator component. Note

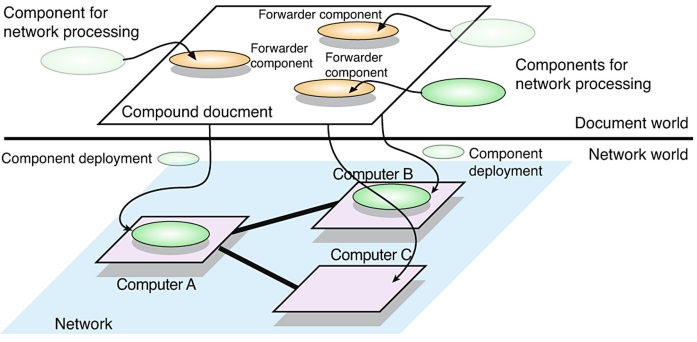


Fig. 8. Compound document for deploying components at computers

that the document is just a configuration for network processing and can be stored in secondary storage as a first class object just like visual components.

6 Current Status

We implemented the framework using Java language (JDK 1.4 or later version), and we developed various components for compound documents and network processing. Since the Java virtual machine and libraries abstract away differences between the underlying systems, e.g., operating systems and hardware, components can migrate between and be executed on runtime systems running on different computers, whose underlying systems may be different.

Fig. 9 has a screen-shot of this framework. The left window is a palette of part components and the center and right windows are compound documents contained in the components corresponding to GUI-windows. When a user wants to place a component on his/her editing compound document, he/she drags the wanted component from the palette and then drops it on the estate of the document. Since the palette itself is implemented as a container component of part components, it can migrate to another computer and be saved in secondary storage. We can register new components, which may be edited or modified, in the palette through GUI-based data-transfer operations, e.g., drag-and-drop or cut-and-paste. End-users can also define and customize their application-specific network processing by combining forwarder components through GUI manipulations in the same way as if they were editing visual components in documents.

6.1 Basic Performance

Even though our current implementation was not built for performance, we evaluated some basics of the framework.

Component Migration. We conducted a basic experiment on component migration with computers (Pentium III 1.2-GHz with Windows XP and Sun's JDK 1.4.2). The time for component migration measured from one container to another in the same

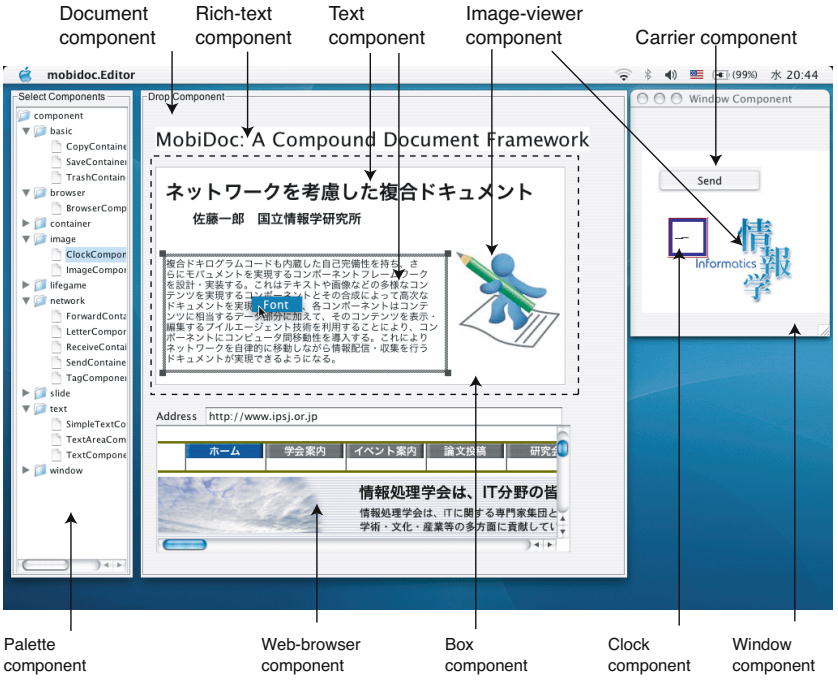


Fig. 9. Example compound documents

hierarchy was 10 msec., including the cost of drawing the visible content of the moving component and checking whether the component was permitted to enter the destination component. The cost of component migration measured between two computers connected through a Fast-Ethernet was measured at 64 msec.. The cost was the sum of marshaling, compressing, opening a TCP connection, transmitting, acknowledging, decompressing, verifying security and consistency, unmarshaling, laying out the visual space, and drawing the content. The moving component was a simple text viewer and its size (sum of code and data) was about 9 KB (zip-compressed). The latency of component migration was reasonable for a Java-based visual environment for exchanging compound documents between computers.

Component Size. Since each component in this framework not only contains its content but also program code, documents or components transmitted over a network or stored on secondary storage tend to be large. We compared the sizes of documents in this framework and the size of documents created with MS-Word, which is the most typical office application.[4] The sizes of two typical kinds of content were as follows:

[4] Note that the sizes of documents, which contain multimedia content, created with MS-Word may vary for no reason, so that we could not systematically compare the sizes of our documents and MS-Word documents. The results presented in this section have not always been generalized but have focused on several samples.

– The first content was plain text. The size of the component for viewing and editing the text was 5.6 KB (0), 6.3 KB (1,000), 9.9 KB (10,000), and 19.5 KB (100,000), whereas the size of the document created with MS-Word was 19.5 KB (0), 21.0 KB (1,000) 47.1 KB (10,000), and 306.2 KB (100,000), where the numbers in parentheses represent the length of the text where the text-content was part of this paper.
– The second was image content within the dotted box in Fig. 9. The content is composed of three components: box, text, and image-viewer, where the first contains the second and third components. The content was 68 KB, where the document corresponding to the content created with MS-Word was 24 KB.

The size of the text component was not proportional to the length of the text because our components were migrated over a network or stored in secondary storage and they were compressed in JAR-format, which was ZIP-based compression. This meant that our components were not always larger than documents created with commercial applications, e.g., MS-Word. We also found that the size of our components greatly depended on the kind of content and the complexity of their codes. When various types of components were embedded into a document, the document tended to be larger, because each type needed to embed its code for the content type into the document. Otherwise, the size was often smaller than that of corresponding documents created by existing applications, when the document contained a few component types. The size of our components was reasonable because the network bandwidth and the capacity for storage have recently increased.

7 Application

We developed a variety of components based on this framework. This section introduces several of these and their uses.

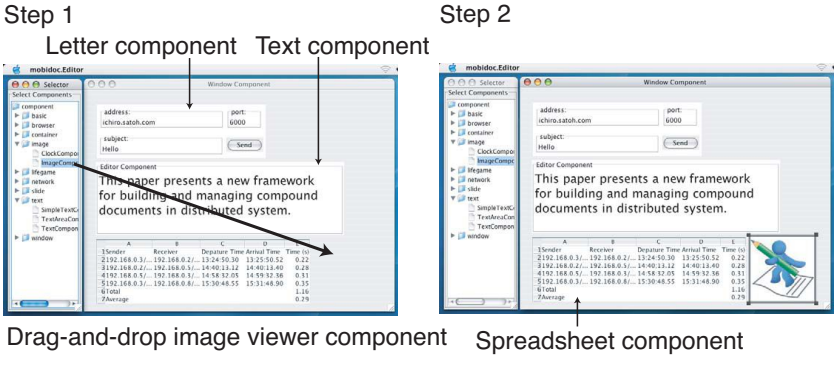


Fig. 10. Letter document

7.1 Compound Document Letter

Although documents are often sent to their destinations through electronic mail systems, it is difficult for these systems to send the documents to multiple destinations along specified itineraries. To solve this problem, this framework provides carrier components that can travel between multiple computers under their own control. Figure 10 shows a letter constructed as a compound document. It consist of three components: carrier, image viewer, and text components. The carrier component has a UGI corresponding to the header on an electronic mail and it can migrate with its inner components, i.e., the text and spreadsheet components, to its destination. We can easily add new components to the letter compound document by dragging them from the palette and dropping them on the document. Since not only the content but also the codes for viewing and editing the content of the components are transferred to the destination, the letter document can be viewed at the destination with no applications necessary for the content. The spread-sheet component has its own profile because it can automatically record when it has been (un)marshaled and where it has visited.

Most electronic mail systems disallow letters from traveling among multiple destinations along their own itineraries. We developed a legacy decision-making system, called *ringi*, for group decision-making, which has been widely used throughout Japan. When an employee proposes something to his/her company, he/she describes the proposition on a workflow document, called a *ringi-sho*. The document must be handed to all sections involved with the proposed issue. When the managers of the sections deem the

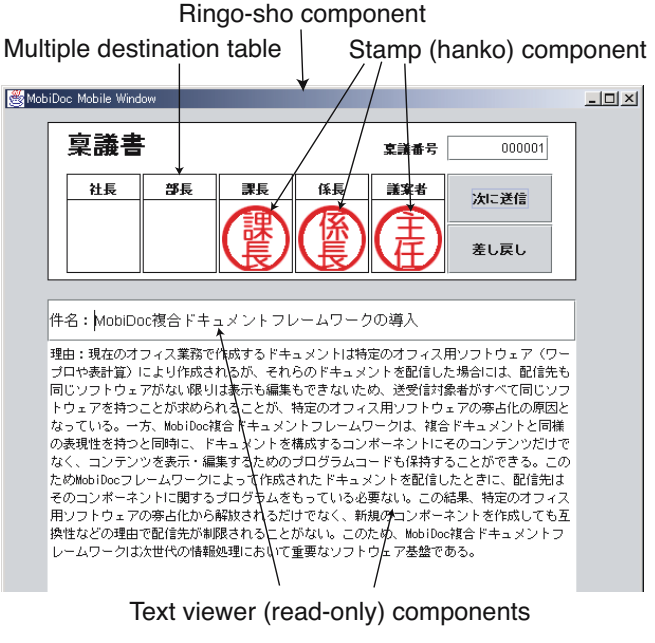


Fig. 11. Ringi-sho compound document

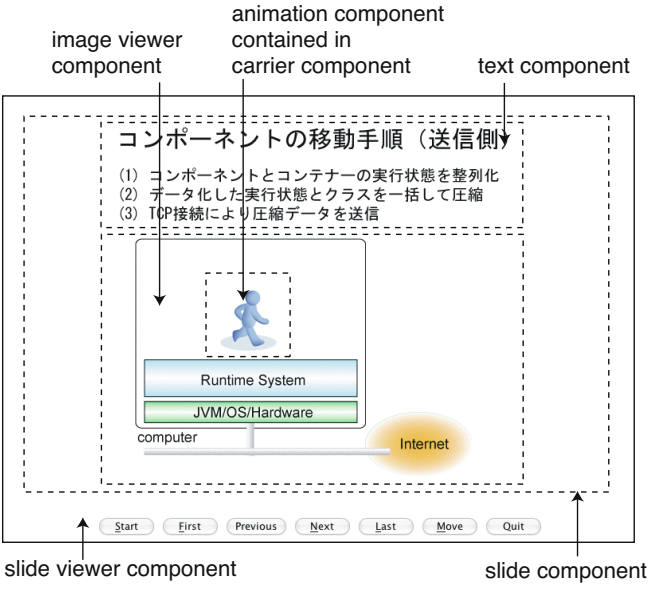


Fig. 12. Slide component

proposal to be of worth, they stamp it with their hanko, or give rights access through its carrier component, and then resume its process as seen at the right of the computer.

7.2 Distributed Presentation System

This system is unique to other existing presentation systems because it can exchange slides or its visual parts, e.g., text and images, between different computers. Fig. 12 is a slide-presentation compound document that can contain more than one slide component inside it, where each slide component corresponds to one slide and can contain and view one or more visual components. It stacks its slides by using the

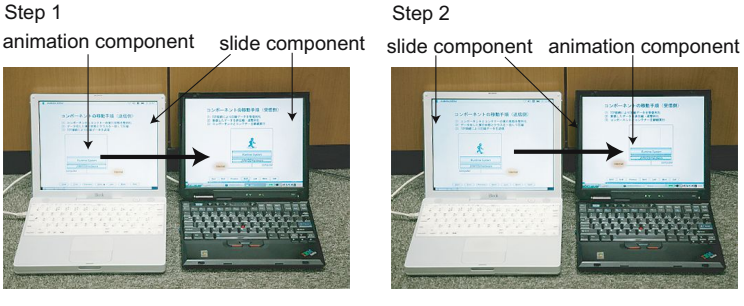


Fig. 13. Migration of components between slide components running on different computers

`java.awt.CardLayout` class as a layout manager. It enables us to change the order of the stack and exchange slides with other slide-presentation compound documents running on different computers through a GUI-based control panel at the bottom of its window. The center image is a GIF-animation-viewer component contained in a carrier component. When the carrier component is made active by clicking the mouse within its estate, it migrates to its specified component or computer. Figure 13 shows that the animation-viewer component has migrated between slides from the left (Macintosh) to the right computer (Windows PC) through its carrier component and has then resumed its animation at the right of the computer.

7.3 Application-Specific Document Distribution

This framework enables end-users to easily define and operate CSCW applications. The third example is an editing system for an in-house newsletter. Each newsletter is edited by automatically compiling one or more text parts, which are written by different people as we can see from Figure 14. A newsletter compound document has one page component, which can contain editor components for visual content, e.g., text and images. When the newsletter is being edited, it forwards the page component to a duplicator component to make replicas of the component to match the number of writers. The duplicator component then migrates the replicas to forwarder components so that each of the page components is forwarded to a window component on its writer's computer. When it arrives at the destination, it displays a window for its editor program on the

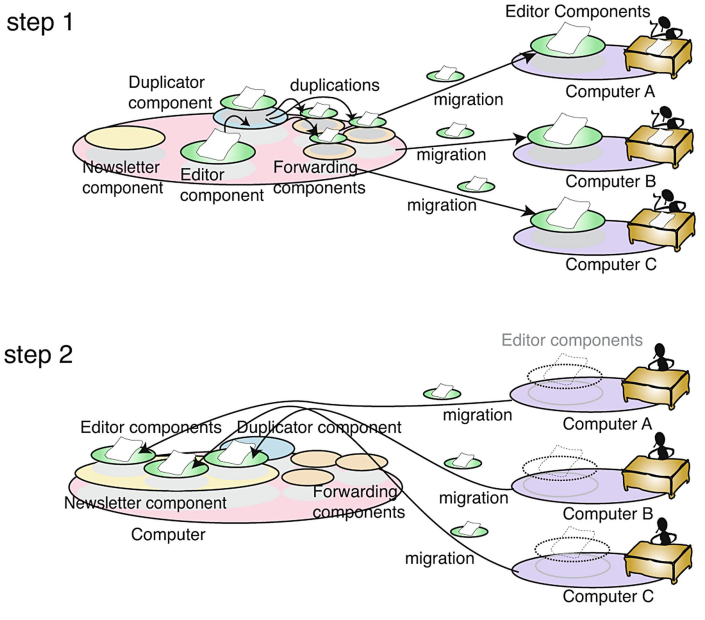


Fig. 14. Newsletter editing system

screen of the computer to assist the writer. Each writer can add his/her visual components to the page component. If a page component can contain sharing components, the components contained in the sharing components can maintain common content for all the replicas, even when one of the replicas has changed. For example, each page of the newsletter can have common text and images in its header and tail. Sharing components are located at the head and tail of each page. When a writer modifies text in text components or images in image-viewer components containing sharing components, this modification can be reflected on the heads and tails of all pages, while other writers are editing their pages. This modification goes back to the document after the writer has finished writing his/her text and then the document arranges the arriving components as a bound set. Since the newsletter document, duplicator, and forwarder components are still mobile, they can easily be deployed and coordinated according to the requirements of applications.

8 Conclusion

We presented a compound document framework for document-centric network processing. The framework made three novel contributions. The first introduced the notion of a component hierarchy and mobile components. This notion enabled an enriched document to be composed of various components and to migrate between these components, which may run on different computers, under its own control. The framework provided several value-added mechanisms for visually manipulating components embedded in a compound document and for seamlessly combining multiple visible components into one. The second contribution was that it made the content of each component and its codes inseparable. It allowed us to view or modify components without the need for any applications. It was also useful in protecting content because it prevented the content of components from being accessed by external systems and it enabled us to easily insert security mechanisms in each component's codes to view or edit its content. The third contribution enabled documents to pass other documents from/to other components or computers. Components were introduced as the only constituent of our network processing for documents or components. It also offered several basic operations for network processing, e.g., carrying, forwarding, duplication, synchronization, and sharing. The operations were still document components; they could be dynamically deployed at local or remote computers through GUI-based manipulations. It therefore allowed an end-user to easily and rapidly configure network processing in the same way as if he/she had edited the documents. In fact, a variety of advanced network processing can be implemented as a collection of components for network processing. We constructed a prototype implementation of this infrastructure and its applications.

To conclude, we would like to point out further issues that need to be resolved. Resource management and security mechanisms in the current system were incorporated in a relatively straightforward manner. These should now be designed to incorporate compound documents. This framework does not support interoperability with other document frameworks, because these assume that each component can be viewed or modified by external applications. Nevertheless, it is open to supporting such existing document formats because the framework can provide viewers and editors components

that can encapsulate such documents with their viewers and editors corresponding to their applications. When a component migrates to another component or computer, its visual resources, i.e., the size of its estate and colors in the destination, may not be the same as those in the source. It must adapt its visibility to the resources available in the current location; however, the current implementation relies on Java's layout manager. We need a more sophisticated and flexible mechanism to enable adaptation.

References

1. Apple Computer Inc. (1994) OpenDoc: White Paper, Apple Computer Inc.
2. K. Arnold, and J. Gosling, The Java Programming Language, Addison-Wesley, 1998
3. K. Brockschmidt, Inside OLE 2, Microsoft Press, 1995.
4. Cable, L. (1997) Extensible Runtime Containment and Server Protocol for JavaBeans, Sun Microsystems, http://java.sun.com/beans.
5. D. P. Friedman, M. Wand, and C. T. Haynes, Essentials of Programming Languages, MIT Press, 1992.
6. A. Fuggetta, G. P. Picco, and G. Vigna, Understanding Code Mobility, IEEE Transactions on Software Engineering, vol.24, no.5, 1998.
7. The GNOME Project, Bonobo, http://developer.gnome.org/ arch/component/ bonobo.html, 2002.
8. Y. Goldberg, M. Safran, and E. Shapiro, Active Mail - A Framework for Implementing Groupware, Proceedings of ACM CSCW'92, pp. 75-83, ACM Press, 1992.
9. Hamilton G. (1997) The JavaBeans Specification, Sun Microsystems, http://java.sun.com/beans.
10. G.H. ter Hofte, H.J. van der Lugt, CoCoDoc: a framework for collaborative compound document editing based on OpenDoc and CORBA, Proceedings of International Conference on Open Distributed Processing and Distributed Platforms, pp.15-33, 1998.
11. G. Kiczales, et al, Aspect-Oriented Programming Proceeding of European Conference on Object-Oriented Programming (ECOOP'97), LNCS, vol. 1241, Springer, 1997.
12. B. D. Lange and M. Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.
13. R. Litiu and A. Parakash, Developing adaptive groupware applications using a mobile component framework, Proceedings of ACM conference on Computer Supported Cooperative Work (CSCW'2000) , pp.107 - 116, ACM Press, 2000.
14. J. Morin, HyperNews, a Hypermedia Electronic-Newspaper Environment based on Agents, Proceedings of HICSS-31, pp.58-67, 1998.
15. M. Potel and S. Cotter Inside Taligent Technology, Addison-Wesley, 1995.
16. D. Rogerson, Inside COM, Microsoft Press, 1997.
17. I. Satoh, MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System, Proceedings of International Conference on Distributed Computing Systems (ICDCS'2000), pp.161-168, IEEE Computer Society, April 2000.
18. I. Satoh, MobiDoc: A Mobile Agent-based Framework for Compound Documents, Informatica, vol.25, no.4, pp.493-500, December 2001.
19. I. Satoh, Building Reusable Mobile Agents for Network Management, IEEE Transactions on Systems, Man and Cybernetics, vol. 33, no.3, part-C, pp.350-357, August 2003.
20. I. Satoh, Configurable Network Processing for Mobile Agents on the Internet, Cluster Computing, vol. 7, no.1, pp.73-83, Kluwer, January 2004.
21. I. Satoh, A Compound Document Framework for Multimedia Networking, Proceedings of 1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA'2005), pp.80-87, IEEE Computer Society, February 2004.

22. I. Satoh, Selection of Mobile Agents, Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS'2004), pp.484-493, IEEE Computer Society, March 2004.
23. I. Satoh, Network Processing of Documents, for Documents, by Documents (short position paper), to appear in Proceedings of ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'2005), Lecture Notes in Computer Science (LNCS), December 2005.
24. C. Szyperski, Component Software (2nd), Addison-Wesley, 2002
25. D. L. Tennenhouse et al., A Survey of Active Network Research, IEEE Communication Magazine, vol. 35, no. 1, 1997.
26. P. Werle, F. Kilander, M. Jonsson P. Lonqvist, C. G. Jansson, A Ubiquitous Service Environment with Active Documents for Teamwork Support, Proceedings of 3rd International Conference on Ubiquitous Computing (UBICOMP'2001), Lecture Notes in Computer Science, vol. 2201 pp.139 - 155, Springer, 2001.
27. J. E. White, Mobile Agents, in Software Agents (eds. J. M. Bradshaw), pp.437-472, AAA Press/MIT Press, 1997.

Appendix: Component Programming Model

Each visual component needs to be defined as a subclass of either the `java.awt.Component` or `java.awt.Container` from which most of Java's visual or GUI objects are derived.[5] When a component is moved to another location or its estate is resized, its component node invokes `repaint()`, which is a callback method to redraw the visible content of the `java.awt.Component` class. Therefore, each visual component is required to define its own visual features in addition to its application logic.

The framework also provides an event listener interface, called the `MDLifecycleListener`, which defines several callback methods to inform components about changes in their lifecycle state from the runtime system, like the Java AWT event listener interfaces. If a component implements the interface, before or after the lifecycle of the component has changed, the runtime system invokes these methods.

```
interface MDLifecycleListener
implements java.io.Serializable {
  void create(MDContext mdcx) throws IllegalAccessException;
  void destroy(MDContext mdcx) throws IllegalAccessException;
  void save(MDContext mdcx) throws IllegalAccessException;
  void suspend(MDContext mdcx) throws IllegalAccessException;
  void duplicate(MDContext mdcx) throws IllegalAccessException;
  void leave(MDContext mdcx, URL dst) throws IllegalAccessException;
  void arrive(MDContext mdcx, URL src) throws IllegalAccessException;
  void add(MDContext mdcx, URL src) throws IllegalAccessException;
  void remove(MDContext mdcx, URL src) throws IllegalAccessException;
  ....
}
```

A component, which implements the `MDLifecycleListener` interface, can define the processing it requires as the behaviors of the methods. These methods can also define processing for authentication and content rights management. Note that, even

[5] `java.awt.Container` is a subclass of `java.awt.Component`.

when a component does not support the interface, the Java virtual machine invokes the `writeObject()` or method of a component, before the component is marshaled or after it is unmarshaled. The `MDContext` class in the `MDLifecycleListener` interface defines service methods available in components as follows:

```
class MDContext implements java.io.Serializable {
  public void go(URL url) throws NoSuchHostException { ... }
  public URL save(URL url) throws IOException { ... }
  public ComponentID duplicate() { ... }
  ....
}
```

When a component executes `go()` or `save()`, the runtime system migrates (or stores) the component, including its inner components, to the component (or in the file) specified as `url`. The `duplicatee()` method makes a replica of the component itself.

Shepherdable Indexes and Persistent Search Services for Mobile Users

Michael Higgins, Dominic Widdows, Magesh Balasubramanya, Peter Lucas, and David Holstius

MAYA Design Inc.*
{higgins, widdows, balasubramanya, lucas, holstius}@maya.com

Abstract. We describe a range of designs for supporting rich search queries in a peer-to-peer network. Our implementation is based upon uniquely identified data objects which are replicated upon request by agents called Shepherds. Several abstract data structures are built upon this framework, supporting dataset management, lexical search, and distributed GIS interfaces in an application called the Geobrowser. Our results demonstrate that it is possible to layer higher-level data structures upon a basic peer-to-peer transport and replication layer. When users perform a given query, parts of the index as well as the query results themselves are shepherded to the user's local venue. A natural benefit of this approach is that mobile users can repeat previous searches if they become disconnected from the rest of the network. Some of the data structures that prove to be successful are peer-to-peer adaptations of traditional indexing structures. We review some of the properties that lead to successful designs in this domain, giving examples of deployed systems in the Geobrowser.

1 Introduction and Related Work

This paper presents methods by which indexes supporting rich content-based search interfaces can be built in a peer-to-peer system, by layering sophisticated index structures on a common object location and routing infrastructure.

Cooperative systems for storage and collaboration have experienced a surge of investment and activity over recent years. Of particular interest are flexible decentralized systems, such as Gnutella [1] and OceanStore [2], where nodes may join and leave at any time, and where peer-to-peer replication ensures the high availability of data. One of the major drawbacks to these systems, however, is the lack of cooperative indexing and querying techniques. Users expect to be able to search these systems just as they search the Web, with the same net performance and quality of results, even though current Web search engines are supported by coordinated and/or centralized systems built on guarantees unmet by peer-to-peer storage. The general information retrieval methodologies

* This work was supported by Defense Advanced Research Projects Agency grant SB031-008 for the Cluster-Based Repositories and Analysis project.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1576–1593, 2006.
© Springer-Verlag Berlin Heidelberg 2006

outlined for coordinated distributed systems, on which much research has been conducted [3,4], are not directly applicable.

In peer-to-peer systems, basic one-dimensional queries such as keyword and prefix search are currently supported, but in much the same manner as routing or messaging (see, for example, [5,6,7,8]), with logarithmic guarantees of low order. Nodes answer queries about data objects that they store, and route queries they cannot directly answer to a more knowledgeable host. Hybrid techniques improve performance by balancing local work with global work, e.g., by sharing a global keyword lexicon.

Replication of search structures, such as B-trees, is not new [9,10,11]. Direct storage of search structures in a persistent object store, where chunks of data have unique identifiers but no type, has also been examined [12]. Experiments with replicated search trees have shown that they lead to high availability, overcoming some of the bottlenecks associated with hierarchical structures [13,10].

But even in systems that make use of index replication, the indexes are generally treated in a way that is distinct from other data — index components are not freely replicated in the same epidemic manner,[1] partly because the information they carry is tied to specific nodes in the system. One approach that *does* propagate index (*key*, *value*) pairs epidemically in a peer-to-peer network is the Passive Distributed Indexing system of Lindemann and Waldhorst [11,14]. While this work is the nearest in spirit to our approach, it still treats index replication and document propagation separately using different infrastructure.

We take a different approach. Instead of having support for a few specific query types (such as keywords or prefixes) built into the infrastructure, we propose a deliberate layering of search functionality on top of the basic object location and routing infrastructure. Our system models index structures using exactly the same attribute-value syntax as is used to model other data objects, such as those used to represent objects in the physical world. Optimistic replication [15] is used to to disseminate the components of a shared index, just like any other data.

Our experiments demonstrate the potential of this approach in an application called the Geobrowser, which is a portal to a peer-to-peer "Universal Database" network [16]. As in the early Visage system [17] and its descendants, all information objects in this network are represented in *u-forms*, a u-form being a mutable bundle of attribute-value pairs identified by a universally unique identifier or UUID. In the peer-to-peer version of this architecture, u-forms are optimistically replicated upon demand to many different venues in the network by artificial agents called *Shepherds*. An index that enables persistent search operations (in the sense that in performing a search, a user will have the data necessary to repeat that search replicated to their local venue) is described as *Shepherdable*.

[1] In this context, the term *epidemic* is used to describe replication strategies in which a new version of an object can be transmitted between any two venues that are in contact.

This approach carries immediate benefits. Since replication is a core component of peer-to-peer systems, our system automatically gains the advantages afforded by current peer-to-peer storage layers, such as high availability [18] and local caching to support offline activity. Since u-forms are extensible and have dynamic schemata [19], it is always possible to introduce new data structures enabling specialized query functionality or optimizations in performance. In this paper we describe specific implementations of B-trees [20] and R-trees [21], scalable collections (that can be used to support many structures including inverted indexes [4]), as well as a novel self-indexing spatial structure that supports the Geobrowser's distributed mapping application. Each of these structures has been built and optimized using standard u-form data objects, without requiring any modification to the underlying storage and transportation layers.

We also believe that separating content-driven indexes from the storage and transportation layers will have long term benefits. This is because our system will be able to take advantage of improvements in peer-to-peer storage and transportation, and advances in these layers will automatically improve indexing and search reliability and performance with minimal or no changes to application layer code.

The long term success of the Geobrowser platform depends on collaborative addition of data, which implies collaborative contributions to datasets and indexes. Prior work with replicated structures assumes either the existence of a coordination mechanism for contributors to a shared index: a load balancer [10,9]; a locking mechanism; or a recovery mechanism [22] to ensure that actions on the global index can be effectively serialized. Our general approach to collaboration with u-forms is discussed in more detail in [23].

The rest of this paper is structured as follows. Section 2 describes the underlying database architecture used in our implementations. Section 3 begins to describe higher level u-form data structures, with a design for distributing large collections of u-forms into several navigable chunks. Section 4 describes the use of shepherdable B-trees for search in a totally ordered keyspace, and Section 5 describes a shepherdable R-tree structure with optimizations for distributed spatial search. Section 6 gives the results of some preliminary tests of the R-tree data structure against a large reference collection. Section 7 describes the novel self-indexing structure used for the Geobrowser's distributed mapping environment. Section 8 gives a brief summary of some of the issues involved in creating and maintaining our distributed index structures, and Section 9 analyzes some of the data structures that were prototyped but performed poorly in the Shepherds system.

2 U-Forms, Replication and Shepherding

This section describes the underlying database architecture used in our applications, emphasizing some of its most relevant properties. All of the implementations described in this paper use a "Universal Database" architecture in which all information is represented in "u-forms", a u-form being an extensible bundle

of attribute-value pairs identified by a universally unique identifier or UUID. A more detailed view of u-forms and the VIA repository is given in [16].

A formal similarity has been suggested between the (UUID, attribute, value) triple and the (URI, predicate, value) structure of the Semantic Web's RDF graphs [24] (and numerous other "triple store" structures for knowledge representation). It is important to note that a u-form in itself is an abstract data object, and physical copies of u-forms can be serialized using many formats, including (for example) XML. For efficiency reasons, our repositories represent u-forms using a recursive bytecode serialization called VSMF (Visage Standard Message Format).

The u-form architecture developed out of the Visage projects for collaborative information visualization [17,25], and the policy of representing information in u-forms is referred to as the Visage Information Architecture (VIA). The architecture is currently used to support commercial information collaboration applications such as CoMotion and CPOF (Command Post of the Future, a command and control system used by the US military [26]), as well as MAYA Design's Geobrowser and a variety of community information websites [27]. An important requirement of the Geobrowser design is that any information a user accessed in the past when online should still be available offline, along with the search functionality used to locate and select that information.

An important property of u-forms is that their values may contain relations (that function as pointers or external references) to other u-forms. U-forms themselves are by definition location independent, and thus may be replicated to many different venues in the system. This includes the local venue of a user whose contact with the network is through a mobile and sporadically connected device. Over the years, our implementations have increased in scale, making the adoption of peer-to-peer implementations naturally appealing. The adaptation of the u-form concept to a peer-to-peer environment is managed by artificial agents called *Shepherds*. Client applications interact with the Universal Database by reading attributes of u-forms from their local repository interface, and writing the values of certain attributes. The shepherds system makes this possible by performing the following actions:

- If a user or client requests a u-form which is not present in that user's local repository, the shepherds try to locate a copy in the network and replicate an instance of that u-form to the user's local venue.
- The shepherds try to ensure that all replicas of a u-form hold the same contents.

Instances of u-forms may be located in the network by a variety of indexing techniques such as manually configured routing tables, centralized Google-like indexing services, and/or distributed hash-tables [28,29]. Considerable research and development has been carried out by ourselves and others on this problem, and while it is difficult, initial solutions exist and it is reasonable to believe that they are not bounded in scale. For the purpose of this paper, we assume that the storage layer works efficiently and scales effectively.

Nonetheless, some operations are costly in our system. These include:

- **Following "null" references.** If a client somehow generates or receives a UUID of a u-form that does not exist anywhere in the network, the process of requesting this u-form causes expensive and unrewarding network traffic, also causing high latency for the user before a "null" answer is returned.
- **Conflicts.** If two users update the same u-form concurrently, this will generate a conflict when the shepherds realize that there are two incomparable versions of the the u-form.

In general, it is difficult to resolve u-form conflicts automatically, and a more stable approach is to design information architectures for collaboration that distribute authority and avoid conflicts on the same u-form [23]. This is of relevance when trying to organize writes to shepherdable indexes.

3 Managing Collections and Datasets

In this section, we begin to give examples of distributed data structures created out of u-forms. In the earliest examples of Visage, shared collections are represented in a single collection u-form containing several UUID references in its `members` attribute. Users can add/remove u-forms to/from a collection using drag and drop. The whole collection can be dragged onto a map frame or a bar chart, enabling a user to see and select objects in certain geographically or statistically defined regions.

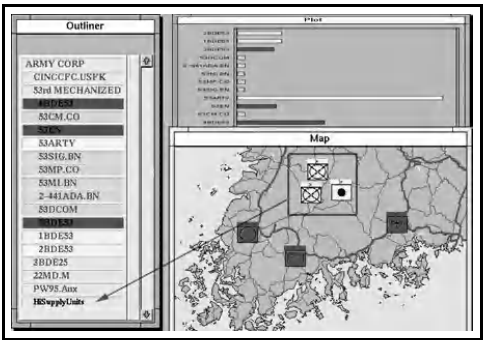


Fig. 1. Visage users morph collections of objects from one display to another

Unfortunately, the method scales poorly. This is because the basic operation of reading or writing an array of UUIDs to the repository is linear in the length of the array. For large collections, a better method is to break such collections up and distribute their members over several linked u-forms. One simple way to do this is to chunk the collection into smaller collections of (e.g.) 100 members each, and create a linked list structure out of the new chunks. By adding extra information to this data structure, performance can be enhanced considerably, as follows:

- A pointer from the head to the tail of the collection can guarantee (amortized) constant time append.
- By recording an associative hash of the member UUIDs on the head node, the fact that edits have been made can be recorded, and checked by other users without scanning the collection.

– By adding a B-tree structure [20, p. 262] or skip list pointers [30], access to any part of the collection can be provided in $O(\log N)$ space and time.
– Given suitable annotation and refactoring of child node counts or skip probabilities, an entire collection can be added recursively to another collection.

The combination of these techniques leads to the creation of a u-form data structure that we call a "Navigable Recursive Scalable Collection". Such a collection, displayed in the Geobrowser interface that enables peer-to-peer browsing, navigation and editing of the collection, is shown in Figure 2.

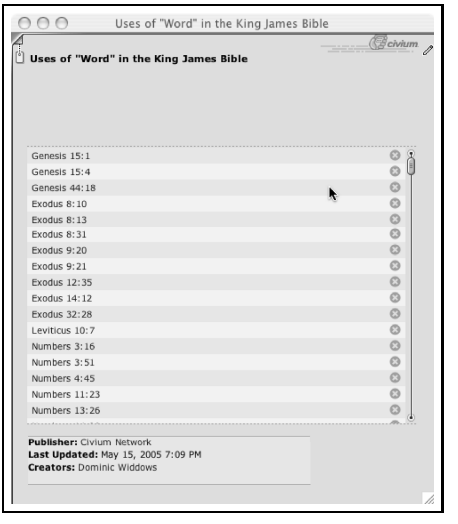


Fig. 2. Interface enabling distributed edits of a collection

In large collections, the positions for items in scalable collections are computed from knowledge of how many descendants each individual node in the B-tree has, and so B-tree nodes in scalable collection implementations have to carry enough information about how many descendants they have for quick comparison with their siblings.

Such a design is typical of the issues involved in adapting complex data structures to a peer-to-peer environment for distributed collaboration. A scalable design for building and editing a collection can be achieved using traditional techniques such as B-trees [20, pp. 262-265] or skip lists [30]. Neither method requires all of the member u-forms to be present in any one location, and enables the construction of user interfaces that perform well for the collection management operations. The choice between these data structures depends on intended usage patterns. We have found the B-tree representation to be more space efficient for static collections. On the other hand, skip lists (being a probabilistic data structure) are more robust in the case where several users in different venues may be editing the collection frequently and asynchronously.

The Geobrowser interface demonstrates that the scaling problem can be solved so that a collaborator in a peer-to-peer network can have apparent access to a dataset containing several millions of u-forms, while only needing to shepherd the u-forms they are interested in and a handful of navigational u-forms that enable the user to find information effectively.

4 Distributed B-Tree Indexes for Lexical Search

Once a user has access to a large distributed dataset, it is natural to wish to find (for example) all u-forms in a collection whose `name` attribute contains

a particular word. Such functionality is provided using the support of a B-tree structure [20, pp. 262-265], represented in u-forms to make a shepherdable index.

The main difference between our use of B-trees for building search trees, and their use for managing navigable scalable collections in the previous section, is that the records in search trees are key-value pairs, whereas in the scalable collection, the records are simply values (UUIDs of member u-forms) whose 'key' is effectively its position in the collection. The search tree behaves like a distributed associative array or dictionary, whereas the scalable collection behaves like a distributed array.

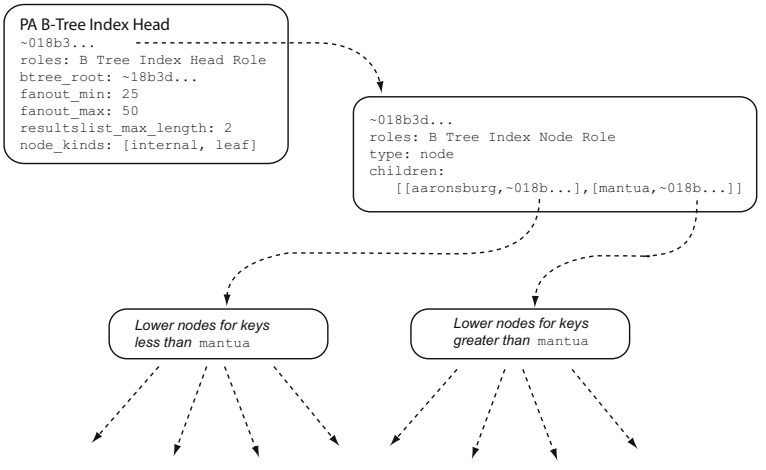


Fig. 3. Upper nodes in B-tree of names of populated places in Pennsylvania

An example u-form B-tree for searching for populated places in Pennsylvania according to their `name` attribute is shown in Figure 3. To search the tree for a particular key such as *pittsburgh*, one proceeds from the head node to the root node, at which point the keys *aaronsburg* and *mantua* are available. Since *pittsburgh* is between *mantua* and the end of the alphabet, the next u-form requested is the one whose UUID is listed after the *mantua* key in the root node, and so on.

The most important attribute that enables this descent is the `children` attribute, which stores a sorted array of (*key*, *UUID*) pairs. The rule is simply that the query should follow the UUID relation of the highest key that precedes the query expression.

The search starts at a separate "head node" rather than a constant "root node", because B-tree root nodes are occasionally split and subsumed due to rebalancing. The head node records structural data about the tree, including the maximum and minimum fanout allowed in nodes below the root.

There is also a `resultslist_max_length` attribute that is used for deciding how many results to pack into a leaf node, based upon how many values match

a particular key. For example, the query term *highland* matches with the name attribute of 19 places in the dataset, and since this is a reasonably large number, this collection of results is posted in a separate u-form. On the other hand, the term *versailles* only occurs in 2 placenames, and as an optimization, these 2 results are placed directly in the leaf node.

This is typical of a standard trade-off in information retrieval, between the structure of the key-space or lexicon, and the structure of inverted indexes or 'postings lists' (see e.g., [4, §4]). In choosing to push some of the postings into the lexicon itself, we are reducing latency for infrequent lexical items, at little extra up front space cost.

Other applications of the B-tree data structure for lexical searching operations are as follows.

- **Full text search.** Utilizes the B-tree data structure to keep a lexicon, and the scalable collection of Section 3 to store the inverted indexes or postings lists. Extra information for more sophisticated indexing, including term-weights and offsets, can be placed in attributes parallel to the members of the postings list collection.
- **Boolean semantics for collections.** The scalable collection design of Section 3 does not enable an efficient Boolean membership test. Such a requirement can be supported by creating and maintaining an auxiliary B-tree whose keys are the UUIDs of the collection, ordered lexically.
- **Virtual references.** Another use of UUIDs as keys is to record when one u-form has referred to another. The use of indexes of this sort to store virtual references in support of backpointers, standoff annotation and publisher-contributed content is described in [31,23].

5 Distributed R-Tree Indexes for Spatial Search

This section describes the design and use of a shepherdable index for low-dimensional continuous data, which enables the Geobrowser to dynamically find and display local content from large geospatial and geotemporal datasets.

The most stable solution to this problem has in practice been a distributed R-tree approach. R-trees were introduced by [21] and are described along with related structures in [32, §6.3]. In an R-tree, the index keys are bounding boxes in an N-dimensional key space, and are ordered by containment. That is, the children of a particular tree node A are index nodes or data items whose bounding boxes are contained in the bounding box of A. New items can be added to the index by adding them to the node whose bounding box volume is minimally increased by the insert operation, and nodes are (heuristically) split by trying to minimize overlap and minimize the amount of "dead space", that is, the volume within each bounding box that is not actually contained in any of its children.

For many of the datasets we wish to visualize in the Geobrowser, there are some items that are considered more important than others, and these items should be presented first. For example, for a dataset of world populated places,

large cities such as *Shanghai* should be presented before smaller towns and villages. In many user interfaces, it is appropriate to limit the number of items shown to (for example) 100 objects from each dataset.

5.1 Priority Item Annotation

The user experience can be greatly enhanced by adding prioritized items to higher nodes on the R-tree.[2] Prioritization enables a search to return the p most important items within a particular bounding box, without searching all the way down to leaf nodes in the tree to obtain these items. A simplified example is shown in Table 1, which contains the main attributes and values of the current root node of the u-form index of the world's populated places. This shows the bounding boxes of the children, and the first five prioritized items (*Shanghai, Mumbai, Seoul, Moscow, Manila*). A query that only wants to find the world's five or fewer most populated places only needs to look at this u-form.

Table 1. Current root node of World Cities populated places index. The items in italic type are UUID relations to other u-forms.

`aggregate_child_count`	2462019
`bbox`	[[-54.93, -179.98], [90.0, 180.0]]
`children`	[[*(no name)*, [[-51.8, -179.98],[90.0, 179.75]]], [*(no name)*, [[-54.93,-160.20], [70.96, 178.87]]], [*(no name)*, [[-47.05, 66.95], [36.4833333, 180.0]]]]
`priority_items`	[[*Shanghai*, [[31.22, 121.45],[31.22,121.45]], 0.992], [*Mumbai*, [[18.97,72.82], [18.97, 72.82]],0.991], [*Seoul*, [[37.56, 127.00],[37.56, 127.00]], 0.990], [*Moscow*, [[55.75, 37.61], [55.75,37.61]], 0.990], [*Manila*, [[14.60,120.98], [14.60,120.98]], 0.989]

Another simple way to optimize query performance is to annotate child pointers with an extra property listing the maximum importance or priority score of any item that can be reached via that node. This is called *Maximum Importance Annotation*, and is used to prune searches when the Geobrowser interface specifies that it will only display information above a certain priority or importance score.

6 Reference Dataset and Performance Tests

The development work described in this paper was carried out by MAYA Design as part of the Information Commons and Geobrowser projects. The immediate purpose of this effort was to create an index with good query performance to enable Geobrowser users to access to find places from the Information Commons

[2] Note that adding priority items to higher nodes is different from the "prioritized R-tree" data structure of [33], which optimizes worst-case search by adding prioritized *nodes*, rather than optimizing simple cases by adding prioritized *items*.

Gazetteer [34], a public resource of populated places and worldwide administrative divisions, containing fused information about over 5 million populated places and their political affiliations. The bounding boxes in the R-tree index for this dataset are shown in Figure 4, which gives an interesting statistical picture of world population distribution.

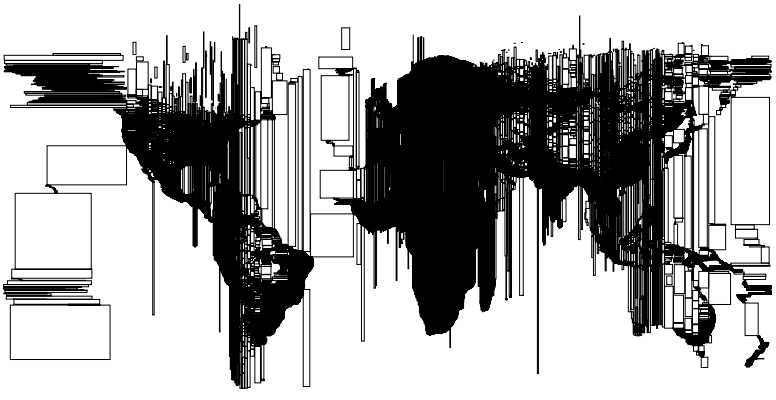


Fig. 4. Leaf-nodes in the R-tree representing World Populated Places

Because the path through the index is replicated as well as the results of the query, subsequent queries can make use of relevant portions of the index. Table 2 shows how some simple queries are affected by this caching property. For each test condition, we measured the time cost of performing a query on an empty local venue, thus inducing index replication over the network. We then wiped the local venue of all u-forms, and performed a related query. Without wiping the local venue, we performed the original query again, and measured the speed-up gained by being able to take advantage of cached index structure from the related query. Replication was performed over a wide-area network. The test query was a geo-spatial bounding box. The related query types were:

Repeated The related query was identical to the test query, so all of the index structure could be re-used.

Sub The related query was a bounding box properly containing the test query. Again, all of the index structure could be re-used.

Expanded The related query was a bounding box properly contained in the test query. Only some of the index structure could be re-used.

Adjacent The related query was a bounding box adjacent to the test query. Only some of the index structure could be re-used.

Distant The related query was a bounding box very distant from the test query. Very little of the index structure could be re-used.

When we can re-use a significant portion of the index structure, performance is naturally much better. When we can re-use less, a corresponding drop in performance improvement is seen. When we cannot re-use any significant portion

Table 2. Results summary of R-tree query performance experiments

Related Query Type	Initial Query(s)	Subsequent Query(s)	Speed-Up Factor
Repeated	4.5	0.77	5.8x
Sub	2.8	0.53	5.4x
Expanded	3.6	2.1	1.8x
Adjacent	4.1	1.7	2.4x
Distant	7.7	8.2	0.95x

of the index, we see that natural variance in the testing overwhelms any benefit from caching: the subsequent query for the "distant" condition is actually slightly worse than the original.

Given space constraints, we elide measurements of similar performance improvements for B-tree based structures. The measured performance improvements are more noticeable in real applications for geo-spatial queries. We believe this is because that query locality is more natural in the geo-spatial domain than it is for, e.g., keyword searches.

7 Distributed GIS Data and Recursive Self-indexing

The most novel index structure we have so far built upon the u-forms and shepherds infrastructure supports distributed mapping in the Geobrowser. Not only point data items, but the vector shapes of the base maps themselves, are represented in u-forms and found by following references to new u-forms and requesting these u-forms through the shepherds system.

Naturally, a list of vector points can easily be expressed as an attribute of a u-form, so geometric objects can be represented in u-forms. The design challenge in this domain is to do this in such a way that shapes can be suitably factored and composed to present a persistent and appealing user interface in a peer-to-peer network. Basic requirements of the Geobrowser map interface include:

1. It should be possible to drag out a spatial object into its own frame, add it to a collection, and add comments, just as with other phenomena in Visage interfaces.
2. The user should see a basic outline map of the area of interest as soon as possible.
3. More detailed shape data should be rendered as the shape u-forms become available.
4. The user should be able to zoom in to obtain greater levels of detail in specific areas.

Requirement 1 is supported by representing the shape of each object in a Cartesian coordinate frame that is optimized for the object in question. (For example, with shapes on the earth's surface such as boundaries of landmasses and political subdivisions, the approximate centroid is located, and then the

local east, local north, and outward normal are used as orthonormal $x-$, $y-$ and z-axes.)

Requirements 2, 3 and 4 are met in the following way. A simplified outline shape is calculated using the Douglas-Peucker line simplification algorithm [35]. This outline shape then stores relations to more detailed line segments, along with information saying which points in the main shape should be replaced by the extra detail in the subshapes. An example of this approach to shape rendering is shown in Figure 5, which depicts the Supercontinent (Asian, African and European landmasses) with both the first level (roughest) and second level (slightly more detailed) decomposition. The first level points are contained in a single u-form, whereas the second level points are broken across 10 different u-forms of roughly similar length. In Figure 5, both levels are depicted together for explanatory purposes, though in the actual Geobrowser, the second level data actually replaces the first so that there is a single coastline. The benefit is that the user only needs to shepherd in the u-forms containing more detailed information for the parts of the world that are of interest. Once accessed, these local detail u-forms stay available the user for as long as they are wanted.

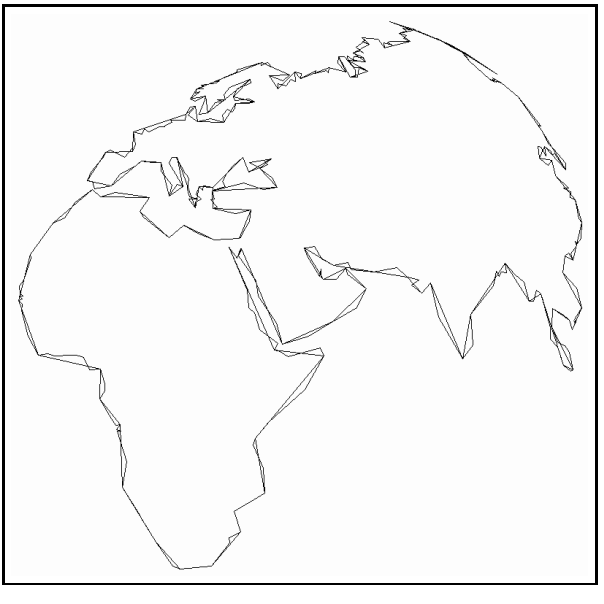


Fig. 5. Shape of the Supercontinent (orthogonal projection), showing first and second levels of detail combined

To support all of this functionality, shapes must be able to include one another recursively. Sometimes shapes are included without replacement (for example, adding an island to a continent in requirement 1), and sometimes subshapes replace parts of their parent shapes (meeting requirement 3). In both cases, a parent shape creates an annotated collection of child shapes, for each child shape giving:

1. The bounding box of the child shape in the parent's coordinate frame; and
2. The linear transformation used to map the child points into the parent frame.

In this way, shape u-forms act as containers for lists of geometric points, and as an index to subsidiary shapes. These subsidiary shapes may be topologically distinct, or may be more detailed parts of the main shape itself. In this way, a shape u-form acts as an index to its own parts, and is described as a *self-indexing* structure.

A rendering algorithm proceeds by:

1. Testing the bounding boxes in turn to see whether the shape intersects with the user's field of view; and
2. Retrieving the child points, and using the linear transformation given to map the points to the parent frame, from which they are mapped to screen coordinates.

These mappings are all calculated using standard linear algebra as used in many computer graphics algorithms (see e.g., [36, §5.6]), and can be optimized significantly by hand-coding the floating point operations for our specific use case. Since all shapes use a form of Maximum Importance Annotation (as described in Section 5.1), we can control the amount of information presented to the user, and a judicious choice of relative priorities and important features enables the Geobrowser to present a high quality user experience with acceptable latency in the interface.

The results are demonstrated in Figure 6, a screenshot from a version of the Geobrowser delivered to the Greater New Orleans Non-Profit Knowledge Works (`www.gnonkw.org`) as part of the New Orleans redevelopment effort following the Hurricane Katrina disaster. This interface actually combines many of the technologies described in this paper. The category tree on the left hand side is modelled using a scalable collection: so by using the interface shown in Figure 2, users can easily add new categories to the map and new items to a category. Each of these scalable collections also has a u-form R-tree index, so the items that intersect with the user's map panel can be efficiently identified. Shapes such as the coastline, the water features, and the roads, are all represented using the recursive u-form shape format described in this section: once an object is returned by the R-tree index, if it has a geographic shape attached, the extended shape of the object can be rendered.

An interesting side effect of our approach to shape simplification and reassembly is that it can reduce the data required and hence the computational complexity of performing a point in polygon test, which is traditionally an $O(n)$ operation, or at best, $O(\log(n))$ for convex polygons [37], where n is the number of vertices in the polygon. Recommended optimizations for point in polygon tests involve (for example) dividing the line segments into quadrants and finding integer versions of the necessary arithmetic [38].

The Douglas-Peucker simplification produces a "tolerance" measure ε along with the simplified shape, where ε is the maximum distance that a point in the original shape is from the simplified shape. Using standard $O(n)$ methods, it is

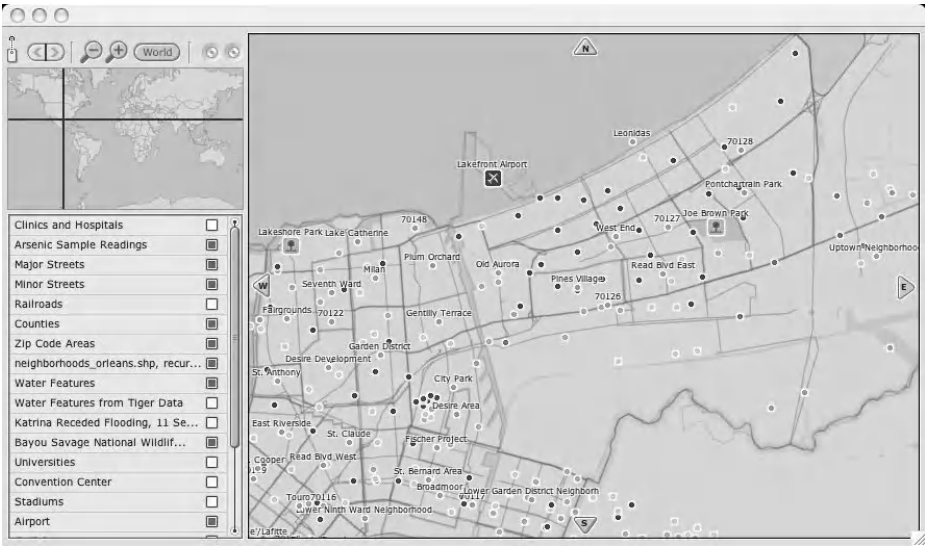


Fig. 6. Map showing part of New Orleans (orthogonal projection), with data such as locations of streets, parks, and levels of arsenic concentration

easy to write an algorithm that (i.) tests whether a test point is inside or outside the simplified polygon, and (ii.) measures the minimum distance from the test point to the simplified polygon. If this distance is greater than ε, then the result also holds for the fully detailed polygon. Otherwise, the u-forms containing more detailed points for the nearby segments are requested and the test is repeated. While this algorithm produces appreciably faster results, formal analysis of this algorithm's complexity is complicated, because it depends to a large extent on the choice of appropriate levels of simplification, which itself depends on number of points, point density, average curvature, and the performance / detail tradeoff.

The more general issue is that logical agents should behave in robust ways given naturally limited resources. Finite resources include computational power and time, and also the data required to solve a problem [39]. It is especially important with distributed information systems to design solutions that can answer a user's questions with the smallest possible bandwidth requirements and the maximum flexibility in terms of *where* computation is performed.

8 Multiple Writers and Keyspace Locality

The shepherdable designs presented in this paper have focused on enabling efficient distributed read access to indexes. It is also necessary to enable many writers to contribute information, which leads to concerns about concurrency and consistency. Managing decentralized contribution and collaboration in u-forms is a large topic that is discussed more fully in [23]. Due to space constraints. this

section presents only summary results that are relevant to the index structures we have discussed.

The simplest way to have many users contributing to a shared index is to have them all write to a common data structure with a single root node. Unfortunately, u-forms in this data structure soon become conflicted, especially if any of the users are offline. This is compounded by the fact that offline users usually only have access to parts of the index that they have read from, and so lack the ability to read another part of the keyspace to decide where in the index to put a new item. UUIDs that are manufactured by concatenating an agreed prefix with a semantically meaningful suffix can help users to put new data in predictable u-forms, but this only compounds the problem that having multiple writers to the same u-form generates conflicts. The generation of conflicts is not an insurmountable barrier, but does lead to some difficult design considerations (see [23]).

One way to solve this problem is to have index writes managed by agents in specific, reliable venues. Requests for items to be added are made by using u-forms and shepherds to mediate an asynchronous messaging system [23, §4.1]. Users should keep their proposed additions in a separate index, which can be flushed when a query to the main shared index demonstrates that their addition has been successful. This in part reduces the problem of managing decentralized index changes to the problem of managing large scale centralized indexes, to which there are known solutions [4]. To distribute load for large indexes, it is sometimes possible to partition the keyspace into different regions covered by different indexing services without greatly affecting the balance of the tree as a whole.

In some cases, the approach of partitioning the index into different parts of the keyspace works particularly well, because there is close correspondence between some parts of the keyspace and the activity of certain user groups. This is particularly likely in geospatial indexes, where users are often focused on a particular region of interest, that corresponds closely to a particular branch or branches of the index. (Compare this with an alphabetic B-tree, where it is unlikely that a user maintains interest in words beginning with a certain string of letters over a long period of use.) Keyspace locality also enhances querying both online and offline, which we believe explains the relative reuse benefit of R-tree searches (Section 6). To gain similar bulk reuse of u-forms for lexical search, topical datasets have to be carefully designed, which can reduce either the scope or the performance of general system-wide searches.

9 Rejected Designs

This section gives a brief summary of designs that we prototyped but eventually rejected because they performed poorly or had undesirable features.

- **Single U-Forms.** Making a single large "index" u-form whose attributes correspond to the terms in the index performs poorly for large vocabularies.
- **UUID concatenation.** One can create a "base UUID" U for the index as a whole, and find the index entry for a particular term T by manufacturing

the UUID $U+T$. This performs well for positive searches, but very badly for out of vocabulary items, because of the 'manufactured null UUIDs' problem (see section 2).

- **Quad tree.** A spatial index structure that used manufactured UUIDs resulting in similar performance problems was the quad tree (see [32, §6.2]).
- **Trie structure.** Digital search trees or trie structures [20, Ch 17] proved to be reasonably effective in our system, though the dependence of query latency on the length of the query terms lead to relatively poor performance compared with the B-tree. Also, the natural support of B-trees for range queries is a benefit for one-dimensional search of continuous data, which is used by one of our community directory websites (`http://goguide.3rc.org/`) for calendar search. A similar design for spatial search (where more general quad tree bounding boxes kept pointers to those of their children that were occupied) was similarly more effective than the regular quad tree, but poor compared with the more balanced R-tree.

10 Conclusion

We have presented a range of shepherdable data structures that support efficient rich-query peer-to-peer search for a variety of interesting applications. We achieve this flexibility by creating search trees and other data structures out of basic database objects, relying on the underlying infrastructure for location, routing and replication. This strategy enables Geobrowser users to repeat previous searches even if they lose network connectivity.

Applications of this technology are not limited to keyword searches, but can be used to create a variety of sophisticated user interfaces such as distributed mapping. At the same time, the careful structuring of information in shepherdable indexes can sometimes encourage developers to construct algorithms that perform more efficiently in a fragile network, by giving results based upon partial data that may nonetheless be sufficient.

Writing to shepherdable indexes is more complex, because naive implementations lead to conflicts, which introduces issues not all of which have yet been solved. To enable reliable decentralized index writes, some combination of service centralization and index partitioning may be necessary.

References

1. Ripeanu, M.: Peer-to-peer architecture case study: Gnutella network. Technical report, University of Chicago (2001)
2. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: Proceedings of ACM ASPLOS, ACM (2000)
3. de Kretser, O., Moffat, A., Shimmin, T., Zobel, J.: Methodologies for distributed information retrieval. In: International Conference on Distributed Computing Systems. (1998) 66–73

4. Melnik, S., Raghavan, S., Yang, B., Garcia-Molina, H.: Building a distributed full-text index for the web. In: World Wide Web. (2001) 396–406
5. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science **2218** (2001) 329
6. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. (2001) 149–160
7. Harvey, N., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: A scalable overlay network with practical locality properties. In: In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03), Seattle, WA (2003)
8. Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R., Wu, J.: Advanced peer-to-peer networking: The P-Grid System and its Applications. PIK Journal - Praxis der Informationsverarbeitung und Kommunikation, Special Issue on P2P Systems (2003)
9. Johnson, T., Colbrook, A.: A distributed data-balanced dictionary based on the b-link tree. In: Proceedings of the 6th International Parallel Processing Symposium, Washington, DC, USA, IEEE Computer Society (1992) 319–324
10. Cosway, P.R.: Replication control in distributed B-trees. Technical Report MIT/LCS/TR-705 (1997)
11. Lindemann, C., Waldhorst, O.: A distributed search service for peer-to-peer file sharing in mobile applications. In: Proc. 2nd IEEE Conf. on Peer-to-Peer Computing (P2P 2002). (2002)
12. Crespo, A., Garcia-Molina, H.: Archival storage for digital libraries. In: Third ACM International Conference on Digital Libraries. (1998)
13. Kroll, B., Widmayer, P.: Distributing a search tree among a growing number of processors. In: SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1994) 265–276
14. Lindemann, C., Waldhorst, O.P.: Exploiting epidemic data dissemination for consistent lookup operations in mobile applications. SIGMOBILE Mob. Comput. Commun. Rev. **8** (2004) 44–56
15. Saito, Y., Shapiro, M.: Optimistic replication. ACM Computing Surveys **37** (2005)
16. Lucas, P., Senn, J., Widdows, D.: Distributed knowledge representation using universal identity and replication. Technical Report MAYA-05007, MAYA Design (2005)
17. Roth, S., Lucas, P., Senn, J., Gomberg, C., Burks, M., Stroffolino, P., Kolojejchick, J., Dunmire, C.: Visage: A user interface environment for exploring information. In: Proceedings of Information Visualization, San Francisco, IEEE (1996) 3–12
18. Viles, C.L., French, J.C.: Dissemination of collection wide information in a distributed information retrieval system. In: SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, ACM Press (1995) 12–20
19. Lucas, P., Widdows, D., Hughes, J., Lucas, W.: Roles in the universal database: Data and metadata in a distributed semantic network. Technical Report MAYA-05009, MAYA Design (2005)
20. Sedgewick, R.: Algorithms in C. Addison-Wesley (1990)
21. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of SIGMOD. (1984) 45–47

22. Lomet, D.B., Salzberg, B.: Concurrency and recovery for index trees. VLDB Journal: Very Large Data Bases **6** (1997) 224–240
23. Higgins, M., Roth, S., Senn, J., Lucas, P., Widdows, D.: Managing distributed collaboration in a peer-to-peer network. 14th International Conference on Cooperative Information Systems (CoopIS 2006) (2006)
24. Manola, F., Miller, E.: RDF primer (2004)
25. Higgins, M., Lucas, P., Senn, J.: VisageWeb: Visualizing WWW Data in Visage. In: Symposium on Information Visualization (Infovis), IEEE (1999) 100–107
26. Project, D.: Command post of the future (CPOF) (2005) `http://www.darpa.mil/ato/programs/CPOF/DT.htm`.
27. Allegheny County Department of Human Services: HumanServices.net (2006) `http://www.humanservices.net/`.
28. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable Peer-To-Peer lookup service for internet applications. In: Proceedings of the 2001 ACM SIGCOMM Conference. (2001) 149–160
29. Li, J., Stribling, J., Morris, R., Kaashoek, M.F.: Bandwidth-efficient management of DHT routing tables. In: Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05), Boston, Massachusetts (2005)
30. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. In: Workshop on Algorithms and Data Structures. (1989) 437–449
31. Balasubramanya, M., Higgins, M., Lucas, P., Senn, J., Widdows, D.: Collaborative annotation that lasts forever: Using peer-to-peer technology for disseminating corpora and language resources. In: Fifth International Conference on Language Resources and Evaluation (LREC 2006), Genoa, Italy (2006)
32. Rigaux, P., Scholl, M., Voisard, A.: Spatial Databases. Morgan Kauffmann / Academic Press (2002)
33. Arge, L., de Berg, M., Haverkort, H., Yi, K.: The priority r-tree: A practically efficient and worst-case optimal r-tree. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04), Paris, France (2004) 347–358
34. Lucas, P., Balasubramanya, M., Widdows, D., Higgins, M.: The Information Commons Gazetteer: A public resource of populated places and worldwide administrative divisions. In: Fifth International Conference on Language Resources and Evaluation (LREC 2006), Genoa, Italy (2006)
35. Douglas, D.H., Peucker, T.K.: Algorithms for the reduction of the number of points required to represent a line or its caricature. The Canadian Cartographer **10** (1973) 112–122
36. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics. Addison Wesley (1990)
37. Haines, E.: Point in polygon strategies. In Heckbert, P., ed.: Graphics Gems. Volume IV. Academic Press (1994) 24–46
38. Hormann, K., Agathos, A.: The point in polygon problem for arbitrary polygons. Computational Geometry **20** (2001) 131–144
39. Gabbay, D.M., Woods, J.: The New Logic. Journal of the Interest Group in Pure and Applied Logics **9 (2)** (2001) 157–190

Distributed Abstract Data Types

Gian Pietro Picco[1], Matteo Migliavacca[1],
Amy L. Murphy[2], and Gruia-Catalin Roman[3]

[1] Dip. di Elettronica e Informazione, Politecnico di Milano, Italy
{picco, migliava}@elet.polimi.it
[2] Faculty of Informatics, University of Lugano, Switzerland
amy.murphy@unisi.ch
[3] Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, USA
roman@wustl.edu

Abstract. In this paper we introduce the concept of *Distributed Abstract Data Type* (DADT), a new programming model simplifying the development of distributed, context-aware applications. A DADT instance logically encapsulates a collection of ADT instances distributed throughout the system. DADT operations specify the semantics of access to this distributed state by using dedicated programming constructs. The scope of these operations can be restricted using DADT *views*, i.e., projections over the target ADT instances, defined declaratively and dynamically based on ADT properties. Interestingly, DADT constructs can be used to deal not only with application data, but also with the space where it resides. This leads to a uniform treatment of the data and space concerns, simplifying the development of context-aware applications and providing the programmer with considerable flexibility and expressive power. We argue that DADTs are amenable to incorporation in existing object-oriented programming languages, as demonstrated by our prototype implementation.

1 Introduction

Modern distributed computing places new demands on application programmers, not only due to the increasing scale, decentralization, and dynamicity, but also because of new requirements about visibility and control over the physical space where the application executes. Examples include scenarios that fall under the umbrella of pervasive and ubiquitous computing, and ambient intelligence. Here, programmers must deal with simultaneous access to a plethora of devices, sensors, or application objects dispersed in the environment to collect the application and contextual data determining the program behavior. However, despite the popularity of the field, available models and systems often treat the physical space—or context—where the application executes as external, introducing specialized constructs and increasing the programming effort. Similarly, abstractions for dealing with distribution are often quite primitive, forcing the programmer to deal explicitly with individual remote accesses.

The model described here provides constructs simplifying the access to distributed state while explicitly taking into account spatial concerns. We accomplish this by extending the well-established notion of abstract data type (ADT) into a *distributed abstract data type* (DADT). A DADT instance logically encapsulates a collection of ADT

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1594–1612, 2006.
© Springer-Verlag Berlin Heidelberg 2006

```
datatype Sensor extends Object {
  data:
       int sensorType;
      bool isActive;
    double value, resolution;
  operations:
    double read();
      void reset();
}
```

Fig. 1. ADT interface for a simple sensor

instances distributed throughout the system. This distributed state can be manipulated through the operations in the DADT interface, whose distributed behavior is defined by the DADT programmer with appropriate constructs. Moreover, the application programmer can dynamically restrict the scope of DADT operations by defining, in a declarative way, projections over the distributed state, called *views*. The definition of DADTs and views, differently from conventional models and languages, is used also to represent space. The programmer can model the notion of space best suited for the application with a DADT, and use its properties to define views over space (e.g., proximity). Our model fully integrates the spatial and data concerns involved in context definition under a single, unified framework centered on the notion of DADT. Data and space become two distinct, and yet intimately related, perspectives enabling distributed manipulation of application entities.

The paper is structured as follows. Section 2 introduces a simple and yet realistic example, used throughout the paper for illustration purposes. Section 3 presents the basic concepts of our DADT model. Section 4 discusses the constructs enabling distributed access, while Section 5 illustrates the aforementioned notion of view. Section 6 reports about the design and implementation of our proof-of-concept prototype. Section 7 places DADTs in the context of related work and Section 8 ends the paper with brief concluding remarks.

2 A Motivating Example

This section introduces an example used throughout the paper to concretely illustrate the main concepts of the model. Consider an environment where several sensors are deployed to report about physical parameters (e.g., temperature, light). A laptop monitoring station needs to collect average temperature readings in its proximity, as well as to manipulate the configuration of the sensor nodes (e.g., reset them or change their transmission range).

In a conventional approach, interaction with sensor data can be modeled with an ADT, as shown in Figure 1. However, this enables only local access to a single sensor (i.e., `Sensor` instance) at a time. To build the functionality above, requiring computation of a global property, the programmer must deal with individual, remote access to sensors, and determine (and keep track of) which are relevant to the computation (e.g., only nearby temperature ones). Clearly, a new ADT (say, `SensorGroupProxy`) can be created to encapsulate these concerns. The point, however, is that *i)* the burden

```
datatype DSensor distributes Sensor with {
  operations:
    void resetAll();
    double average();
}
```

Fig. 2. A data DADT providing access to multiple sensors

of defining the behavior of this latter ADT is entirely on the programmer; *ii)* this is accomplished by mixing the application logic with the management of where data is deployed and how it becomes dynamically accessible through physical space; therefore *iii)* program understandability and maintainability are negatively affected, and so is the possibility of reuse across applications.

The goal of this paper is to simplify the programmer's task through a programming model that *i)* separates the application logic from the management of contextual information; *ii)* simplifies the specification of the distributed computation through composition of elementary, reusable building blocks; *iii)* is easily integrated in mainstream languages. To provide the reader with a concrete glimpse of how this is accomplished, Figure 2 shows the definition of a *distributed abstract data type* (DADT) that embeds the logic for resetting a group of sensors and computing their average reading. The behavior of the operations is specified by the programmer through the constructs we provide, which relieve her from the need to deal explicitly with the details of distribution. Moreover, the execution of the application logic embedded in the operations is decoupled from the specification of the system portion affected. For instance, the computation of the average reading detected by nearby temperature sensors is intuitively achieved by:

```
double v = ds.average() on temperature within proximity;
```

Here, `ds` is an instance of the `DSensor` DADT in Figure 2, while `temperature` and `proximity` are projections over the set of all possible sensors, based on their properties. For instance, the former is defined so that it selects only `Sensor` instances with the proper value of the attribute `sensorType`. The spatial properties related to `proximity` are handled analogously through space ADTs and DADTs, defined later.

In the following we incrementally present the concepts necessary to support this example. Our discussion starts with the basic definitions and connections between data and space ADTs and DADTs. We then move on to the support for defining the internal operations of the DADTs. Finally we bring back the concepts of views to limit the scope of DADT operations. Although each concept is introduced with the language elements supporting it, our focus and main contribution is in the DADT model they support.

3 Basic Concepts

We chose to cast our ideas into the notion of *abstract data type*, as it represents a well-understood programming concept and is general enough to allow us to present our novel constructs without the distractions and idiosyncrasies of a specific programming language or model. While our presentation remains abstract, Section 6 describes an instantiation of the DADT concept for the Java language.

```
spacetype GPSSite extends Site {
  data:
    Location l;
  operations:
    Location getLocation();
}
```

Fig. 3. A space ADT representing a physical location

Data and space ADTs. At the core of our model is the notion of ADT. As mentioned in the example, we draw a sharp line between the data necessary to the application behavior, and the space where such data resides, accordingly distinguishing between data ADTs and space ADTs.

Data ADTs are conventional ADTs encoding the application logic, e.g., `Sensor` in Figure 1. Instead, space ADTs (or *sites*) represent and characterize an abstract notion of the computational environment hosting data ADT instances, e.g., a computer, virtual machine, a PDA, or a car. Many application-dependent notions of space are meaningful. Traditionally, the structure of space is somehow hard-coded in the run-time of the distributed application, and programmers retain only limited—if any—control over it. Instead, in our approach we empower the programmer with the explicit ability to use (and even define) the notion of space that is most appropriate for the application. For instance, mobile applications may make use of network topology, and represent it as a collection of sites with appropriate operations. As such, the notion of site built in our approach is minimalistic, consisting of an ADT `Site` that must be specialized by the programmer. For instance, Figure 3 shows a space ADT whose position is characterized by a physical location. How this location is acquired and defined (e.g., from GPS) is entirely encapsulated in the ADT implementation.

The only syntactic difference between data and space ADTs is the use of the keywords `datatype` and `spacetype`, essentially to enable type checking and improve code readability and understanding.

From ADTs to DADTs. Distributed ADTs specialize the notion of ADT by providing the ability to treat a set of homogeneous ADT instances as a collective unit, accessed through the operations defined in the DADT interface. Returning to our example, the declaration of the `DSensor` DADT in Figure 2 enables distributed access to instances of the ADT `Sensor` defined in Figure 1, through the `resetAll` and `average` operations. Similarly, one can define a space DADT for collective access to a set of sites. For example, later on (in Figure 8) we define a `WirelessNetwork` space DADT that allows collective access to the configuration of network hosts.

The declaration of a DADT is similar to the one of an ADT. This is not surprising, since DADTs are ADTs themselves. The only difference is the presence of the `distributes` relation, which defines a new relationship among ADT types, analogous to inheritance. A `distributes` B, where A is a DADT and B an ADT, states that a set of instances of B[1] can be collectively accessed through the operations defined in

[1] Or any other type compatible with B according to the typing rules of the target language.

A's interface. Clearly if A `distributes` B and A is a space DADT, B must be a space ADT, i.e., it must be a subtype of `Site`.

The application programmer—which in large development efforts is likely to be different from both the ADT and DADT programmers—can manipulate the distributed state exported by ADT instances by creating a DADT instance and invoking its operations. For example,

```
DSensor ds = new DSensor();
double v = ds.average();
```

creates a `DSensor` instance and uses it to trigger the distributed processing encapsulated in its `average` operation, as specified by the DADT programmer with the constructs illustrated in Section 4. Note how the invocation above is indistinguishable from any other on a conventional ADT instance: the application programmer may even be unaware of the distributed nature of the reference `ds`.

The DADT member set. Analogously to ADTs, a DADT may declare attributes, which may assume different values for each of its instances. However, every DADT instance always encapsulates the distributed state constituted by the set of ADT instances available for distributed computation through its operations, as a sort of implicit attribute. This set of ADT instances is called the *member set* of the DADT, and its elements are ADT instances of the type on the right hand side of the `distributes` relation. In principle, the content of the member set is the same across all the DADT instances in the system. Fulfilling this requirement, however, is impractical in any distributed system, as asynchrony and concurrency complicate the task of maintaining globally consistent state. Moreover, different applications may allow different, weaker notions of member set consistency. For this reason, we assume that the underlying run-time provides *best effort* consistency. As a consequence, different DADT instances may have different values of the DADT member set. At the same time, however, as we discuss in Section 6, the flexible architecture of our DADT run-time provides mechanisms to enable customization of the middleware with alternative consistency algorithms.

Binding an ADT into a DADT. Although we defined the notion of member set, we did not discuss how ADT instances become part of it. In our model, this is represented as a binding between each instance of the ADT and the DADT. This can be explicitly managed on a per-instance basis by the application programmer with bind and unbind operations, as in `bind(new Sensor(),"DSensor")`. However, this constitutes also the basic functionality for higher-level automatic mechanisms, e.g., to automatically bind ADTs to DADTs based on information about types and the `distributes` relation.

Note how an ADT instance is bound to a DADT *type*, not to a specific *instance*. In fact, it is the DADT name that serves to identify collectively a group of ADT instances; DADT instances, instead, provide an *entry point* towards this group. Moreover, while the effect of binding (and unbinding) is global, as mentioned previously the model does not guarantee a globally consistent view of the resulting member set, and therefore these entry points may have different values of the member set.

Putting it all together: The interplay of data and space. Figure 4 provides a graphical representation of the concepts discussed thus far and introduces some new ones. The

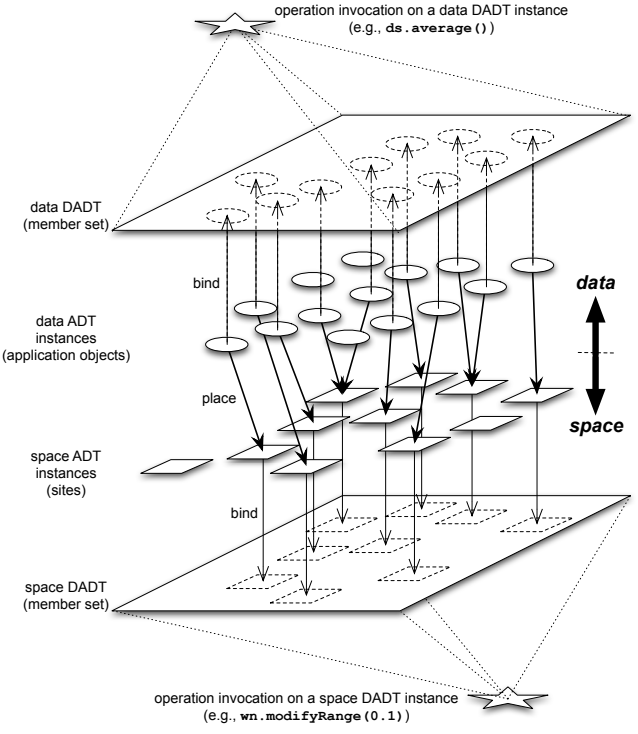


Fig. 4. Data and space in the DADT model

ADT instances in the figure are bound to a DADT and become part of its member set. The latter is represented visually as a plane, onto which ADT instances (the solid circles and squares) are projected as a consequence of binding. Note that some ADT instances are left unbound either explicitly by the application programmer or by some property of the implicit binding. Collective access to the member set is enabled through a DADT instance (represented as a star), which serves as the entry point to the member set.

While this allows access to either data or space, we have not yet discussed how the two are related. Conventional programming languages implicitly bind data to space when an object is created, including it in the local "computational environment". As we abstract the latter into a space ADT instance (site), we need to establish the proper binding between a data ADT instance and the site where it resides. In our model, this is achieved using the `place` operation, represented by the thick lines in Figure 4. It can be invoked explicitly by the programmer by binding data ADT and space ADT instances co-located in the same computational environment. For example, if `g` is a `GPSSite`, then `place(new Sensor(),g)` binds a newly created sensor to the existing site `g`. For applications that do not require this degree of flexibility, a default site can be specified and every newly created data ADT instance is automatically *placed* on it. In either case, the explicit definition of a site allows symmetric treatment of data and space.

```
1  void DSensor::resetAll() {
2    (all in targetset).reset();
3  }
4  double DSensor::average() {
5    double sum = 0;
6    double[] readings = (all in targetset).read();
7    for(int i=0; i<readings.length; i++)
8      sum += readings[i];
9    return sum/readings.length;
10 }
```

Fig. 5. Aggregating sensor data through DADTs

As shown in the figure, multiple ADTs can be placed on the same site. Also, the same ADT can be bound to multiple DADTs; an option that could be represented in the figure by drawing another plane (data or space) for the new DADT. By creating instances of both DADTs, the same ADTs can be accessed through multiple application perspectives at different times. To see why this is useful for sites, consider a host with multiple network interfaces, e.g., Bluetooth and WiFi. This can be supported with two space DADTs, one for each network interface. Hosts supporting both interfaces should bind to both DADTs. For the sake of simplicity the remainder of this paper assumes a site is bound to a single space DADT, however the same assumption is not made for data.

Data and space ADTs can be regarded as the basic elements of the world relevant to the application, that can be manipulated globally through the "knobs" provided by the DADT definition. In doing this, the application programmer is free to choose the perspective (data or space) more appropriate for the functionality at hand. For instance, she can request a global average through a data DADT but modify the communication range of nodes through a space DADT, as shown in Figure 4. The two perspectives are coherently integrated in a single model, as the `bind` and `place` operations effectively connect data and space ADTs among themselves and to the corresponding DADT planes. Moreover, the model is further enriched by the ability to limit the portion of the system affected by this global manipulation through view definition, again based on data and/or space concerns. Before delving in this subject, however, we need to illustrate distributed access to ADTs.

4 Distributed Access to ADTs

After the DADT's interface and target ADT are declared, as shown for instance in Figure 2, the DADT behavior realizing distributed, transparent access must be defined by specifying the body of the DADT methods. Appropriate constructs are needed to access and manipulate the distributed state in the DADT member set. For this, we introduce two programming constructs: *operators* and *actions*.

4.1 Operators

We start by focusing on the task of implementing the `resetAll` method of `DSensor`, whose intended behavior is to reset all the sensors in the system. Since the DADT

operates on a set of ADT instances, one would like to specify the desired behavior by operating on the set in a declarative way. For instance, in a Z-like formal language, with

$$\forall x \mid x \in \mathcal{M} \bullet \texttt{x.reset}()$$

where $\mathcal{M}$ is the member set of `DSensor`. This is expressed in our DADT language as shown in Figure 5, where the expression above is represented by the statement on line 2. In this statement, the invocation target—normally a reference to an ADT instance—is replaced by an expression denoting the set of instances on which the method `reset` is executed. The semantics of execution is such that `reset` gets invoked independently and concurrently on each of the ADT instances belonging to the invocation target. Figure 5 also shows the implementation of the `average` method, where the results of the various invocations are collected and used by the DADT implementation.

Selection vs. condition operators. Next we look more closely at the expression representing the invocation target in line 2 of Figure 5. The variable `targetset`, to which every DADT operation has implicit access, is the set of ADT instances available for distributed processing. At this point of our presentation, the target set always coincides with the member set, however this changes when DADT views are introduced in Section 5. The keyword `in` plays the role of the mathematical membership operator $\in$. Finally, the operator `all` allows one to extract a collective reference to the instances in the target set.

Other operators also make sense. The dual operator `any`, for example, is such that the effect of

```
(any in targetset).reset();
```

is to reset one among the sensors in the target set, chosen non-deterministically. Both `all` and `any` are *selection operators* or, *selectors*, because they allow selection of a subset of the instances contained in the target set.

Selectors essentially enable the programmer to specify declaratively a reference to a distributed invocation target constituted by multiple actual ADT instances. Interestingly, this is achieved transparently, i.e., the programmer does not require any knowledge of the actual *identity* of the instances. In addition, we provide *condition operators*, which can be used to make the code of a DADT method dependent on a global condition over the target set. The operator `in?` tests whether one set is contained in another, while # returns the number of elements currently in it. With reference to Figure 5, it is possible to rewrite `resetAll` so that it resets all the sensors only if a given "master" sensor (whose identifier we assume known) is not available, and rewrite `average` to compute the average only if more than 3 sensors are around:

```
if (!({master} in? targetset)) (all in targetset).reset();
if ((# targetset)>3) readings = (all in targetset).read();
```

We support also other operators, e.g., to explicitly access a set of ADT instances based on their identifiers as in `({a1,a2,a10} in targetset).read()`, and a notion of *iterator*. The latter enables the programmer to access the ADT instances one at a time rather than interacting with all of them at once, which in some cases may reduce communication overhead. Due to space limitations we redirect the interested reader to [10].

```
1  double DSensor::average() {
2    action double reliableRead() {
3      double reading; int tries = 3;
4      while (tries > 0) {
5        reading = local.read();
6        if (reading == ERROR) --tries;
7        else break;
8      }
9      if (reading == ERROR) {
10       local.reset();
11       reading = local.read();
12     }
13   }
14   double[] readings=(all in targetset).reliableRead();
15   bool found = false; int i = 0;
16   while (!found && i < readings.length)
17     found = (readings[i++] == ERROR);
18   if (!found) {
19     double sum = 0;
20     for (int i=0; i<readings.length; i++)
21       sum += readings[i];
22     return sum/readings.length;
23   } else /** report fault to the application **/;
24 }
```

Fig. 6. Accessing remote ADTs using actions

Other operators beyond those discussed here can be defined. Examples include a variant of `any` that non-deterministically selects a given number of instances (e.g., `any(4)`), or selectors relying on contextual information (e.g., a `nearest` operator that returns the geographically closest instance). Our current prototype provides a built-in implementation for the operators described thus far, as well as the required mechanisms to enable the programmer to define her own, as illustrated in Section 6.

4.2 Actions

The use of operators enables concurrent access to remote ADT instances and thus far we have assumed that such access occurs by invoking one of the ADT's operations. However, in many cases, this is not sufficient. For instance, assume the ADT's `read` operation is capable of signaling a malfunction by returning an `ERROR` value. In this case, it may be reasonable to circumvent transient faults (e.g., due to interference of the sensor with physical phenomenons) with simple countermeasures, e.g., retry the read operation a number of times, after which the sensor is reset and the read repeated again. If also this last attempt fails, the fault is reported.

To implement this behavior we could leverage off the operators presented, by attempting a distributed `read` operation (e.g., using the `all` operator as in line 6 of Figure 5) and then dealing with faulty nodes. However, this may be inefficient. If the `all` operator is used again to retry the reads and possibly reset sensors, its effect is global instead of being limited to faulty sensors. On the other hand, if faulty sensors are singled out by using the aforementioned ability to access explicitly a given sensor based on identifier, a great deal of communication is generated because retries and reset operations need be invoked from the (remote) DADT instance. Both solutions are inherently

unsatisfactory because they ignore a fundamental point: the sequence of failing `read` and corrective `reset` operations do not require intervention of the DADT instance and instead can be controlled local to the ADT instance. In other words, a distinction is necessary between the application logic that determines how to recover from a fault (which is entirely local to a sensor) and its distribution across the system.

This separation can be achieved elegantly and efficiently with the notion of *action*. An action is an operation that is defined in the DADT but whose execution occurs local to the ADT instance on which it is invoked without requiring additional communication. Loosely speaking, actions enable the programmer to write DADT code that operates on ADT instances as if they were exporting a richer interface.

Figure 6 illustrates the concept. The action declaration is contained in lines 2–13, and is identical to the one for a standard programming language routine, prepended by the keyword `action`. The only difference is the use of the keyword `local`, which is bound at runtime to the ADT instance on which the action is currently being evaluated. Note how `local` is different from the traditional `this` keyword, pointing in this case to the DADT instance on which the operation containing the action (`average`) is being invoked. This should not be surprising, given that although the action *definition* belongs to the DADT and its execution is triggered through one of the DADT operations, the action *execution* is entirely local to the sensor, as if it were just another operation of the ADT. These semantics can be visualized by considering actions as mobile code [4] being shipped dynamically and remotely evaluated on the ADT instances. However, mobile code is only one of the options available for their distributed execution.

The action code in Figure 6 performs the local read and, if an error is reported, retries the read and possibly resets the sensor. The `average` DADT operation exploits this action definition by simply invoking the action over the sensors in the target set (line 14) using the notation described in Section 4.1. The remainder of the operation scans the obtained readings for error codes (returned by sensors with a persistent fault) and either computes and returns the average as in Figure 5 or reports the error to the application.

5 Restricting the Scope of Operations

Distributed sharing of ADTs as we have defined it thus far is a powerful concept. However, the invocation of an operation over *all* the instances in the member set may not always be desirable. For example, performance reasons may render it impractical or application needs may suggest policies to constrain an operation to a subset of the ADT instances.

Our reference example helps to clarify these concepts. In our scenario, different kinds of sensors (e.g., temperature, light, humidity) may be present, accounted for in Figure 1 with the attribute `sensorType`. As discussed so far, the sharing rules include all instances of a given ADT, therefore including multiple types of sensors in the same `average` computation, yielding a meaningless result. Instead, we would like to select a subset of the available sensors, namely those of a given kind. One could argue that the sensor kind could (or should) be encoded as a different ADT, e.g., inheriting from an abstract sensor. Unfortunately, subtyping is not possible for other characteristics of

```
1  datatype DSensor distributes Sensor with {
2    properties:
3      bool isActive();
4      bool isSensorType(int sensorType);
5      bool isPrecise(double resolution);
6    actions:
7      double[] reliableRead();
8    operations:
9      double average();
10     void resetAll();
11 }
12 bool DSensor::isActive() {
13   return local.isActive;
14 }
15 bool DSensor::isSensorType(int sensorType) {
16   return local.sensorType == sensorType;
17 }
18 bool DSensor::isPrecise(double resolution) {
19   return local.resolution >= resolution;
20 }
21 void DSensor::resetAll()  {/* ...as in Figure 5... */}
22 double DSensor::average() {/* ...as in Figure 5... */}
```

Fig. 7. Restricting the scope of operations over data

a sensor, such as those representing a portion its state, as opposed to a static characteristic like the sensor type. For instance, the laptop monitoring station in our example may need to reset all the sensors currently inactive or to compute the average over only sensors guaranteeing a given resolution. For these cases one would like to specify something such as

$$\forall x \mid \neg(x \text{ is active}) \wedge \ x \in \mathcal{M} \bullet x.\texttt{reset}()$$

We provide this level of flexibility and expressiveness in DADTs through the notions of property and view. A *property* is a characteristic of a DADT defined in terms of an ADT's data and operations, and evaluated local to an ADT instance. In programming terms, properties are specified as part of the DADT interface as boolean operations. Figure 7 shows the `DSensor` defined earlier augmented with properties. For instance, `isActive` returns true if the corresponding local attribute (see Figure 1) on the `Sensor` instance where the property is being evaluated is true as well. Similarly, `isSensorType` and `isPrecise` return true if the target sensor is of the kind specified or provides a sufficient resolution, respectively. The definition of these properties relies on the `local` keyword to access the ADT instance they are currently being evaluated upon, similarly to actions in Section 4.2. Indeed, like actions, properties are defined on the DADT but evaluated on (remote) ADTs.

This simple concept enables the definition of dynamic projections over the member set, called DADT *views*. Views are defined by the DADT programmer using properties and dedicated constructs. For instance,

```
dataview active on DSensor as isActive();
```

defines the view `active` as the subset of the member set of `DSensor` that contains only those `Sensor` instances for which the evaluation of the property `isActive` yields true. The DADT name refers implicitly to its member set.

Properties can have parameters. For instance, the view containing all temperature sensors can be defined as:

```
dataview temperature on DSensor as isSensorType(TEMP);
```

After a view is defined, it can be used to restrict dynamically the scope of a DADT operation. For instance,

```
ds.resetAll() on !active;
```

resets all sensors that are currently inactive. The execution semantics is such that the `targetset` in line 2 of Figure 5, which in Section 4.1 was bound to the member set, is here bound at invocation time to the identifiers of the ADT instances belonging to the view. Of course, the resulting view may be empty, i.e., no instance satisfies the view definition. In this case no operation is performed.

Just as we used negation in the above formula, we support the general ability to compose views by connecting properties with boolean operators. While the mechanics are simple, the feature itself is powerful as it allows one to express views using union and intersection. For instance,

```
dataview preciseOn on DSensor as isPrecise(0.1) && isActive();
```

captures the subset of sensors that are active and provide a resolution greater than 10%.

Additional features of our model enable the definition of *asymmetric* views. The content (i.e., ADT instances) of these views, unlike those defined thus far, depends on attributes of the specific DADT instance on which the operation restricted by the view is invoked. Although this feature increases the expressiveness of view definition, space limitations force us to refer the reader to [10] for further details.

If no view is specified at invocation time, the operation is performed on the whole member set, as discussed in Section 4.1. Indeed, the member set defines the most general view containing all ADT instances bound to the DADT. Moreover, all the constructs we discussed in Section 4 can be used with views, as these are ultimately sets of ADT instances. This holds not only *inside* the definition of an operation (as specified by the DADT programmer), but also *outside* the DADT definition (as specified by the application programmer). For instance, the following program fragment reliably retrieves the values sensed by all the active temperature sensors using the action whose code[2] appeared in Figure 6, provided that at least three of them provide sufficient resolution:

```
if ((# (preciseOn && temperature)) >= 3)
  readings = (all in (active && temperature)).reliableRead();
```

All the considerations made thus far hold both for *data* and *space* views. For example, in our motivating scenario, the laptop requests an operation to be executed over nodes *within proximity*. This is accomplished in a two-step process, first defining the site, then the view. Figure 8 shows the definition of the `WirelessNetwork` space DADT. It defines a property `isReachable` that yields true if the target site is within a specified number of hops. A space view can represent proximity based on this property. This view can be used to restrain not only operations on a data DADT, but also on a space DADT, e.g., to reduce by 10% the communication range of nearby hosts as in:

[2] In Figure 7 the action is declared in the interface of `DSensor`, contrary to Figure 6, and is therefore visible to and can be reused by any of the DADT operations and by the client objects calling them. The action code is encapsulated in the DADT, thus providing information hiding.

```
spacetype WirelessNetwork distributes WirelessSite with {
  properties:
    bool isReachable(int hops);
  operations:
    void modifyRange(double percent);
}
```

Fig. 8. A space DADT providing distributed access to sites

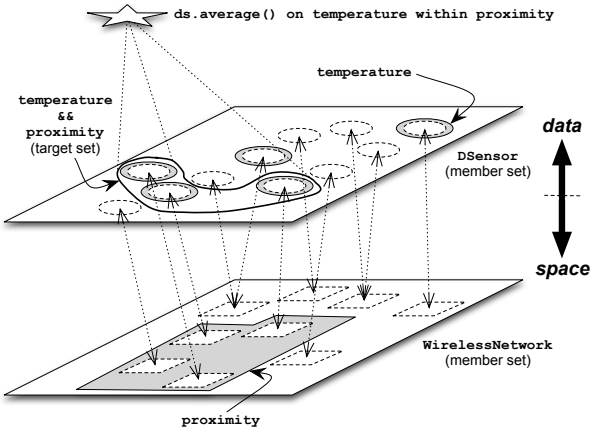


Fig. 9. DADT views

```
spaceview proximity on WirelessNetwork as isReachable(2);
wn.modifyRange(-0.1) within proximity;
```

where wn is an instance of the WirelessNetwork DADT. Note how proximity is asymmetric, as it depends on the location of the DADT instance on which it is invoked.

One of the most powerful uses of views arises in the combination of data and space views, as in:

```
double v = ds.average() on temperature within proximity;
wn.modifyRange(-0.1) on temperature within proximity;
```

which combine the temperature data view with the proximity space view to return either the proximal average temperature with an operation over data, or reduce the communication range of proximal temperature sensors with an operation over space. The content of targetset inside the body of the invoked (data or space) DADT operation is computed as the overlap of the subsets defined by the two views. Figure 9 illustrates the concept w.r.t. the first of the two statements above. Different from Figure 4, data and space ADT instances are not shown—only their "projection" on the member sets and the place relation between data ADTs (circles) and sites (squares).

As mentioned in Section 3, our DADT model provides a unified representation of data and space, where they are simply two different perspectives for accessing and manipulating the application ADTs. The notion of DADT provides the mechanism for

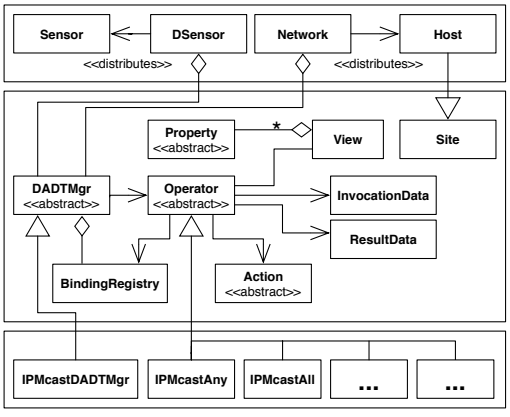


Fig. 10. The architecture of the $\mathcal{J}$DADT run-time

defining the application behavior manipulating the distributed state, as well as the properties that can be used to define views. DADT instances are instead the dynamic entities through which distributed access is effected, and whose scope is restricted dynamically by means of views. The programmer is free to decide what is the best "vantage point" (data or space) for accessing the distributed system. In our model, data and space become easily interchanged during the programming practice, with our model coherently maintaining their semantic interaction.

6 Prototype Implementation

To verify the feasibility of the DADT model we developed a proof-of-concept prototype, providing the DADT constructs described thus far in the context of the Java language. The prototype, currently nicknamed $\mathcal{J}$DADT, is divided into two parts: a translator and a run-time. The *translator* takes care of translating a Java program augmented with statements from our DADT language into a conventional Java program, through a precompilation step. The code generated by the translator implements the DADT constructs by using the classes defined in the *run-time* library. Once the translation is generated, the resulting code can be directly executed on any Java virtual machine where the run-time packages are deployed.

The translator for the source-to-source transformation is implemented using the ANTLR [1] parser generator. The source grammar is a modification of the Java 1.5 grammar by Studman [12] with extensions for DADT constructs. When launched, the ANTLR generator builds the Abstract Syntax Tree (AST) with custom nodes for DADT constructs which are next modified by *tree walkers* to reconstruct a plain Java AST. This AST is emitted with a standard ANTLR Java emitter into code that contains invocations to the run-time as detailed next.

The run-time architecture is composed of several components, the main classes of which are outlined in Figure 10. The top layer is composed of application classes, like

those we used in our example. While the definition of data ADTs and DADTs is entirely up to the programmer, our implementation provides built-in notions of `Host` and `Network`. The layer below constitutes the API of our DADT run-time used by the translator, to map the DADT constructs into the appropriate run-time objects and invocations. The class `DADTMgr` provides the methods handling the binding of ADTs to DADTs, to specify the `Site` instance abstracting the *node*[3] where the run-time executes, as well as other auxiliary methods managing configuration aspects of the run-time. There is only one `DADTMgr` instance per node. The `BindingRegistry` object associated to the `DADTMgr` contains information about which node's ADT is bound to which DADT, provides methods to determine the local ADT instances that satisfy a given view specification, and contains information about the available operators.

The abstract classes `Property` and `Action` represent the corresponding concepts. Both represent DADT methods whose execution takes place on a (remote) ADT. To understand the translation strategy, let us focus on the `isPrecise` property shown in Figure 7. The translator generates the corresponding class:

```
class isPrecise_Property extends Property {
  double resolution;
  isPrecise_Property(double resolution) { this.resolution = resolution; }
  boolean evaluate(Object o) {
    Sensor local = (Sensor) o;
    return local.resolution >= resolution;
  }
}
```

where the property parameter is now a class attribute, and the property body is contained in the `evaluate` method. The latter accepts as a parameter an instance of the ADT (`Sensor` in our case) distributed by the DADT. This can be safely substituted for the keyword `local` used in Figure 7. A similar strategy is used for translating actions. `Property` and `Action` objects are created upon action invocation or view definition, serialized, and transmitted on the node of the target ADT instance. There, their `evaluate` method is executed by using this ADT instance as a parameter, to enable invocation of its operations. Note that here we assume that the DADT class is pre-deployed on the interested nodes. A more dynamic and open design is easily obtained with a "code on demand" paradigm [4], enabling class relocation only if and when needed.

A view (on data, space, or both) is represented by a `View` object containing the set of `Property` objects involved in the view definition, properly arranged in an abstract tree representing the view predicate. To determine whether a given ADT instance belongs to the view, the abstract tree is navigated from the leaves (the property objects) to the top (the view object), by invoking the `evaluate` method of each property and composing it with others according to the boolean operator in each abstract tree node. The process terminates at the root with the boolean result of the evaluation. Composition of views is easily achieved by composing their predicates, i.e., their abstract trees. Finally, a view object's `apply` method allows execution of an action on the ADT instances selected by an operator.

[3] The term *node* here refers to the computational environment where the run-time executes—a JVM in our implementation. This is not be confused with a *site*, which is a node's abstract representation as a space ADT.

Operators, both built-in and programmer-defined, are subclasses of `Operator`. The corresponding implementation is retrieved at run-time from `DADTMgr`, using a Factory pattern. The operator behavior is specified by overriding only two methods. `evalRemote(View v)` is invoked on the caller node (e.g., where `ds.average` is called), and embeds the logic for distributing the information necessary to the collective evaluation of the operator on the view, and the retrieval of the results. Instead, `evalLocal(InvocationData d)` is performed on all the other nodes involved in the computation (i.e., those hosting the target ADT instances), and contains the logic for evaluating the operator local to a single node and sending the results back to the caller. The parameter contains the information representing an invocation, i.e., the view specification, the operator to be applied, the action to be executed, and the caller's identifier. Both methods above return results packed in a simple `ResultData` class. In essence, `Operator` subclasses embed all the distribution logic connecting the caller node and the ADT instances. In our prototype, the implementation of the `all` operator leverages off UDP unicast and multicast sockets, as shown in the bottom layer of Figure 10 containing the classes specializing our framework. Multicast is exploited for dispatching invocations to the ADT instances bound to a given DADT, while UDP is used for returning results. More details in [10].

To clarify how the various components cooperate, consider the following sample DADT program statements

```
dataview preciseOn on DSensor
                  as isPrecise(0.1) && isActive();
spaceview proximity on Network as isReachable(2);
ds.average() on preciseOn within proximity;
```

using the fault-tolerant definition of `average` from Figure 6. The translator generates the `Property` subclasses based on the definition of the two views, creates the corresponding objects, and inserts them in the abstract tree, either directly or by invoking the methods (`and`, `or`) for logically composing properties:

```
View preciseOn = new View(new isPrecise_Property(0.1)
                      .and(new isActive_Property()));
View proximity = new View(new isReachable_Property(2));
```

Moreover, it transforms the signature of `average` into

```
double average(View targetset)
```

In our case the target set is the conjunction of `preciseOn` and `proximity`, and is passed upon method invocation:

```
ds.average(new View(preciseOn).and(proximity));
```

Note how a new `View` instance is generated on the fly to represent the clause `on...within...` of the invocation by merging the data view and the space view. In our implementation, both are represented by `View` objects, which are then composed like properties. Different constructors are provided to create a view out of its properties or based on already existing views. Moreover, the translator generates an `Action` subclass representing the action `reliableRead` in the figure, as we described earlier. Line 14 of Figure 6, containing the action invocation in conjunction with the `all` operator, becomes

```
double[] readings = (double[]) view.apply("all",new reliableRead_Action());
```

The body of `apply` retrieves from the node's `DADTMgr` the proper implementation of the operator based on the name passed as a parameter, and starts its distributed execution on the bound ADTs, as shown in the following excerpt

```
Object apply(String name, Action a) { ...
  Operator op = DADTMgr.getOperator(name);
  ResultData d = op.evalRemote(
    new InvocationData(this,name,a,caller));
... }
```

where `this` is the `View` instance requesting the invocation and `caller` is the identifier of the corresponding node.

The design just described is conceived as a *framework* in the object-oriented sense, and can be customized (e.g., to use different communication protocols) by redefining only two classes, i.e., `Operator` and `DADTMgr`. Moreover, although $\mathcal{J}$DADT is a proof-of-concept prototype, its realization is still non-trivial. It is fully decentralized and, although thus far we have tested it only in a fixed environment, its reliance only on the most basic network facilities leaves open the opportunity for a seamless migration to more dynamic ones, using the appropriate network routing protocols (e.g., for MANET). Nevertheless, it can clearly be improved in many respects. Most notably, we are currently studying distributed algorithms for efficiently managing the distributed dissemination of actions and results, and for maintaining views. In doing this, we are supported by our previous work on data sharing middleware for mobile computing (e.g., LIME [8] and EgoSpaces [7]), for which DADTs actually provide a generalization.

7 Related Work

The closest works come from the context of parallel systems. Shared ADTs (SADTs) [5] focuses on providing implementations of several standard data types, whose implementation is designed to scale well in the parallel environment. Concurrent Aggregates (CAs) [3] provides language-level support for defining both the ADT interface and the implementation of its distributed components. Each component is defined in terms of message processing and is explicitly enabled to send messages to fellow components, creating aggregate behavior. In comparison to these systems, not only do DADTs target the more general distributed setting, but they also provide a unique and uniform treatment of data and space, as well as the increased flexibility coming from the view concept. In the other systems, not only is the view the same at all times, but the components contributing to it cannot change during execution.

Recently, the Spatial Programming model [2] has been proposed for embedded systems. It makes an analogy between space and memory and exposes space to applications through spatial references, which enable the definition of regions including components interested in a given computation. Similar ideas have also been applied to mobile ad hoc networks in the SpatialViews language [9], where *virtual networks* can be defined depending on the physical location of a node and the services (expressed as Java interfaces) it provides. Common to our work is the notion of scoping virtual networks provide. However, DADTs provide a clean separation between data and space and the notion of data views, not supported by the aforementioned approaches, allows us to

limit execution not only based on spatial constraints, but also on application ones, with the two nicely integrated in our model. Finally, unlike our approach, computation is distributed across nodes in a virtual network by migrating mobile agents from node to node, and this is an integral part of the model. Instead, in our model we provide simultaneous access to multiple instances through a single DADT reference, and mobile code is only one of the implementation options.

Some aspects of our work may look reminiscent of distributed object platforms (e.g., CORBA or Emerald [6]). However, these platforms rely on remote method invocation as a means to access explicitly identified, single object instances. Instead, in our DADT model the identity of remote objects remains hidden (unless explicitly needed), enabling transparent access. Moreover, collective access occurs through a single DADT interface, while the same aggregation would require extensive programming effort in a distributed object system. Finally, distributed object systems hide the object location in references, while DADTs foster a clean separation between data and space, hiding location when dealing with data, but enabling its (direct or indirect) access when dealing with space.

Finally, the idea of sets as a programming abstraction was pioneered in the early 70's (e.g., [11]). However, the aim and results of these approaches are profoundly different from ours. Indeed, their goal was to use sets as the fundamental abstraction for *all* programming tasks. Instead, here we use a (much less powerful) set abstraction only to deal with distribution, and rely on standard imperative programming constructs for the dealing with the other concerns.

8 Conclusion

Developing distributed applications is a complex task, especially when the physical space must be taken into account, as in the case of context-aware computing. In this paper, we proposed DADTs as a novel distributed programming model enabling collective access to data and space entities by means of operations whose distributed behavior is encapsulated in the DADT using dedicated constructs, and whose invocation scope can be dynamically defined based on application and contextual information. We conjecture that the unified treatment of data and space concerns inside the model, together with our choice to embed these features in a well-known and widely used programming technique, is likely to improve programming practices in modern distributed and context-aware computing.

Acknowledgments. The authors wish to thank Domenico Bianculli for his work on the implementation of the translator. The work described in this paper was partially supported by the European Union under the project IST-004356-RUNES and by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project. Roman was supported in part by the US Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. ANTLR Web page. `www.antlr.org`.
2. C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS)*, March 2004.
3. A.A. Chien and W.J. Dally. Concurrent Aggregates (CA). In *Proc. of the Symp. on Principles & practice of parallel programming*, pages 187–196, 1990.
4. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Soft. Eng.*, 24(5), 1998.
5. D.M. Goodeve et al. Toward a model for shared data abstraction with performance. *J. of Parallel and Distributed Computing*, 49(1):156–167, 1998.
6. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(2):109–133, February 1988.
7. C. Julien and G.-C. Roman. Active Coordination in Ad Hoc Networks. In *Proc. of COORDINATION*, 2004.
8. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the Int. Conf. on Software Engineering*, pages 368–377, May 1999.
9. Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, 2005.
10. G.P. Picco, M. Migliavacca, A.L. Murphy, and G.-C. Roman. Distributed Abstract Data Types. Technical report, Politecnico di Milano, 2004. Available at `www.elet.polimi.it/upload/~picco`.
11. J. T. Schwartz et al. *Programming with sets; an introduction to SETL*. Springer, 1986.
12. M. Studman et al. Java 1.5 Grammar. `www.antlr.org/grammar/1109874324096/java1.5.zip`.

Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services

Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway
{kraemer, herrmann, rolv.braek}@item.ntnu.no

Abstract. In our service engineering approach, services are specified by UML 2.0 collaborations and activities, focusing on the interactions between cooperating entities. To execute services, however, we need precise behavioral descriptions of physical system components modeling how a component contributes to a service. For these descriptions we use the concept of state machines which form a suitable input for our existing code generators that produce efficiently executable programs. From the engineering viewpoint, the gap between the collaborations and the components will be covered by UML model transformations. To ensure the correctness of these transformations, we use the compositional Temporal Logic of Actions (cTLA) which enables us to reason about service specifications and their refinement formally. In this paper, we focus on the execution of services. By outlining an UML profile, we describe which form the descriptions of the components should have to be efficiently executable. To guarantee the correctness of the design process, we further introduce the cTLA specification style cTLA/e which is behaviorally equivalent with the UML 2.0 state machines used as code generator input. In this way, we bridge the gap between UML for modeling and design, cTLA specifications used for reasoning, and the efficient execution of services, so that we can prove important properties formally.

1 Introduction

The ongoing convergence of the communication and the computing domain enables a wide range of advanced services, involving a complex mixture of technologies, devices and networks. The development has reached a degree of complexity in which formal reasoning about specifications and corresponding tool support are increasingly important to design services of high quality within acceptable time and cost limits. In consequence, service engineering has become a discipline in its own right. In earlier publications we demonstrated the close conceptual relationship between services and collaborations, and the suitability of collaborations as a framework for service specifications [1,2,3]. Collaborations model cross-cutting, partial behavior involving several participants. Service specifications consisting of several sub-functionalities may be constructed from collaborations, which can be reused in several services.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1613–1632, 2006.
© Springer-Verlag Berlin Heidelberg 2006

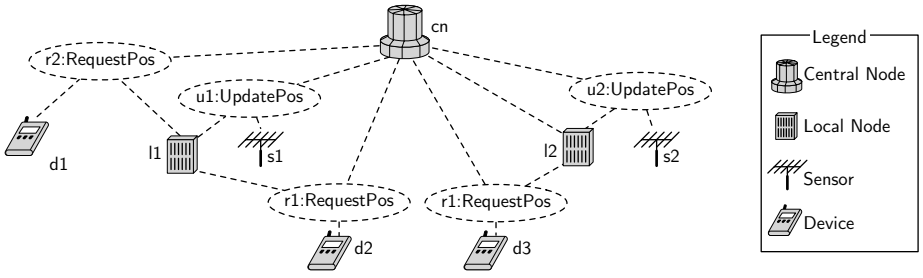


Fig. 1. Collaboration for the entire system

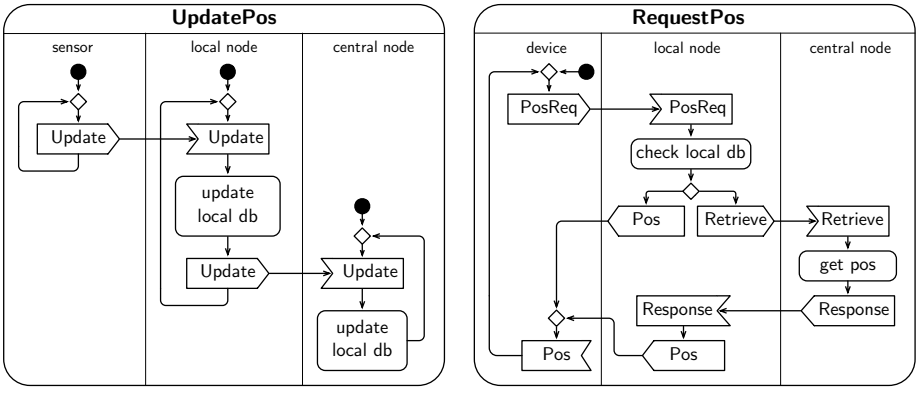


Fig. 2. Activities describing collaborations *GetPosition* and *UpdatePosition*

While we use the concept of UML 2.0 collaborations [4] to model the structural aspects of collaborations and services, we use UML 2.0 activities to specify their behavior. Figures 1 and 2 describe a service that retrieves the locations of small devices as part of a group communication service. In Fig. 1, we use icons for the collaboration roles and omit the frame of the system collaboration for clarity. Each device is connected to one local node, and all local nodes are connected to one central node. Sensors capture the movement of the devices and update the position information in the local nodes. This is specified by the collaboration uses *u1* and *u2* of the collaboration type *UpdatePos*. The behavior of this collaboration type is expressed in detail by the activity on the left side of Fig. 2. The sensors send updates of the device positions to their connected local node, which updates the entry in its local table. Thereafter, the local node forwards the update to the central node, which refreshes the central table containing the location of all devices. Furthermore, devices can ask their local node for the position of other devices. This behavior is specified by collaboration *RequestPos* outlined on the right side of Fig. 2. Device *d1* can, for instance, ask for the position of *d2* by sending *PosReq* to its local node. As *d2* is registered on the same local node, *l1* has the position of it in its local table and can therefore

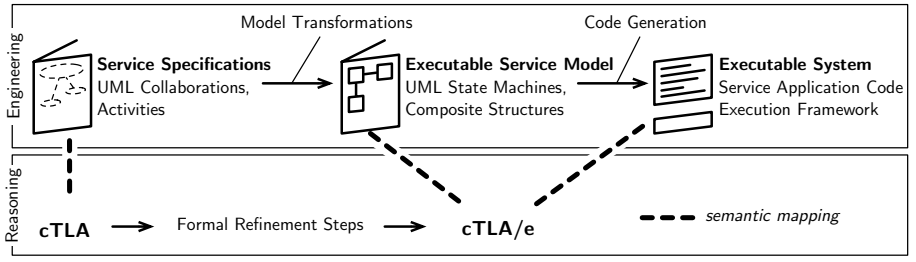


Fig. 3. Development approach using UML and cTLA

answer right away. If *d1* asks for *d3*, then *l1* sends a request to the central node and returns the reply to *d1*.

Our experiences using collaborations and activities to specify services are very encouraging, as collaborations allow intuitive but yet precise specifications of services. In order to execute a service, however, a behavioral description for each participating component is needed that can be efficiently executed in form of a program running on available platforms. To accomplish that, we follow the approach of stepwise refinement, adding more and more details until we get models that can be directly transformed into executable code. We intend to achieve these design steps by a set of model transformations in the spirit of MDA, as shown in the engineering part of Fig. 3. The result of the model transformations is an executable service model that is based on UML 2.0 state machines and composite structures. It is the input for our existing code generators that can generate programs executing the services on various Java platforms, appropriate for both the telecommunication as well as for the computing domain (cf. [5]).

To ensure the correctness of these transformations, we need a formal reasoning technique. The temporal logic cTLA [6] offers operators and techniques suitable for refinement [7], and it can capture such transformation steps in a formal way quite well, as shown in [6,8,9]. Moreover, the composition of services from collaborations can be directly expressed with the well-understood concept of process composition in cTLA.

Of course, a correctness-preserving development approach is only meaningful if we can guarantee that the generated code corresponds to the executable service model. Thus, we have to clarify formally that the executable code is a correct refinement of the executable service model in spite of the practical limitations of execution frameworks such as finite message buffers. For this sake, we introduce a cTLA specification style cTLA/e. It corresponds directly to the executable service model and describes which form a cTLA specification must have to be efficiently executable on existing service platforms. In this way, we establish a relationship between an intuitive service execution model in UML 2.0, the efficient execution of services on real platforms, and a formal model allowing reasoning and analysis.

In the following, we describe the execution platform in Sect. 2 and outline a profile for the executable service model based on UML 2.0 state machines in

Sect. 3. This execution model is based on the experience of about three decades of system engineering and originates from the SDL-based design methodology SOM [10], which described the basic modeling and execution mechanisms also used in the projects SISU and SISU II [11]. These projects resulted in the system engineering method TIMe [12] and had a *"major impact on the SDL methodology guidelines as well as on the SDL and MSC standards"* [13, p. 171]. As UML 2.0 adopted most of the language elements for the SDL mechanisms used in TIMe, we use it as a base for our model description. In addition, we sketch the specification technique cTLA in Sect. 4 and the specification style cTLA/e in Sect. 5. In Sect. 6 we outline the conformance of the executable service model with the executable system based on cTLA/e and discuss the properties a cTLA/e specification should have in order to properly address practical software and hardware limits. We close with a reflection about related approaches and some concluding remarks.

2 Service Execution Based on State Machines

Systems that execute services fall into the category of reactive systems as characterized by Pnueli [14]. A service typically requires the coordinated effort of several physically distributed devices [15], so that a system delivering a service needs to be decomposed into a number of reactive components running on different execution nodes. To define the behavior of the service components, we use communicating extended finite state machines in the form of UML 2.0 state machines. Similar descriptions are applied in ROOM [16] as well as in the formal description techniques Estelle [17] and SDL [18]. We assume that state machines communicate asynchronously using buffered message passing. This enables both asymmetrical client-server interactions typical for the computing domain as well as symmetrical peer-to-peer interactions common in the telecom domain (cf. [5]). Buffered communication also helps to decouple the different state machine instances and simplifies distribution, as this mechanism can be implemented for local as well as for remote communication without making changes to the model.

State machines define an executable abstract machine that can be implemented as a virtual machine layer providing runtime support. This gives the benefits of virtual machines in terms of adaptability and portability of applications. Contrary to other virtual machine approaches, the communicating state machines enable a highly efficient solution due to the following reasons:

- Asynchronous message passing avoids blocking on message sending.
- Transition-based execution models enable a very simple scheduling.
- Several state machines can be efficiently integrated in one native process.
- Generic mechanisms for input protection, error handling, and testing can be provided easily as part of the runtime support.

In this section, we outline how state machines can be executed using runtime support systems or execution frameworks. As an example, we present the runtime support system JavaFrame that facilitates execution of state machines on Java. Thereafter, we sketch other execution frameworks built on JavaFrame.

2.1 Runtime Support Systems and Execution Frameworks

To achieve a good performance of the executable code, the integration of state machines to native processes (e.g., operating system processes, Java threads) plays a significant role. A naive approach is to execute each state machine instance in a separate native process. This, however, would result in a significant space and time penalty caused by excessive context switching of the operating system. Therefore the common practice is to integrate several state machine instances into a single native process, which is called *light integration* in [13]. Scheduling of such state machines is extremely space and time efficient, providing the state machines have equal priority and can be allowed to run each transition to completion. Support for state machines can be integrated into a general virtual machine layer supporting the execution of state machines, the so-called *runtime support system* (RTS). Alongside process management and scheduling, an RTS can offer a range of services to the application layer, such as communication, timer routines, instance creation, logging, debugging and monitoring, as well as mobility management and load control. This approach has been used on numerous performance-critical products by many different companies in the telecommunication and automotive industry. Layered approaches can also be found in the computing domain. Instead of an RTS, one uses execution platforms like J2EE or newer platforms as for example JAIN SLEE [19], which try to more directly target the needs of traditional telecommunication services.

2.2 The Runtime Support System *JavaFrame*

To illustrate runtime support, we introduce *JavaFrame* [20], which is an RTS and Java execution framework facilitating the execution of UML 2.0 state machines. It is based on an RTS in C++ presented by Bræk and Haugen in [21], which implements an abstract SDL machine. JavaFrame provides a scheduler and base classes for state machines that can be extended with application-specific logic. Mediator objects encapsulate various communication protocols and routing functionality to send signals between state machines. Mediators can also be used to connect the state machines to environments not modeled by state machines.

With the behavior in form of single transitions, state machines naturally offer scheduling units that can be executed individually. Transitions are programmed in JavaFrame using transition methods containing nested if-statements. These if-statements differentiate the available trigger and the current control state and thereby realize the transition table of the state machine. The bodies of these nested statements contain the code that is executed as the effect of the transition. In particular, signals are sent to other state machines, operations are performed on local data or timers, and the next control state is determined. If an incoming signal should not be handled in the current state, it can be deferred by putting it into a dedicated defer queue, which is moved back into the input queue when another transition is executed. This corresponds to the save-concept of SDL. A scheduler controls a set of state machine instances and dispatches the events in their input queues by executing their transitions with the transition method

described above, using a FIFO ordering of queues. Following the *action-oriented* approach [13,21], the state of each state machine instance is stored explicitly in a data structure. This facilitates an efficient implementation, where the scheduler can manage the states of a large number of state machine instances. To execute a transition, the scheduler retrieves the current state and makes it available to the transition method as a parameter once a transition should be executed. Consequently, the transition method is reentrant, and needs to be provided only once per state machine type.

With this transition method, JavaFrame handles the selection of transitions and their execution in one single method call (as opposed to a more general solution, where transition selection and execution are implemented in separate methods [13]). This simplifies the scheduler further and considerably reduces the computation time to find an enabled transition, as the scheduler simply calls the transition method each time an event is available in the input queue. In consequence, transitions are enabled depending on their source state and their trigger only, and must not contain additional enabling conditions. In practice, this is not a real constraint, as transitions still may include decisions.

The scheduler of JavaFrame runs in one Java thread and executes only one transition at a time. In this way, JavaFrame complies to the run-to-completion semantics assumed in modeling languages like SDL, ROOM, and UML. The fact, that the Java thread of the scheduler may be interrupted by another thread, is not problematic, as these do not access any data of the interrupted state machine. The simple structure of the transition method (i.e., with the nested conditional statements) also implies that if several transitions are enabled in the same source state by the same input trigger, only the first one written in the transition method will be executed, while the code of the other transitions is never reachable. Such a situation can easily be avoided by combining these competing transitions to a single one, which contains a choice leading to the different effects of the original transitions.

2.3 State Machine Execution on Other Platforms

JavaFrame can be used directly to implement state machine-based specifications on the standard Java platform J2SE. Due to its simplicity, it may also be seen as a prototype that can guide the implementation of execution frameworks on other platforms. To facilitate the execution of telecommunication services further, a prototypical service execution framework *ServiceFrame* [22] was devised by Ericsson extending the basic JavaFrame execution mechanisms with concepts targeted to service engineering and deployment. Specifically for the domain of telecommunication services, ServiceFrame contains a part defining service components like *user agents* and *terminal agents*. In addition, a number of resource adapters were defined to connect the system to existing technologies, interfaces and transport protocols like Parlay-X, SIP, as well as Bluetooth connections and location tracking via GSM or WLAN for mobile devices. The part for the execution of services, called *ActorFrame* [23], is an extension of JavaFrame, adding routing mechanisms, an addressing scheme and protocols for the management of

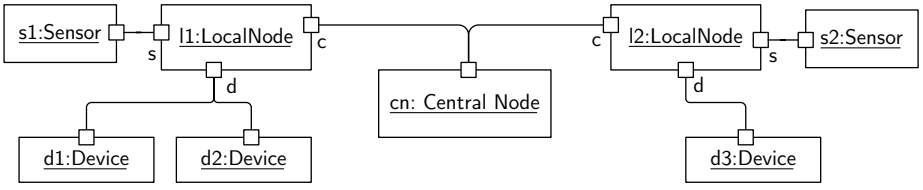


Fig. 4. Object diagram of the system

the system structure. The initial version of ActorFrame was implemented on the standard Java platform, J2SE, followed by versions running on the J2EE [24] and later also the J2ME platform, which makes it possible to run parts on the system also on mobile devices.

2.4 Code Generation

As parts of our integrated service engineering tool suite Ramses [25], we developed code generators [26,27] for the ActorFrame platform, both the J2EE as well as the J2SE version. Based on them, a number of prototypical service applications were realized and deployed, including services running in the operational network of the Norwegian telecom operator Telenor. As input, Ramses uses UML 2.0 models based on state machines as presented in the next section.

3 Executable Service Models in UML 2.0

In the following, we outline a profile in UML 2.0 that can be directly mapped to an execution in JavaFrame-based systems. The complete profile is presented in [28]. In particular, we introduce constraints on transitions to facilitate scheduling and refine some semantic variation points of UML 2.0.

Figure 4 shows an object diagram of our example system with a number of devices, two local nodes, and the central node in the middle. Each state machine owns some ports that are used to transmit the signals to other state machines via the links connecting them. A state machine can send a signal by putting it into the output queue of a port. Thereafter, the signal is transmitted via a link. At the receiver side, the signal is added to the common input queue of the state machine. If one port is connected to several others, the signal contains some routing information that can be used by the sending port to choose the correct receiver. We assume hereby that signals are transferred in an order-preserving and reliable way, and that the queues are unbounded[1].

Figure 5 shows the state machine for a local node. For the execution model, we use only a subset of UML 2.0 state machines. In particular, we assume that a state machine has exactly one region and that transitions have a certain structure, which is described later. To distinguish such state machines from other state machines in UML, we mark them with the stereotype «executable».

[1] In Sect. 6, we describe in the context of the formal method cTLA under which circumstances these practical limitations can be handled.

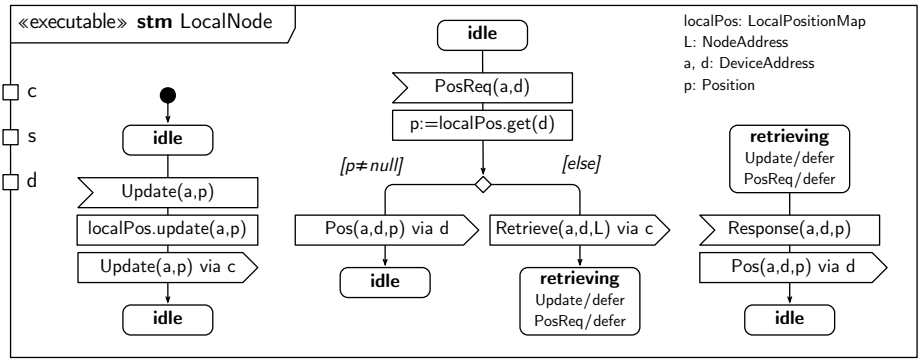


Fig. 5. State machine for a local node

As in JavaFrame, events are either the reception of signals or the expiration of local timers. UML assumes the events arriving at a state machine to be stored in an event pool, and gives no further rules for the order of dispatching them, intentionally allowing different strategies. We assume the input pool to be a FIFO queue like in SDL processes, so that events are dispatched in the order of their arrival. This matches the scheduling procedure in JavaFrame. Deferred events are specified in UML by writing them into the state symbol with the keyword */defer*. For example, in the state *retrieving* in Fig. 5, incoming *Update* or *PosReq* signals are deferred, until *Response* arrives and the state machine changes into state *idle*.

Actions can be executed as the effect of transitions. A state machine operates on its auxiliary variables, controls local timers, and sends signals to other state machines. Actions may also call operations defined for the auxiliary data. Such actions must execute within the same run-to-completion step and therefore be local and not waiting on external events. In our example, after receiving an update, the state machine updates the local data structure *localPos* by using its operation *update()*. Send signal actions can be used to transmit signals to other state machines. We assume that these actions are assigned to a port (using the keyword *via*) and that the signals contain information so that the port may decide about the destination. The local node, for instance, includes its own address L in signal *Retrieve* that is sent via port c towards the central node. This address may be used by the output port of the central node to route signal *Response* for the answer.

As we only use simple states and pseudo states of the kinds *choice* and *initial*, we may distinguish the following forms of transitions:

- A *simple transition* connects two control states without any decisions or pseudo states.
- A *compound transition* is similar to a simple transition but can contain choices, so that its effects and the target state can depend on a decision. The decision is made by the guards that are declared on each branch originating

from a choice. UML requires that at least one of these guards is true, so that a compound transition can always be completed once it is started.
– An *initial transition* originates from an initial pseudo state and is executed when the state machine is started. Each state machine has exactly one initial transition. Like a compound transition, an initial transition may also use choices that result in different branches.

Due to the scheduling mechanism in the execution platform, we assume that all transitions, that do not originate in an initial pseudo state, have exactly one trigger that matches either a signal reception or a timer expiration event. The scheduler assumes a transition to be enabled if the state machine is in the declared source state and the next event to be dispatched matches the one declared as trigger by the transition. In consequence, a transition may not declare any additional guards that would prevent its execution. We further assume that an event is not deferred in a state if a transition originates from that state with the same event as trigger. Thus, an incoming event is either consumed by a transition or deferred in a given state.

4 Compositional Temporal Logic of Actions (cTLA)

Lamport's Temporal Logic of Actions (TLA, [29]) is a linear-time temporal logic modeling the behavior of a system as a set of infinitely long state sequences

$$\langle s_0, s_1, s_2, \ldots \rangle.$$

Thus, the TLA semantics fits excellently with that of the state machines introduced above which, in the end, also model infinite sequences of states s_i starting with an initial state s_0. Compositional TLA (cTLA, [6]) was derived from TLA to provide more easily comprehensible specifications and offer a more flexible composition of specifications. cTLA is oriented at programming languages and introduces the notion of processes. A cTLA process can be in a simple form which directly describes system behavior by means of state transition systems. A process can also be compositional and describe systems as a combination of other process instances each specifying a sub-functionality of the system.

An example of a simple process type is sketched in Fig. 6. The header *LocalNode* declares the name of the process type while generic module parameters like *DeviceAddr* enable to specify a spectrum of similar process instances by a single process type. *Signals* is a constant record-typed expression. The body of a simple cTLA process type describes a state transition system. It contains a set of variables like *state* or *inQueue* modeling the state space. The subset of initial states is specified by the predicate *INIT*. The transitions are expressed by actions (e.g., *enqueue*, *dequeueC*) which are predicates on pairs of a current and a next state describing a set of transitions each. Variables in simple form (e.g., *inQueue*) refer to the current state while the next state is described by the so-called primed form (e.g., *inQueue'*). The statement *UNCHANGED* lists

```
 PROCESS LocalNode (DeviceAddr: ANY; MyDevices: SUBSET(DeviceAddr);
                    NodeAddr: ANY; MyAddress: NodeAddr;
                    Pos: ANY; unknownPos: Pos)
CONSTANTS
  Signals ≜ [[ t: {Start, Update, PosReq, Response, Retrieve};
               a: DeviceAddr; d: DeviceAddr; l: MyAddr; p: Pos ]] ;
VARIABLES
  state: {initState, idle, retrieving}; localPos: [MyDevices → Pos];
  inQueue: QUEUE OF Signals;    deferQueue: QUEUE OF Signals;
  outQueueC: QUEUE OF Signals;   outQueueD: QUEUE OF Signals;
INIT ≜
  state = initState ∧ inQueue = EMPTY ∧ deferQueue = EMPTY ∧
  outQueueC = EMPTY ∧ outQueueD = EMPTY ∧ localPos ∈ [MyDevices → Pos];
ACTIONS
  enqueue (inSignal : Signals) ≜
    inQueue′ = inQueue ∘ ⟨inSignal⟩ ∧ state ≠ initState ∧
    UNCHANGED ⟨deferQueue, state, outQueueC, outQueueD, localPos⟩;
  dequeueC (outSignal : Signals) ≜
    outQueueC ≠ EMPTY ∧ outSignal = FIRST(outQueueC) ∧
    outQueueC′ = TAIL(outQueueC) ∧
    UNCHANGED ⟨inQueue, deferQueue, state, outQueueD, localPos⟩;
  dequeueD (outSignal : Signals) ≜ ... ;
  initial ≜ state = initState ∧ state′ = idle ∧
    localPos′ = [d ∈ MyDevices | d ↦ unknownPos] ∧
    UNCHANGED ⟨inQueue, deferQueue, outQueueC, outQueueD⟩;

INTERNAL ACTIONS
  update ≜ state = idle ∧ FIRST(inQueue).t = Update ∧
    state′ = idle ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue) ∧ deferQueue′ = EMPTY ∧
    localPos′ = [localPos EXCEPT FIRST(inQueue).a ↦ FIRST(inQueue).p] ∧
    outQueueC′ = outQueueC ∘ ⟨ [[t ↦ Update; a ↦ FIRST(inQueue).a;
                                 d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                                 p ↦ FIRST(inQueue).p ]] ⟩ ∧
    UNCHANGED ⟨outQueueD⟩;
  requestPos ≜ state = idle ∧ FIRST(inQueue).t = PosReq ∧
    state′ = IF FIRST(inQueue).a ∈ MyDevices THEN idle ELSE retrieving ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue) ∧ deferQueue′ = EMPTY ∧
    localPos′ = localPos ∧
    outQueueC′ = outQueueC ∘ IF FIRST(inQueue).d ∈ MyDevices THEN EMPTY
                             ELSE ⟨ [[t ↦ Retrieve; a ↦ FIRST(inQueue).a;
                                      d ↦ FIRST(inQueue).d;
                                      l ↦ MyAddress ]] ⟩ ∧
    outQueueD′ = outQueueD ∘ IF d = FIRST(inQueue).d ∈ MyDevices
                             THEN ⟨ [[t ↦ Response; a ↦ First(inQueue).a;
                                      d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                                      p ↦ localPos[First(inQueue).d]] ]⟩
                             ELSE EMPTY;
  retrievePos ≜ ... ;
  deferInRetrieving ≜ state = retrieving ∧ FIRST(inQueue).t ∈ {Update, PosReq}
    ∧ inQueue′ = TAIL(inQueue) ∧ deferQueue′ = deferQueue ∘ ⟨FIRST(inQueue)⟩ ∧
    UNCHANGED ⟨state, localPos, outQueueC, outQueueD⟩;
WF: dequeueC, dequeueD, initial, update, requestPos,
    retrievePos, saveInRetrieving;
END
```

Fig. 6. cTLA/e process modeling the local node

variables not changed by an action. Action parameters like *inSignal* allow to model different actions by a single representation. Actions can be distinguished into two classes. External actions can be coupled with actions of the process environment while internal actions cannot.

We can provide actions with weak and strong fairness properties guaranteeing that they are carried out in a lively manner. In particular, weak fairness forces the execution of an activity if it would be enabled continuously otherwise. Strong fairness forces the execution even if the action is sometimes disabled. Unlike TLA, cTLA provides for conditional fairness assumptions to ensure the consistency of the process compositions introduced below. A fairness statement refers to periods of time in which an action is both enabled and the environment of the process is ready to tolerate the action. The statement *WF: dequeueC, dequeueD,...* indicates that the listed actions have to be carried out weak fairly.

A process type describes a set of TLA state sequences. The first state s_0 of each modeled state sequence has to fulfill the initial condition *INIT*. The state changes $\langle s_i, s_{i+1} \rangle$ either correspond with a process action or with a so-called stuttering step in which the current and the next states are equal (i.e., $s_i = s_{i+1}$). The fairness assumptions have to be fulfilled as well. cTLA also allows to define additional real time properties [30] and the description of continuous behavior [31] which we omit here for the sake of brevity.

Compositional cTLA processes model systems as compositions of concurrent process instances. Since the process variables are encapsulated and can only be referenced by the actions of the process defining them, the system state space is basically the vector of the variables of all process instances belonging to the system. We compose processes with each other by coupling their external actions to joint system actions. Formally, a system action is a conjunction of the corresponding process actions which therefore are executed simultaneously. A process may contribute to a system action with either exactly one process action or with a stuttering step. An internal process action, however, must only be coupled with stuttering steps of the other processes.

Figure 7 describes a compositional process type. It consists of the process instances *c*, *l1*, *l2*, etc. which are listed in the section *PROCESSES*. At that place, we also specify the module parameter instantiations (e.g., the parameter *myDevices* of the instance *l1* of process type *LocalNode* in Fig. 7 is instantiated with the set *{d1, d2}*). The system actions are depicted in the lower part of the specification[2] as conjunctions of process actions. For instance, the system action *ctol1* corresponds to the joint execution of the process actions *dequeueC* of process *C* and *enqueue* of *l1* while the other processes perform stuttering steps. The data transfer between *c* and *l1* is modeled by the action parameter *sig*. Moreover, we added an additional conjunct *sig.l = l1* enabling the execution of the action for certain action parameter settings only. In [6] we proved that compositional cTLA processes can be transformed into equivalent simple processes which enables nested system specifications.

[2] To keep the specification short, we omitted processes performing stuttering steps in each system action description.

```
PROCESS System
CONSTANTS
  DevAddr ≜ {d1, d2, d3}; NodeAddr ≜ {l1, l2};
  Ps ≜ [[ x : REAL; y : REAL; z : REAL ]] ;
  uPs ≜ [[ x ↦ 0, y ↦ 0, z ↦ 0 ]] ;
  Sig ≜ [[ t : {Start, Update, PosReq, Response, Retrieve};
           a : DevAddr; d : DevAddr; p : Ps ]] ;
PROCESSES
  cn: CentralNode (DeviceAddr ← DevAddr, Pos ← Ps, unknownPos ← uPs);
  l1: LocalNode (DeviceAddr ← DevAddr, myDevices ← {d1, d2},
                 NodeAddr ← NodeAddr, MyAddress ← l1,
                 Pos ← Ps, unknownPos ← uPs);
  s1: Sensor (DeviceAddr ← DevAddr, myDevices ← {d1, d2}, Pos ← Ps);
  d1: Device (DeviceAddr ← DevAddr, myDeviceAddr ← d1, Pos ← Ps);
```

...initializations of local node l2, sensor s2 and devices d2 and d3...

```
INTERNAL ACTIONS
```

Initial

```
  cnInitial ≜ cn.initial; l1Initial ≜ l1.initial; l2Initial ≜ l2.initial;
  d1Initial ≜ d1.initial; d2Initial ≜ d2.initial; d3Initial ≜ d3.initial;
  s1Initial ≜ s1.initial; s2Initial ≜ s2.initial;
```

local nodes ↔ central node (portC)

```
  l1toc(sig: Sig) ≜ l1.dequeueC(sig) ∧ c.enqueue(sig);
  l2toc(sig: Sig) ≜ l2.dequeueC(sig) ∧ c.enqueue(sig);
  ctol1(sig: Sig) ≜ c.dequeueC(sig) ∧ l1.enqueue(sig) ∧ sig.l = l1;
  ctol2(sig: Sig) ≜ c.dequeueC(sig) ∧ l2.enqueue(sig) ∧ sig.l = l2;
```

... actions for other connections ...

```
END
```

Fig. 7. cTLA/e process modeling the global system

5 cTLA/e: An Executable Form of cTLA

cTLA is a powerful means to describe various forms of behavior. The cTLA specifications, however, may have a form that is difficult to implement efficiently. Therefore, we describe a special cTLA specification style (cTLA/e), which directly models the mechanisms of the execution platforms exemplified by JavaFrame in Sect. 2. cTLA/e determines a form for simple processes corresponding to state machines explained in Sect. 5.1 and a form for compositional processes to couple the state machines in Sect. 5.2.

5.1 cTLA/e Process for State Machines

In the following, we will sketch the cTLA/e models of state machines by the specification of the local node from the example listed in Fig. 6. This process corresponds to the state machine of the local node depicted in Fig. 5. One state machine is represented by one cTLA/e process. The control state is described by a cTLA variable *state* expressing the enumeration of the control state identifiers. Incoming signals are placed in the data structure *inQueue*, which is a sequence

of signals with the operations *FIRST()* to obtain the first element and *TAIL()* to get the queue after removing the first element. The operator $\circ$ denotes the concatenation of queues. Similarly, the defer queue for signals is modeled by the cTLA variable *deferQueue*. Signals are appended to the input queue by the action *enqueue*, which has the received signal as action parameter.

For each port used to send signals to other state machines, the process contains an output queue[3]. Signals are records, where the field t denotes the type of the signal, a the device address calling for a position or part of an update, d the device address for which a position is requested, l the address of a local node retrieving a position, and p the position information. To send a signal via a port, a transition adds it to the corresponding output queue. A dequeue action defined for each port (e.g., *dequeueC* and *dequeueD*) is used to transmit the signals from the output queues to their respective receivers. Additional variables represent the auxiliary variables of the state machine. For instance, a local node stores the positions of its local devices in the map *localPos*.

Every transition of the state machine is represented by a cTLA action formulated as a conjunction of several sub-actions $t_{trans} = t_{en} \wedge t_{next} \wedge t_{qm} \wedge t_{send} \wedge t_{aux}$, each having a distinct purpose:

- The enabling sub-action $t_{en} = t_{trigger} \wedge t_{\text{prev}}$ determines whether a transition is ready to execute. This depends on the first event in the input queue ($t_{trigger}$) and the current control state (t_{prev}). For example, the action *update* defines a transition enabled in control state *idle* and for the signal *Update* with $state = idle \wedge FIRST(inQueue).t = Update$. As an initial transition has no trigger, sub-action $t_{trigger}$ is omitted in action *initial*.
- The target state sub-action t_{next} specifies the change of the control state. It simply is an assignment to the control state variable. For compound transitions including several branches, the assignment can include an if-statement. The target of the *requestPos* transition, for instance, is either state *idle* or *retrieving*.
- The queue maintenance sub-action t_{qm} describes the move of the content of the defer queue to the front of the input queue, so that they are again available for consumption in the next state. The sub-action $t_{qm} \triangleq inQueue' = deferQueue \circ tail(inqueue) \wedge deferQueue' = EMPTY$ is identical for every transition. As the defer queue is empty when the initial transition is executed, it is not necessary to include this sub-action in the action *initial*.
- Sub-actions t_{send} model the transmission of signals, simply by appending them to the corresponding output queue. The signals sent may depend on conditions, which can be expressed by an if-statement. For example, transition *requestPos* either sends *Response* via port D or *Retrieve* via port C.
- Sub-actions t_{aux} specify the new settings of the local auxiliary variables. The local position map *localPos*, for instance, is updated with the new position in transition *update*.

[3] In the example, no signals are sent from the local node to the sensor via port *s*. Therefore, this port is not represented with an output queue.

Like on our execution platforms based on JavaFrame, deferred signals are moved into a the dedicated defer queue by an explicit action. This action is a conjunct $t_{trigger} \wedge t_{prev} \wedge t_{defer}$, with t_{defer} performing the actual move into the defer queue. For instance, in Fig. 6, *deferInRetrieving* removes the signals *Update* or *PosReq* from the input queue and appends them to the defer queue.

Similarly to SDL, we model timers by means of signals. The starting, stopping and triggering of a timer is specified by auxiliary cTLA actions. Once a timer expires, the runtime support system places a signal representing the timer expiration in the input queue. In our example system, we use timers in the sensors which, however, are not listed for the sake of brevity.

5.2 cTLA/e Process for the Global System

The system is specified by a compositional cTLA process combining the processes for the individual state machines, as shown in Fig. 7. After declaring constants for the used types such as device addresses, signal formats and positions, it defines a process instance for each state machine instance and passes parameters to them. The configuration reflects the system structure given in Fig. 4 by initializing local nodes and sensors with the device addresses attached to them.

According to the system structure, the corresponding dequeue and enqueue actions are coupled together, so that signals can be transfered from an output queue to the input queue of the receiver. In our example, we represent each link between two state machines by an individual cTLA action. For example, the links from the central nodes to each of the local nodes are represented by cTLA actions *ctol1* and *ctol2* in which the additional conjuncts $sig.l = l1$ and $sig.l = l2$ model the routing decision. To enable scalable system models, we can also use coupling descriptions specifying various links and, in particular, dynamic connections by a single cTLA system action (cf. [8]).

6 Executing cTLA/e Specifications

To provide the complete formal proof, that our code generators produce software code implementing a cTLA/e specification correctly, we need to create a fully-fledged cTLA model of the code, which is beyond the scope of this paper. Therefore, we only provide a sketch of the proof. As mentioned previously, the specification style cTLA/e was laid out in a way that its actions correspond with the program steps of the generated code based on JavaFrame. Moreover, the variables used in cTLA/e reflect directly the variables in the executable code. For instance, the sub-action t_{qm} is similar to the step of the implementation where in a transition the first signal is removed from the input queue and previously deferred signals are moved to the front of the input queue in the order of their deferral. Thus, we can describe the execution of a transition as an order of steps:

$$S_i \xrightarrow{t_{trigger}} \widehat{S}_{i,1} \xrightarrow{t_{prev}} \widehat{S}_{i,2} \xrightarrow{t_{aux}} \widehat{S}_{i,3} \xrightarrow{t_{send}} \widehat{S}_{i,4} \xrightarrow{t_{next}} \widehat{S}_{i,5} \xrightarrow{t_{qm}} S_{i+1}$$

As previously mentioned, the implemented state machines follow the run-to-completion semantics, so that the sequence of steps is carried out without

interruptions by other events. Therefore, it is easy to prove formally that this sequence implies the sequence

$$\overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{t_{trans}} \overline{S}_{i+1}$$

That means the first five steps of the executed transition are mapped to stuttering steps in cTLA/e, while the last step is mapped to the cTLA/e action modeling the entire transition in one (atomic) step. This is a well-known example of a formally correct refinement step as described for example in [32]. Likewise, we can verify that a signal deferral consisting of the steps

$$S_i \xrightarrow{t_{trigger}} \widehat{S}_{i,1} \xrightarrow{t_{prev}} \widehat{S}_{i,2} \xrightarrow{t_{defer}} S_{i+1}$$

implements the cTLA/e defer action.

In cTLA/e, a signal transmission is modeled by three distinct actions: (1) the transition putting the signal into an output queue, (2) the action transferring the signal from the output queue to the input queue of the receiver (as a conjunction of two process actions) and (3) the transition triggered by the signal that consumes it. Thus, action (2) is an abstraction of the transmission mechanism of a middleware layer in an implementation and the signals currently in the cTLA/e output queues are assumed to be under transmission.

Of course, we have to consider that resources in the real world are limited and computation steps take time. In particular, the size of signal queues is bounded and buffer overflows may occur. In our example, a sensor may send position updates so frequently, that the local node cannot process all of them. To avoid this, we can introduce mechanisms already on the specification level. We may, for instance, require the sensor to wait for an acknowledgment from the local node before sending another update. Alternatively, updates may only be sent when requested by the local node. In this case, one can verify by cTLA-based invariant proofs that the queues do not exceed an upper bound. Furthermore, we may use real-time reasoning to guarantee the boundedness of queues. For instance, we may enforce a minimum waiting time for the sensor and maximum response time properties for other system actions using the real-time extension of cTLA [30]. Then we can prove that the local node can handle an update signal even in peak situations before the next update is triggered. This is complementary to the technique used in [21] which estimates the execution time of transitions.

A deadlock can occur if there is a signal at the first position of the queue that a state machine cannot handle in its current state (i.e., neither consume in a transition nor defer). To prevent this kind of design flaw, we should verify by means of cTLA invariant proofs that every incoming signal can be handled. In our example, the local nodes have two states[4] *idle* and *retrieving*. The signals *Update* and *PosReq* can always be consumed in state *idle* (by the actions *update*

[4] We can disregard the initial pseudo state *initState* here, as the originating initial transition is enabled independently of the input queue and will be eventually executed due to its fairness property.

or *requestPos* in Fig. 6) and deferred in state *retrieving* (by action *saveInRetrieving*). In contrast, signal *Response* is only consumed in the state *retrieving*. Therefore, we must verify the cTLA invariant, that this signal is only sent by the central node if the local node is in the state *retrieving*. Based on the activity diagram in Fig. 2 it is evident that this invariant is straightforward.

So far, we considered safety properties guaranteeing that "nothing wrong" happens. Beyond that, the layout of cTLA/e, and the scheduling mechanisms based on JavaFrame also allow assertions about liveness properties, describing that *"...something good eventually happens..."* [33]. In cTLA, liveness is expressed by the fairness assumptions introduced in Sect. 4. The layouts of cTLA/e and the JavaFrame-based scheduler guarantee that every transition once enabled will eventually be executed, since the following properties hold:

- Due to the isolation of state machines and the fact that transitions are enabled based on the source state and trigger event only, a transition once enabled will remain enabled until it is executed.
- As explained in Sect. 2, there is at most one transition enabled for each combination of a source state and a trigger event.
- Due to the cTLA invariant proof, all received signals can be handled.
- The scheduler serves all of its state machines in a round-robin fashion.

One can verify that these properties imply the strong fairness properties (and, in consequence, the weak fairness properties) of the corresponding cTLA/e actions. This is a valuable property of our execution platform, as it is the prerequisite to include fairness reasoning on the more abstract specifications of our system as well. If we can prove that fairness assumptions of more abstract collaborations are fulfilled by the cTLA/e refinement, it is evident, that these assumptions are also realized by the executable code.

7 Related Work

Closest related to our work is probably that of the specification approach and language DisCo [34], which, like cTLA, is based on the Temporal Logic of Actions. Similar to collaborations, DisCo is focusing on the cooperation of objects. Instead of processes as in cTLA, DisCo uses layers that may be composed or refined. To facilitate a specification-driven approach, Pitkänen [35,36] introduces an additional level of refinement called TransCo. This is a subset of the DisCo language and oriented towards business components and transactions. TransCo can be derived from DisCo by refinement and then further be translated into J2EE applications by an experimental code generator. The concept of an intermediate formal language like cTLA/e or TransCo is also present in the B-Method [37], where a subset of B — called B0 — is closer to imperative languages that are easier to implement. The intermediate languages TransCo, B0, and cTLA/e focus on different domains or platforms. While TransCo targets at transaction processing, and B0 is close to sequential code like ADA, cTLA/e is an abstraction of the executable state machines described above.

In this paper, we focused on the formal treatment of an execution model to ensure correctness of the resulting programs. If we extend our scope towards the development of reactive systems in general, we naturally find other methods with slightly different aims, specialized towards other domains. One approach that seems to cover the step from specifications to executable code in a rather complete way, is that of Burmester et al. in [38] which is integrated into the FUJABA toolset. They focus on specifications of systems including real-time properties. Similar to collaborations, they specify patterns that can be verified independently and composed together. For the description of these patterns, UML state machines extended with real-time properties are used. To transform a specification into executable systems, an intermediate model is described in [39] that takes platform-specific aspects into consideration, such as the assignment of state machine instances to execution threads. For the implementation they propose a direct mapping of one state machine instance to one real-time execution thread, instead of using a scheduler that takes advantage of the state machines, as described in Sect. 2.

8 Concluding Remarks

We described how distributed services can be efficiently executed based on communicating state machines. Moreover, we outlined the mechanisms of JavaFrame to exemplify how execution platforms and support systems can be constructed. It was further discussed which form UML 2.0 state machines should have in order to be easily transformable to programs using the presented execution mechanisms. We defined a cTLA specification style (cTLA/e) to combine the correctness-preserving service design with the efficient execution mechanisms. cTLA/e is dedicated to an easy and correct mapping of the state machines forming the input of JavaFrame-based implementations. We made plausible that the implementations fulfill interesting properties concerning the fairness of execution and we outlined how boundedness of signal queues can be ensured.

We described a triangle relationship between the efficient execution of services, the intuitive modeling based on UML, and the formal analysis based on temporal logic with cTLA/e. This relationship also aligns the scopes of three different kinds of engineers that perform activities in service engineering:

- Execution platform designers create mechanisms for the execution and deployment of services that need computational models allowing an efficient execution, such as the state machines presented in Sect. 2.
- Service engineers focused on specific applications want to have suitable modeling concepts and generally accepted notations, such as the UML 2.0 state machines of Sect. 3.
- Providers of tools for modeling, analysis, and transformations need a formal logic like cTLA of Sect. 4 to to reason about the correctness of tools and methods.

With cTLA/e we provided such an alignment for the execution of services. It is the final stage in our strategy to generate executable components from

formal collaborations describing the services. In addition to cTLA/e, we developed another cTLA specification style modeling collaborations which uses UML 2.0 collaborations for a structural description and UML activities for the behavioral part, like the ones briefly presented in the introduction. In the next step, we will specify how this cTLA style can be refined to cTLA/e. In particular, we want to provide service engineers with the suitable means for the correctness-preserving top-down construction of distributed services. Here, cTLA already proved its capability for various application domains [6,8,9,31]. As part of this work we have to reduce collaborations to component models as needed for the execution. This means to re-arrange the process structure described by the collaborations and to split it into the behavior that each service component contributes to a collaboration. An integral part of such a refinement is the adaption of the process couplings and the cTLA actions into the form we described by cTLA/e.

The combination of UML 2.0 modeling with cTLA-based reasoning offers a number of practical advantages for service engineering in general. Most prominent is the realization of the correctness-preserving refinement as UML 2.0 model transformations. Here, the cTLA refinement steps are a fundament for creating MDA tools performing suitable model transformations. While these tools use cTLA formalizations of the UML models and the refinement steps, for the service engineer cTLA will in fact be invisible.

References

1. Sanders, R.T., Castejón, H.N., Kraemer, F.A., Bræk, R.: Using UML 2.0 Collaborations for Compositional Service Specification. In: ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems. (2005)
2. Rossebø, J.E.Y., Bræk, R.: Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In: Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06), IEEE Computer Society Press (2006) 206–215
3. Castejón, H.N., Bræk, R.: A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. ICSE's 5th Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06) (2006)
4. Object Management Group: Unified Modeling Language: Superstructure Version 2.0. (2005)
5. Bræk, R., Floch, J.: ICT Convergence: Modeling Issues. In Amyot, D., Williams, A.W., eds.: SAM'04 - Fourth SDL and MSC Workshop. Volume 3319 of Lecture Notes in Computer Science, Springer (2004) 237–256
6. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337
7. Mester, A., Krumm, H.: Composition and Refinement Mapping based Construction of Distributed Applications. In: Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Aarhus, Denmark, BRICS (1995)

8. Herrmann, P.: Formal Security Policy Verification of Distributed Component-Structured Software. In König, H., Heiner, M., Wolisz, A., eds.: Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003), Berlin, Germany. Volume 2767 of Lecture Notes in Computer Science, Springer-Verlag (2003) 257–272
9. Herrmann, P.: Temporal Logic-Based Specification and Verification of Trust Models. In Stølen, K., Winsborough, W.H., Martinelli, F., Massacci, F., eds.: iTrust 2006. Volume 3986 of Lecture Notes in Computer Science, Heidelberg, Springer–Verlag (2006) 105–119
10. Bræk, R.: Unified System Modelling and Implementation. In: International Switching Symposium, Paris, France (1979) 1180–1187
11. SISU II Project: (http://www.sintef.no/units/informatics/projects/sisu/)
12. Bræk, R., Gorman, J., Haugen, Ø., Melby, G., Møller-Pedersen, B., Sanders, R.T.: Quality by Construction Exemplified by TIMe — The Integrated Methodology. Telektronikk **95**(1) (1997) 73–82
13. Mitschele-Thiel, A.: Systems Engineering with SDL: Developing Performance-Critical Communication System. John Wiley & Sons, Inc., New York, NY, USA (2001)
14. Pnueli, A.: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. Current Trends in Concurrency. Overviews and Tutorials (1986) 510–584
15. Floch, J., Bræk, R.: Towards Dynamic Composition of Hybrid Communication Services. In: SMARTNET '00: Proceedings of the IFIP TC6 WG6.7 Sixth International Conference on Intelligence in Networks, Deventer, The Netherlands, Kluwer, B.V. (2000) 73–92
16. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., New York, NY, USA (1994)
17. ISO: ESTELLE: A Formal Description Technique Based on an Extended State Transition Model. International Standard ISO/IEC 9074 edn. (1997)
18. ITU-T: Recommendation Z.100: Specification and Description Language (SDL). (2002)
19. Lim, S.B., Ferry, D.: JAIN SLEE 1.0 Specification, Final Release. Sun Microsytems, Inc. and Open Cloud Ltd. (2004)
20. Haugen, Ø., Møller-Pedersen, B.: JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering (2000)
21. Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall International (1993)
22. Bræk, R., Husa, K.E., Melby, G.: ServiceFrame Whitepaper. Ericsson NorARC, Asker, Norway. (2002)
23. Melby, G., Husa, K.E.: ActorFrame Developers Guide. Ericsson NorARC, Asker, Norway. (2005)
24. Melby, G.: Using J2EE Technologies for Implementation of ActorFrame Based UML 2.0 Models. Master's thesis, Agder University College, Grimstad, Norway (2003)
25. Kraemer, F.A., Samset, H.: Ramses User Guide. Avantel Technical Report 1/2006, Department of Telematics, NTNU, Trondheim, Norway (2006)
26. Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)
27. Støyle, A.K.: Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology (2004)

28. Kraemer, F.A.: Profile for Service Engineering: Executable State Machines. Avantel Technical Report 2/2006, Department of Telematics, NTNU, Trondheim, Norway (2006)
29. Lamport, L.: Specifying Systems. Addison-Wesley (2002)
30. Graw, G., Herrmann, P., Krumm, H.: Verification of UML-Based Real-Time System Designs by means of cTLA. In: Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC2K), Newport Beach, IEEE Computer Society Press (2000) 86–95
31. Herrmann, P., Krumm, H.: A Framework for the Hazard Analysis of Chemical Plants. In: Proceedings of the 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD2000), Anchorage, IEEE CSS, Omnipress (2000) 35–41
32. Lamport, L.: Refinement in State-Based Formalisms. Technical Report 1996-001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California (1996)
33. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21**(4) (1985) 181–185
34. Kurki-Suonio, R.: A Practical Theory of Reactive Systems. Springer (2005)
35. Pitkänen, R.: A Specification-Driven Approach for Development of Enterprise Systems. In: Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'04), Turku, Finland (2004)
36. Pitkänen, R.: Tools and Techniques for Specification-Driven Software Development. PhD thesis, Tampere University of Technology (2006)
37. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
38. Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS). (2004)
39. Burmester, S., Giese, H., Schäfer, W.: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: Proceedings of the European Conference on Model Driven Architecture — Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany. Volume 3748 of Lecture Notes in Computer Science, Springer (2005) 25–40

Developing Mobile Ambients Using an Aspect-Oriented Software Architectural Model

Nour Ali, Carlos Millán, and Isidro Ramos

Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera s/n
E-46022 Valencia, Spain
{nourali, cmillan, iramos}@dsic.upv.es

Abstract. Nowadays, distributed and mobile systems are acquiring greater importance and becoming more widely used to support ubiquitous computing. However, developing systems of this kind is a difficult task. Instead of concentrating on how problems should be solved developers must worry about implementation details. Ambient Calculus is a formalism that provides primitives to describe mobile systems in an abstract way. Aspect-oriented software development and software architectures promise to achieve reusability, maintenance and adaptability, which are all essential for the development of distributed systems. In this paper, we present how a platform-independent model called Ambient-PRISMA combines both Ambient Calculus and Aspect-Oriented Software Architecture for the specification of distributed and mobile systems. A platform-specific model in .Net for supporting Ambient-PRISMA code generation is also presented.

1 Introduction

Computer networks have become pervasive and ubiquitous. They allow users to communicate and to collaborate independently of their locations. They also involve new devices that form part of our daily life such as PDAs or smartphones. Nowadays, software systems have to cope with these advances in order to exploit their distributed nature. Distributed software systems have to take into account particularties; for example complex structures, new non-functional requirements such as fault tolerance and security, and dynamic adaptation. As a result, software development processes must be able to support the distributed, mobile and ubiquitous nature of software systems.

The development of these software systems is a difficult task. Currently, decisions about these characteristics are usually postponed to late stages of the software life cycle (design and implementation). As a result, there is a loss of traceability, and the system is subject to a specific technological platform. Developers of distributed systems also have to spend more time programming than solving and considering distribution problems. The development of systems of this kind introduces important challenges such as: how to specify distribution and mobility features in a technology-independent way; how to consistently manage a distributed state; and how to support non-functional requirements such as security or fault tolerance.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1633 – 1649, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Model Driven Engineering (MDE) [1] is a software development approach based on the transformation among models. Model Driven Architecture (MDA) [2] is the MDE approach proposed by the Object Management Group (OMG). The MDA proposes the development of software by using Platform Independent Models (PIMs), to describe a system in a technology-independent way. Then, PIMs are transformed to a Platform Specific Model (PSM), which describes a system based on a specific technology. An application can be generated from the PSMs.

Component Based Software Development (CBSD) [3] and Aspect-Oriented Software Development (AOSD) [4] are two techniques that promise to achieve reusability, flexibility, and maintainability through the software development process. CBSD proposes to construct the system by connecting entities that provide and require services. AOSD allows the separation of concerns by modularizing crosscutting concerns in a separate entity called an aspect.

In this paper, we present a metamodel called Ambient-PRISMA that allows the definition of distribution and mobility characteristics in a technology-independent way. Ambient-PRISMA extends the PRISMA metamodel [5] with Ambient Calculus (AC) [6] concepts. PRISMA integrates CBSD and AOSD to specify software architectures. AC is a formalism that describes distribution and mobility properties in an abstract way.

In addition to the metamodel, Ambient-PRISMA uses the Aspect-Oriented Architecture Description Language (AOADL) [7] to allow the description of the distribution and mobile characteristics of software architectures at a PIM level. This work shows how a mobile agent case study can be described in a platform-independent way using the AOADL. A PSM is also proposed in order to support the generation of PIMs that are based on the Ambient-PRISMA metamodel to .Net technology.

The paper is structured as follows: Section 2 presents the state of art of distribution and mobility characteristics supported in Architecture Description Languages (ADLs). Section 3 gives an overview of AC. Section 4 presents the Ambient-PRISMA metamodel and describes how the distribution and mobility characteristics of a PIM can be specified using the AOADL. In addition, a PSM of Ambient-PRISMA in .Net technology is introduced. Finally, Section 5 presents conclusions and future works.

2 The State of Art of Distribution in Architecture Description Languages

Currently, few ADLs support the specification of distributed system properties. The first research that provided significant results in distributed software architectures was carried out in the Darwin ADL [8]. Darwin uses π-calculus [9] to define the semantics of distributed message-passing. It builds architectures by defining composite components that are given locations and bound at instantiation time. Darwin also has been used in the CORBA environment to specify the overall architecture of component-based applications [10]. However, in the literature, we have not found new advances to Darwin in constructing software architectures with mobile components. As Darwin is based on π-calculus only, mobility can only be simulated

by the movement of channels. It lacks primitives to express the movement of components that cross boundaries.

The work in [11] introduces features such as composition, reusability, and configuration that an ADL should consider in order to specify dynamic and distributed software architectures. It presents a configuration language that describes a method for a reconfiguration model at run-time. However, the reconfiguration model is not formal. Moreover, it neglects a distribution model that specifies distributed message-passing among components and connectors.

The works of Mascolo and Ciancarini [12] [13] introduce MobiS, which is a specification language that is based on a tuple-space model that specifies coordination by multiset rewriting. MobiS can also be used to specify architectures containing mobile components. However, it does not specify the mobility concern separately from the rest of the functionalities of the software architectures, reducing reusability and adaptability to changes.

The ADL C2Sadel has adapted a style to support both distribution and mobility. The style [14] provides software connectors that are able to move components. It also has an implementation infrastructure to support this architectural style. However, this approach has the drawback that there is no separation between coordination and distribution. Therefore, the components are the only architectural elements that are mobile and the connectors are static.

The work of Lopes in [15] describes the semantics of externalizing a distribution dimension to support distribution and mobility for software architectures. This distribution dimension is very similar to a connector, but instead of containing the business logic, it controls the rules for mobility and location. In this way, a separation between computation, coordination and distribution is achieved. A difference between our work and Lopes's work is that our work defines the semantics of distribution and mobility by using Ambient Calculus. This allows our approach to have an explicit primitive to represent a location with boundaries, which allows the specification of security and authentication.

3 Ambient Calculus

Ambient Calculus [3] (AC) is a process algebra that extends π-calculus [11] to introduce the concept of ambient. An ambient is a bounded place where computation occurs. Thus, an ambient can be anything with a boundary such as a laptop, a web page, a folder, etc. Each ambient has a set of running computations that can control it. These are responsible for moving an ambient. In addition, an ambient can contain other subambients that have running computations.

Thus, mobility is performed at an ambient level, i.e. ambients are mobile. Mobility is also performed by crossing ambient boundaries. AC provides mobility and local communication primitives. These primitives can be expressed in a textual syntax and in a graphical syntax which is called Folder Calculus [3] (see Figure 1). Folder Calculus is a graphical metaphor for AC where ambients are visually represented as folders. For example in Figure 2, the folders n and m represent ambients. In our work, we have chosen a visual notation similar to the one of Folder Calculus as it will be noticed in the paper.

AC uses some of the constructs inherited from π-calculus such as naming, restriction, parallel processes, inactive processes and replication. However, the names in AC are names of ambients instead of names of channels as in π-calculus.

Textual Syntax	Visual Syntax	Comments	Textual Syntax	Visual Syntax	Comments
$(\nu n)P$		New name n in a scope P.	0		Inactive process (often omitted).
			$!P$		Replication of P.
$n[P]$		Folder (ambient) of name n and contents P.	$\langle M \rangle$		Output M.
$M.P$		Action M followed by P.	$(n).P$		Input n followed by P.
$P \mid Q$		Two processes in parallel. (Visually: contiguously placed in 2D.)	(P)		Grouping

Fig. 1. The Textual and Visual Syntax of Ambient Calculus constructs taken from [3]

Some of the primitives that AC provides are called capabilities. Capabilities are actions that can be performed on ambients. There are three main types of capabilities: enter, exit, and open capabilities. The enter capability orders an ambient to enter another ambient on its same hierarchy level (see Figure 2). The exit capability orders an ambient to exit its parent ambient. The open capability dissolves an ambient leaving the processes that were in it.

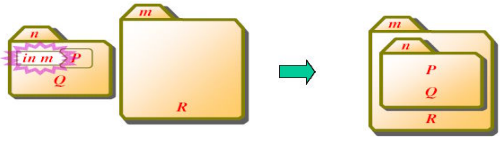


Fig. 2. Applying the enter capability to the ambient n

Cardelli proposed an algorithm for implementing ambient calculus in Java [16]. The implementation describes a tree structure where each node of the tree is an ambient. The tree structure is modified through operations that implement the enter and exit capabilities. However, this implementation has a limited application as it does not consider distributed nodes and has to be extended for distributed applications.

To date, we have only found a few implementations that support distributed software systems for ambient calculus. In [17], a scheme is proposed in C# for supporting ambient calculus primitives in distributed applications through an ambient runtime that sits above the .Net framework. Nevertheless, there has been no futher work to implement or apply the ambient runtime.

4 Supporting the Development of Mobile Ambients Through an Aspect-Oriented Software Architecture Model

In this section, we present the Ambient-PRISMA metamodel that combines ambient calculus primitives and Aspect-Oriented Software Architecture in order to allow the description of the distribution and mobility characteristics in a Platform-Independent way. Then, the AOADL will be used to describe the mobile and distribution characteristics in a case study of the mobile agent field. A PSM in C# will also be presented to support the generation of code from PIMs. The concepts of the AMBIENT-PRISMA are mapped to .NET.

4.1 Ambient-PRISMA Metamodel

A metamodel is needed to allow the definition of PIMs. A metamodel defines the ontology used to define a PIM. This section presents the metamodel of Ambient-PRISMA that allows the definition of PIMs of distributed and mobile software systems. The Ambient-PRISMA metamodel extends the PRISMA metamodel [5] by introducing concepts of the ambient calculus. The result of the Ambient-PRISMA metamodel is to allow the description of distributed and mobile system software architectures by using both the aspect-oriented and the component-based software development techniques.

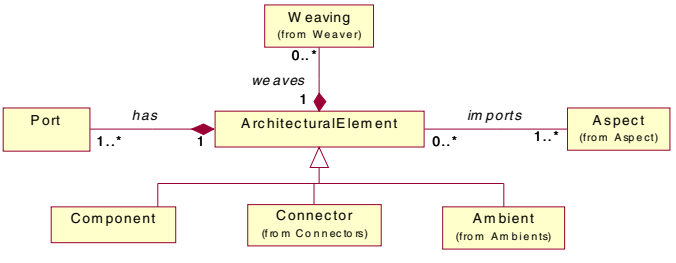
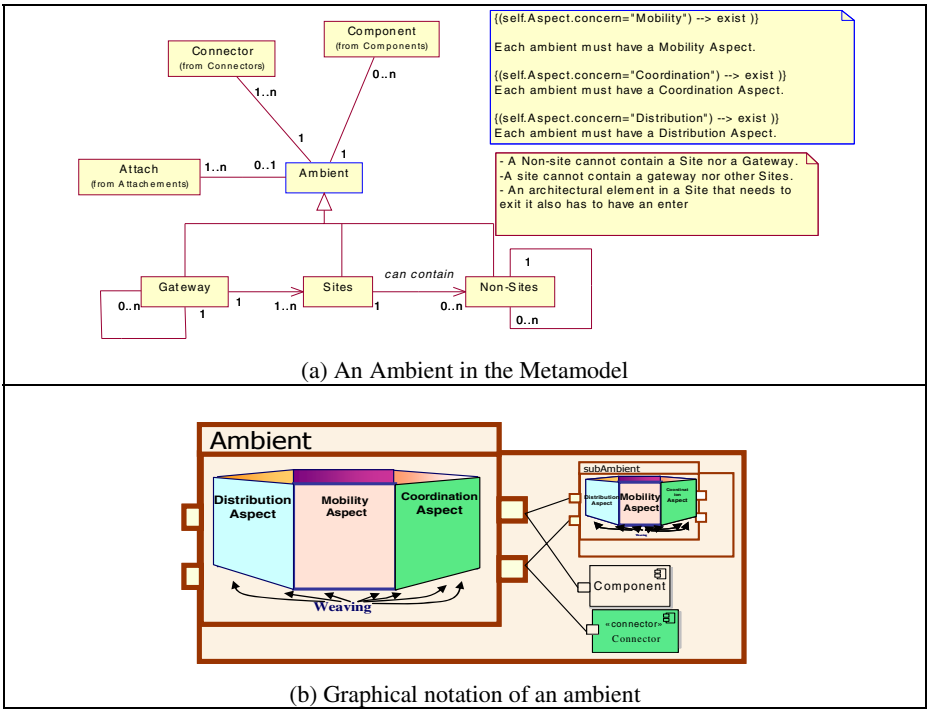


Fig. 3. Ambient as an aspect-oriented, component-based architectural element

The Ambient-PRISMA metamodel introduces the ambient concept as an architectural element (see Figure 3), which is responsible for providing mobility features to distributed architectural elements. As a result, an ambient inherits the CBSD and AOSD characteristics of PRISMA (as do other architectural elements of PRISMA such as components and connectors). The CBSD characteristics describe an ambient as a black box where it communicates with others by using ports that send and receive invocations of services. To enable the communication among architectural elements, channels called attachments are defined. Each attachment is defined by attaching two ports of different architectural elements.

In Ambient-PRISMA, ambients are complex architectural elements that represent the places where components, connectors and other ambients are located. Therefore, in the metamodel, the ambient is associated with the other architectural elements (see Figure 4(a)). For example, in Figure 4(b) the *Component* and *Connector* are located in



(a) An Ambient in the Metamodel

(b) Graphical notation of an ambient

Fig. 4. The Ambient package in the Metamodel

the same ambient. Also, an ambient can have other subambients. This allows the hierarchy of distributed and mobile systems to be modelled in Ambient-PRISMA. An ambient offers two type of services to architectural elements located in it: mobility services and distributed architectural element services. Architectural elements that need these services are connected to the ambient through attachments (lines in Figure 4(b)).

The AOSD view describes the PRISMA ambient with a set of aspects that can be weaved. Weavings indicate that the execution of an aspect service can trigger the execution of services in other aspects. Weavings are the glue of the aspects of an architectural element. This glue is defined using temporal operations called weaving operators. For example, if a weaving with the *after* operator is specified between service s1 of aspect A1 and service s2 of aspect A2, this means that s2 of A2 is executed after s1 of A1.

The ambient uses different aspects to specify the services it offers and requests. As Figure 4(a) shows each ambient must have a Mobility, a Coordination, and a Distribution Aspect. These constraints are specified in the Object Constraint Language (OCL).

As ambients are responsible for the mobility concern, all ambients must have the Mobility Aspect to provide mobility services to their local architectural elements. These services are the ones that provide the ambient calculus capabilities. The following are the Mobility Aspect functionalities:

- It allows an ambient to offer the exit service to the subambients that need to exit from it. (The specification of the AC exit capability).
- It allows an ambient to offer the enter service to the subambients that need to enter other subambients. (The specification of the AC enter capability).
- It allows an ambient to create subambients. (The specification of the AC restriction).
- It allows an ambient to accept a new ambient from external ambients.
- It allows an ambient to execute the open service. The open service allows a subambient to be destroyed without destroying its local architectural elements. As a result, the architectural elements of the destroyed subambient become part of the including ambient. (The specification of the AC open capability)

The separation of the Mobility Aspect concern from the rest of the concerns provides better maintainability of these functionalities because they are not scattered through the ambient specification. In addition, this is a generic aspect that must be reused by all PRISMA ambients. As a result, ambients are defined by importing the generic mobility aspect and adapting it to the needs of the software system through weavings. For example, a LAN ambient may need some security policies that are different from a PC ambient inside of the LAN. Therefore, both the LAN and PC ambient import the same Mobility Aspect, but the Mobility Aspect is weaved with different security aspects.

The coordination aspect of an ambient is needed to coordinate its architectural elements with the exterior. This coordination aspect receives calls from external architectural elements of the ambient (distributed calls) and redirects them to corresponding architectural elements of the ambient. It also receives calls from its architectural elements and redirects them to the exterior. This aspect, like the Mobility Aspect, is also a generic one that all ambients must reuse.

The Distribution Aspect of an ambient stores the name of its parent ambient. This aspect also specifies whether or not an ambient is mobile. A mobile ambient is an ambient that requests mobility services from its parent ambient. When an ambient is mobile, its parent ambient can be changed.

In Ambient-PRISMA, we have defined three types of ambients: Sites, Non-Sites, and Gateways (see Figure 4(a)). This separation is necessary since each one of them has its own semantics and constraints. Sites are ambients that represent devices; that is, they have a physical address. A Non-Site is an ambient that represents a collection of software elements. A Non-Site can only have subambients of type Non-Site. This restriction is important because Sites cannot be located in Non-Sites. Also, a Non-Site can be located either in a Site or in another Non-Site. An example of one use of a Non-Site ambient is to move a group of architectural elements together. Another example of a use of a Non-site is to separate certain architectural elements in a Non-site since they share some security protocols. A Gateway is a virtual ambient that groups either a collection of ambient Sites or other gateways. A Gateway ambient that consists of a collection of Sites represents the borders of a network (e.g. a Local Area Network). The modelling of a Gateway hierarchy can represent the hierarchy of LANs to form a Wide Area Network (WAN). Different gateway hierarchy levels can be represented in the software architecture depending on the software architect and on the distributed requirements. In Ambient-PRISMA there is always a default Gateway ambient that represents the Root.

4.2 Case Study: Mobile Agents in an Auction Site

Mobile Agents [18] are components that can move from one site to another in order to perform tasks in an autonomous way. Mobile agents are helpful as they solve problems that are encountered in distributed systems. Network bandwidth is reduced due to the fact that the agent is moved to a new site where it performs tasks locally instead of passing information through the network. Also, unreliable network connections are not a problem because once an agent has been sent to a site, the operations are locally performed and the results can be sent when a connection is re-established.

Mobile Agents can be applied in different applications. In this paper we consider an electronic commerce application: an auction site where products are auctioned. A customer of this auction site is interested in buying a specific product with certain specifications at a maximum price. To keep track of the new auctions being performed on the auction site the customer designs two mobile agents to take charge of the purchase. One of these agents is responsible for collecting information and the other agent is in charge of managing the purchase. The two agents have to collaborate with each other. The customer sends the agents to the auction site. The agents act for the customer. When the purchase is performed the agents return to their original location.

4.2.1 The PIM of the Case Study Specified in the AOADL of Ambient-PRISMA

To allow the definition of PIMs, a language has to be used. The language can be a graphical language such as UML or a textual language. A graphical language, which is user friendly, is an easy way to allow users to develop their PIMs. However, in order to formalize the semantics of a PIM, it is necessary to have a formal language. In this section we focus on our formal AOADL to describe the distribution and mobility characteristics of the PIM of the case study presented in section 4.2.

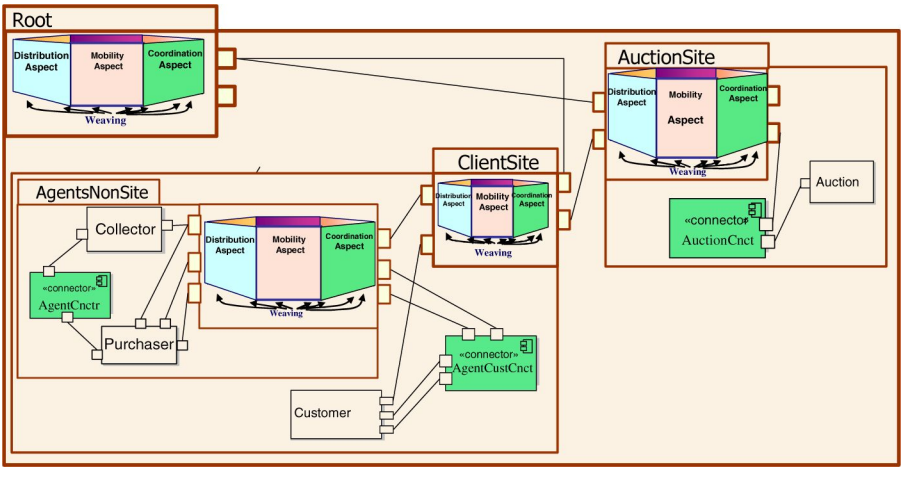


Fig. 5. The Initial Configuration of the Software architecture of the Mobile Agent Case Study

In the case study, the customer designs the agents: the *purchaser* and the *collector*. As the *purchaser* and the *Collector* need to collaborate with each other, a connector called *AgentCnctr* coordinates them. The customer locates these architectural elements in a Non-Site ambient called *AgentsNonSite*. The *Customer* software component is connected with the *AgentsNonSite* by a connector called *AgentCustCnct*. Initially, the *Customer* and the *AgentsNonSite* are located in the *ClientSite*. Figure 5 shows the configuration of the software architecture before the *AgentsNonSite* ambient is moved to the *AuctionSite*. In the *AuctionSite*, there is a component that represents the *Auction*. The *AuctionCnct* is a connector that coordinates the *Auction* with other architectural elements. As the *Auction* component can communicate with distributed architectural elements, the *AuctionCnct* has an attachment with *AuctionSite* ambient.

```
Distribution Aspect ANSDistribution using IMobility, ICapability
 Attributes
  location : loc;
 Services
   … …
   changeLocation(Name: String, NewLocation:loc)
        Valuations
          [changeLocation()] location:=NewLocation;
   getLocation( output Location:loc)
        Valuations
          [getLocation()] Location:=location;

   out exit (Name: String, Parent:loc );
   out enter (self.Name: String, NewAmbient: loc);
   … …
  Transactions move(NewAmbient: loc)
      move::= out exit(self.Name, location)→MOVE1;
      MOVE1::= {exitOK} out enter(self.Name, NewAmbient)→MOVE2;
 Preconditions
  in  changeLocation(Name, NewLocation)
                          if {self.Name==Name};
      … … …
End_Distribution Aspect ANSDistribution
```

(a)

```
Ambient_NonSite type AgentNonSite
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ANSDistribution;
 Ports
     ECapabilitiesPort: ICapability;
     EDistServicesPort: ICall;
     EMobilityPort: IMobility;
     InServicesPort: ICall;
     InCapabilitiesPort: ICapability;
     InMobilityPort: IMobility;
 End_Ports
… …
End Ambient_NonSite type AgentNonSite;
```

(b)

Fig. 6. Specification of the AgentNonSite ambient and its distribution aspect

The mobility of the *AgentsNonSite* is initiated when the *Customer* component requests it to move to the *AuctionSite*. Figure 6(b) shows the specification of the *AgentNonSite* in the AOADL. The *AgentNonsite* receives the mobility request through its port, called *EMobititiyPort*. This causes the execution of *move(NewAmbient:loc)* of the *ANSDistribution* Distribution Aspect that the ambient imports. Figure 6(a) shows the specification of the *ANSDistribution* aspect. The *move(NewAmbient:loc)* specifies that the *AgentNonSite* has to *exit* its parent ambient and then *enter* another ambient. The *enter* and *exit* services are served by the parent ambient of the *AgentNonSite*.

The *AgentNonSite* then requests the *exit(AgentNonSite, ClientSite)* from the *ClientSite* through the port *ECapabilitiesPort* (see Figure 6(b)). The *ClientSite* receives this petition through its port *InCapabilitiesPort* (see Figure 7(b)). As a consequence, the *Mobile* Mobility Aspect is executed.

Figure 8 shows a segment of the Mobility Aspect specified in the AOADL, which all ambients must reuse. It shows that the *exit* is a ***transaction*** that consists of a set of services. First, the *exit* checks if the requested element (*AgentNonSite*) is one of its children (is one of its local elements). Since the *AgentNonSite* is its child, the

```
Distribution Aspect SiteDistribution
 Attributes
  location : loc;
 Services
   … …
   getLocation( output Location:loc)
        Valuations
          [getLocation()] Location:=location;
…
End_Distribution Aspect SiteDistribution
```

(a)

```
Ambient_Site type ClientSite
 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect SiteDistribution;
 Import Security Aspect ClientSecurity;
 Ports
     ECapabilitiesPort: ICapability;
     EDistServicesPort: ICall;
     InServicesPort: ICall;
     InCapabilitiesPort: ICapability;
 End_Ports
 Weavings
     SiteDistribution.getLocation(Location) before
                                    Mobile.getParent(Parent);
   ……
End Ambient_Site type ClientSite;
```

(b)

Fig. 7. Specification of the ClientSite ambient and its distribution aspect in the AOADL

```
Mobility Aspect Mobile using ICapability
   …. ….
 TRANSACTIONS in exit(Requested: String, Ambient: loc):
  EXIT := out isChild(input Requested, output isChildOK)→
          EXIT1;
  EXIT1::= {isChildOK==true & self.Name==Ambient}
           getParent(output Parent)→EXIT2;
  EXIT2::= moving (Requested,Parent);

TRANSACTIONS in enter(Requested: String, NewAmbient: loc):
  ENTER := out areChildren(input Requested, input Ambient,
           output areChildrenOK)→ENTER1;
  ENTER1::= {areChildrenOK==true} moving(Requested, NewAmbient);

TRANSACTIONS in moving(Requested: String, NewAmbient: loc):
  ……..

TRANSACTIONS in accept(input NewAmbient:loc, input Caller: loc,
                input Child:String, input Type:String,
                input AttachmentsList[]:Attachments,
                output AcceptanceOK:boolean):
  ACCEPT::= out addChild(input Type, input Child,
                input AttachmentsList[], output AddedOK)
                {Acceptance==AddedOK}→ ACCEPT1;
  ACCEPT1::=out changeLocation(Child, self.Name);

Preconditions
       in accept(input NewAmbient, input Caller,
                 input Child, input Type,
                 input AttachmentsList[],
                 output AcceptanceOK) if
                                    {self.Name==NewAmbient};
 … …
End_Mobility Aspect Mobile
```

Fig. 8. A segment of an Ambient Mobility Aspect specified in the AOADL

ClientSite executes the *getParent* service in order to know who its parent ambient is. This service, as shown in Figure 7(b), has a ***Weaving*** with the *SiteDistribution* aspect. The service *getLocation* of Figure 7(a) is executed to provide the value of *location*. Figure 5 shows the *ClientSite* has the *Root* ambient as its parent. This is a default parent ambient for all sites and is of type *Gateway* as it is a virtual ambient. When the parent value is achieved, the *moving* transaction in Figure 8 is executed. Briefly, the *moving* transaction is in charge of removing the mobile architectural element (*AgentNonSite*) from the *ClientSite,* modifying its local attachments and ensuring that

the receptor (the *Root*) receives and adds it as its child. The result of the configuration after the execution of the exit is shown in Figure 9.

The *AgentNonSite* ambient then requests the *enter(AgentNonSite, AuctionSite)* to the *Root* ambient. The *Root* ambient also has the *Mobile* Mobility Aspect shown in Figure 8. The Root ambient checks if the *AgentNonSite* and the *AuctionSite* are both its children. If they are, then it executes the *moving* transaction. When the moving transaction is finalized, the *AgentNonSite* ambient becomes a child of *AuctionSite* as shown in Figure 10.

It could happen that *AgentNonSite* does not request an *enter* from the *Root*. As explained in Section 4.1, that a Gateway ambient can only have Site ambients or other gateways. Therefore, if the *AgentNonSite* ambient does not request an enter to another Site, the *Root* would send it back to the *ClientSite*.

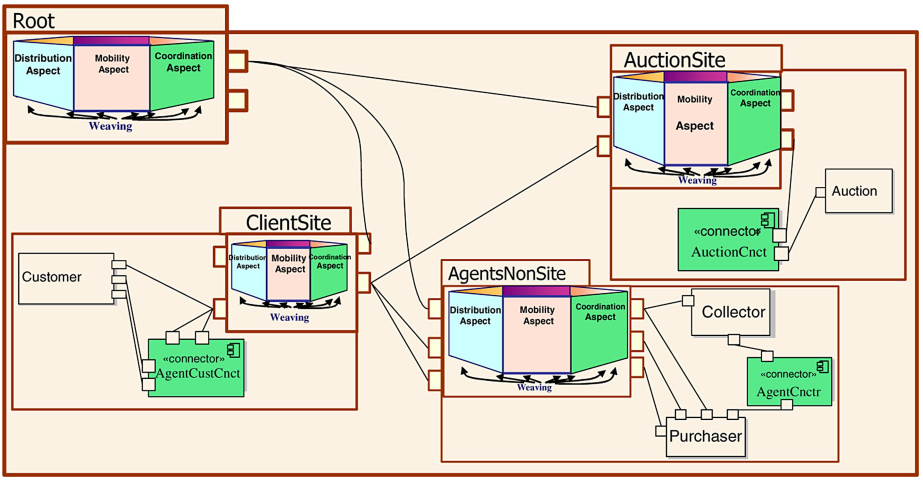


Fig. 9. The Software Architecture when the AgentNonSite executes the exit

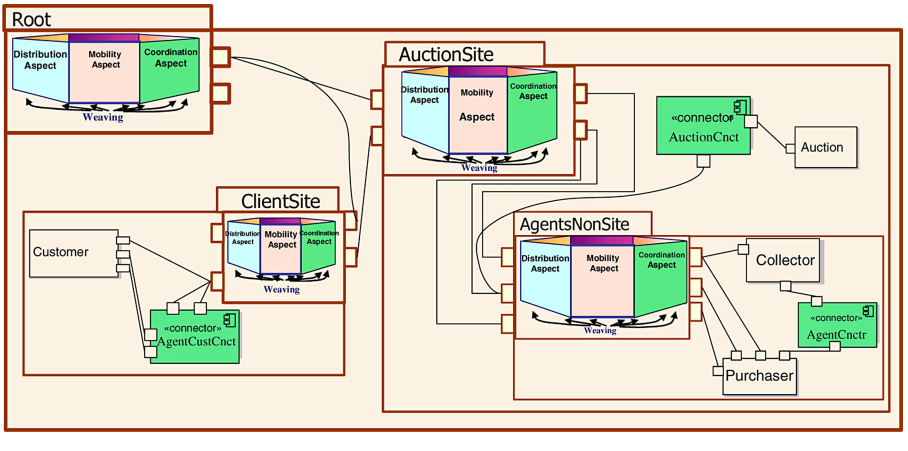


Fig. 10. The AgentNonSite becomes a child of AuctionSite

As Figure 10 shows, the *Customer* component can still send commands to the agents of the *AgentNonSite* in the *AuctionSite*. This is achieved because all the attachments between the *AgentCustCnct* and the *AgentNonsite* (see Figure 5) have been replaced by attachments between these elements and their ambients (see Figure 10). Thus, distributed communication is performed through ambients.

When the *purchase* agent in Figure 10 performs the purchase that the customer desires, it invokes the *move* of the *AgentNonSite* through the *InMobitityPort* (see Figure 6(b)). The entire process is repeated until the *AgentNonSite* returns to *ClientSite* (the architecture configuration returns to the configuration shown in Figure 5).

From the above example, we have seen how the distribution and mobile characteristics of a PIM, in this case of the Mobile Agent field, have been specified using an AOADL based on a defined metamodel.

4.3 Platform-Specific Model in .Net

The PRISMA metamodel is supported in .Net by the PRISMANET middleware [19]. The PRISMANET middleware provides an execution platform for PRISMA aspect-oriented software architecture configurations. For example, it contains classes that map the concepts of aspects, ports and attachments. Each PRISMANET middleware of a host can collaborate with PRISMANET middlewares on other hosts in order to provide a distributed runtime environment (see Figure 11). The runtime environment has a distributed Domain Name Server (DNS) where all middlewares on hosts can access. PRISMANET must be extended in order to provide the mobility and distributed communication of architectural elements that Ambient-PRISMA provides. Therefore, in this section a PSM is proposed for supporting Ambient-PRISMA based on PRISMANET.

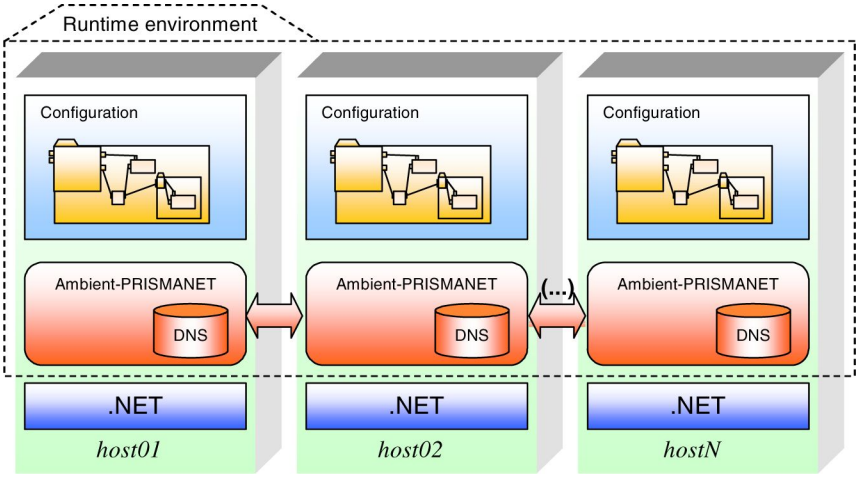


Fig. 11. Distributed Runtime Environment of PRISMANET

Ambient-PRISMANET extends the PRISMANET middleware in order to include the ambient architectural elements. Figure 12 shows the class *ArchitecturalElement* that includes the common functionality of all architectural elements. All architectural elements are composed of aspects, weavings, and ports. The classes *Component*, *Connector* and *Ambient* inherit from *ArchitecturalElement*. Using the PIM of the case study defined in section 4.2 the Collector, the Purchaser, the Customer, and the Auction are represented in C# as classes that inherit from the class *Component*. The AgentCustomerCnct, the AgentCnctr and the AuctionCnctr are classes that inherit from the class *Connector*.

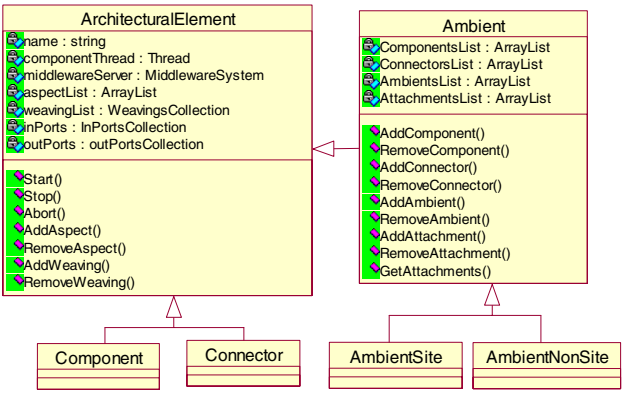


Fig. 12. The Ambient inherits from ArhitecturalElement class of PRISMANET

The class *Ambient* provides services for managing the information related to the architectural elements of an ambient. These services can change the state of an ambient by adding or removing either architectural elements or attachments. For example in the case study, when the AgentNonSite ambient requests to exit from the ClientSite ambient, the ClientSite has to replace the attachments between the AgentNonsite and the AgentCustCnctr with attachments between the AgentCustCnctr and the port of the ClientSite that provides distributed communication. Therefore, an ambient needs services that facilitate the dynamic generation of attachments depending on its state.

In addition, all ambients must have a Mobility Aspect, a Coordination Aspect and a Distribution Aspect. The Mobility and Coordination Aspects of an ambient must be reused by all ambients. Thus, they are predefined and the designers do not have to modify. As these aspects have a predetermined definition, they are included as specific types of aspects that inherit from the class *AspectBase* (the class that represents a PRISMA aspect). Figure 13 shows the class *MobileAspect* that maps the Mobile Aspect and *ACoordinationAspect* that maps the ACoordination Aspect. They are marked with the C# sealed modifier in order to represent that they cannot be extended by the user. The class *MobileAspect* publishes its interface through its port in order to offer the mobility services to its architectural elements. The interface of the predefined Mobility Aspect is shown in Figure 14.

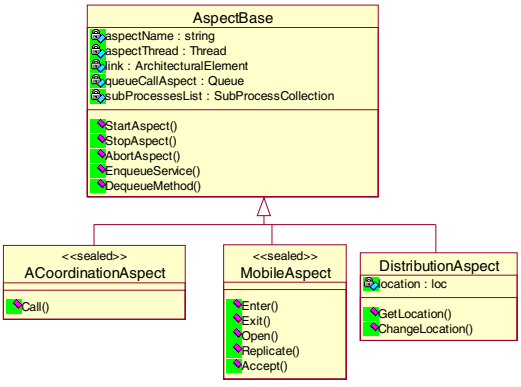


Fig. 13. The Aspects of an ambient

```
   public interface ICapability
   {
   AsyncResult Enter(string AmbientName);
   AsyncResult Exit(string AmbientName);
   AsyncResult Open(string Ambient);
   AsyncResult Replicate(string Ambient);
   AsyncResult Accept(string grandparent, string parent, string type,
string children, ArrayList attachmentsList, out bool acceptanceOK);
   }
```

Fig. 14. The interface ICapability of the Mobility Aspect in C#

On the other hand, the distribution aspect of an ambient can be extended by the user. As a result, a class *DistributionAspect* that also inherits from *AspectBase* has been created. The class *DistributionAspect* implements the minimum functionality of a distribution aspect. That is, it stores the reference of the ambient location and the services needed for managing it. For example, the *ANSDistribution* aspect of Figure 6(a) inherits from class *DistributionAspect* extending it with the mobility requests.

Moreover, the classes *AmbientSite* and *AmbientNonSite* have been introduced by inheriting from class *Ambient* (see Figure 12) in order to represent Ambient-PRISMA Site and Non-Site ambients repectively. Each of these classes expresses the restrictions of the type of ambient they represent. For example, AgentsNonSite ambient must inherit from the class *AmbientNonSite* to be implemented. In the same way, ClientSite and AuctionSite ambients inherit from *AmbientSite* to be implemented.

Another Ambient-PRISMA concept that has to be treated is the virtual Gateway Ambient. At implementation this ambient has been decided to be the set of middlewares that collaborate to form the borders of the distributed runtime environment. In this way, the Ambient-PRISMANET middleware has a layer called DistributionLayer that is in charge of the distributed communication among hosts and the mobility of architectural elements between hosts.

In the case study, when the ClientSite needs to execute the exit requested by the AgentsNonSite ambient, the ClientSite has to call the Root ambient in order to request it to accept the AgentsNonSite. This call is mapped to a request that the ClientSite performs to the DistributionLayer of the middleware at its host. As a result, the DistributionLayer stores the state of the AgentsNonSite ambient and its associated attachments. When, the AgentsNonSite exits the ClientSite, at implementation the AgentsNonSite is neither part of the ClientSite nor part of the AuctionSite. When the AgentsNonSite executes the enter to the AuctionSite, the DistributionLayer fetches in the distributed DNS to find out where the AuctionSite exists in order to create a proxy to the DistributionLayer of the middleware where the AuctionSite exists (the receptor). Then, the DistributionLayer of the middleware where the ClientSite exists (the sender) starts the serialization of the AgentsNonSite ambient and its contents. When the DistributionLayer of the receptor receives the serialized ambient, it calls the accept of the AuctionSite ambient. In this way, the DistributionLayer of the receptor acts as the local view of the parent of the AuctionSite (the Root Gateway ambient).

The middleware establishes a timeout from the moment that the AgentsNonSite exits the ClientSite ambient. If the timeout expires, the mobility process is considered to be failed. This failure is due to the fact that a Non-Site ambient can only stay in this intermediate state for requesting an enter to a Site ambient. It is important to mention, that all the operations related to mobility are transactional. This implies that once a failure occurs at any point of the transaction, the operations performed must be undone in order to return to the state previous to the mobility process, i.e. returns the AgentsNonSite to the ClientSite.

5 Conclusions and Further Work

In this paper, we have shown how ambient calculus can be used in software engineering to develop distributed and mobile applications. These applications, which are based on ambient calculus, can be developed using AOSD and CBSD techniques. The concepts of ambient calculus have been introduced into an aspect-oriented and component based metamodel (PRISMA) in order to achieve Ambient-PRISMA. Ambient-PRISMA provides the construction of distributed and mobile software systems by reusing and adapting specifications.

Ambient-PRISMA has been able to specify the distributed and mobile characteristics of the software architecture of a mobile case study. This has been done in a platform-independent way through MDE. In order to execute distributed and mobile applications based on Ambient-PRISMA, a PSM has been proposed. This PSM supports the implementation of ambient calculus in C#, with the characteristic that it preserves the traceability of the concepts of Ambient-PRISMA.

In the near future, we are going to take advantage of the ambient calculus primitives concerned with security. These primitives will be specified using our AOADL so that the case study presented in this paper could be extended to incorporate security properties. In addition, we are going to introduce the concepts presented in this paper into the PRISMA tool to be able to model and execute mobile distributed software architectures in an intuitive way. This will be done in three

stages: first, a complete implementation of the concepts of Ambient-PRISMA will be introduced in the PRISMANET middleware [19] to execute these concepts; second, ambient graphical metaphor and code templates will be introduced in the modelling framework in order to automatically generate code from the graphical metaphor; third, the implementation will be validated by modelling and executing a complex, distributed, and mobile case study.

References

[1] Schmidt, D.C. "Model-Driven Engineering". IEEE Computer 39 (2), February, 2006.

[2] OMG, 2003. MDA Guide, http://www.omg.org/docs/omg/03-06-01.pdf

[3] Szyperski, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.

[4] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development, ISBN: 0321219767.

[5] Perez, J., Ali, N., Carsí, J.A., Ramos, I. "Dynamic Evolution in Aspect-Oriented Architectural Models", European Workshop on Software Architecture, Pisa, June 2005 © Springer LNCS vol n.3527.

[6] Cardelli, L. "Abstractions for Mobile Computation." In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94.

[7] Pérez, J., Ali, N., Carsí, J.A., Ramos, I. "Designing Software Architectures with an Aspect-Oriented Architecture Description Language", 9th International Symposium on Component-Based Software Engineering (CBSE),Västerås , Stockholm, Sweden, LNCS, June 29th -1st July 2006 (accepted to appear)

[8] Magee, J., Dulay, N., Eisenbach, S., Krammer, J. "Specifying Distributed Software Architectures". 5th European Software Engineering Conference (ESEC 95), Sitges, Spain, 1995, pp 137-153.

[9] Milner, R., Parrow, J., Walker, D. "A calculus of mobile processes", Parts 1-2. Information and Computation, 100(1), 1992, pp. 1-77.

[10] Magee, J., Tseng, A, Kramer, J. "Composing Distributed Objects in CORBA", Third International Symposium on Autonomous Decentralized Systems, Berlin Germany, 1997, pp 257-263.

[11] Virginia C. de Paula, G.R., Justo, Cunha, Ribeiro, P. R. F. "Specifying Dynamic Distributed Software Architectures", XII Brazilian Symposium on Software Engineering, BCS Press, October, 1998.

[12] Ciancarini, P., Mascolo, C. "Software Architecture and Mobility", 3rd Int. Software Architecture Workshop (ISAW-3), November, 1998.

[13] Mascolo, C. "MobiS: A Specification Language for Mobile Systems". 3rd International Conference on Coordination Models and Languages, 1999.

[14] Medvidovic, N., Rakic, M. "Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility". Software Engineering and Mobility Workshop, Toronto, Canada, May 2001.

[15] Lopes, A. Fiadeiro, J.L., Wermelinger, M. "Architectural Primitives for Distribution and Mobility", 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41-50.

[16] Cardelli, L. " Mobile Ambient Synchronization" SRC Technical Note Report 1997-013, Digital SRC, 1997.

[17] Vasudeva, E., Joshi, Rushikesh K. “A Scheme for Implementing Ambient Calculus over an ARC Framework”, Software Design and Architecture Workshop, Dec 2004, IIIT Bangalore (position paper).

[18] Chess, D., Harrison, C., Kershenbaum, A. "Mobile Agents: Are They a Good Idea?" IBM Research Report RC.

[19] Perez, J., Ali, N., Costa, C., Carsí, J.A., Ramos, I. “Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology”,3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005 , 2005

Component QoS Contract Negotiation in Multiple Containers

Mesfin Mulugeta and Alexander Schill

Institute for System Architecture
Dresden University of Technology, Germany
{mulugeta, schill}@rn.inf.tu-dresden.de

Abstract. The explicit consideration of component contracts aims at simplifying the development of component-based applications with non-functional requirements like QoS and security, but it is also a challenging task. QoS contract negotiation is used to select concrete QoS contracts between the collaborating components. This paper presents an algorithm for the QoS contract negotiation of components deployed in multiple component containers. Our algorithm addresses: possible resource constraints at each node and the network, the efficiency of the negotiation process, the selection of a heuristically optimized solution, and over-constrained cases. As a basis to our approach, we used the notion that the required and provided non-functional properties as well as resource demand are specified at the component level. To demonstrate the presented ideas, the interaction of a customer, video provider, and payment provider example scenario is analyzed.

1 Introduction

Component-based software engineering allows the composition of complex systems and applications out of well-defined parts. Nowadays, several mature and commercial component models (e.g. EJB, JavaBeans, .NET, COM+, etc.) exist in the market. However, they provide only limited support for the development of components and applications with Non-Functional Properties (NFPs) like QoS and security.

In order to address the deficiencies of these mainstream component technologies, a lot of research has been and is currently being done. These efforts range from creating new component models [7] to extending the existing mainstream component models [18][16][17]. In realizing all these, various issues have to be addressed like the specification of component contracts, resource management, negotiation, adaptation, etc. Our work focuses on *contract negotiation*. Before highlighting the challenges in this area, we first give a brief note on component contracts and the specification languages our work applies.

Components in EJB, .NET, and COM+, are specified only with interfaces that provide *syntactical* information about which methods are available and how to invoke them. A contract greatly extends the component specification precision [19]. The term contract (in relation to components) can very generally

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1650–1667, 2006.
© Springer-Verlag Berlin Heidelberg 2006

be taken to mean "component specification" in any form [3]. Four different levels of contracts have been identified for components in [2]. These are: syntactic, behavioral, synchronization, and QoS contracts. Our focus in this paper is on the fourth type of component contract, which is the QoS contract.

A component's QoS contract is differentiated into *offered QoS contract* and *required QoS contract* [9]. The *offered QoS contract* specifies the quality values that a component can provide to its clients (other components) while the *required QoS contract* specifies the QoS constraints that the servers of a component must achieve. There is a dependency between offered and required QoS contracts. There exist different description languages for specifying the QoS contract of components [5] [1]. CQML$^+$[11], which is an extension of CQML [1], provides the construct *QoS-profile* that is used in specifying a component's offered QoS contract, required QoS contract, and its resource demand. The notion of components with different QoS-profiles, which this paper applies, has been employed in the research project COMQUAD [7,15] at TU Dresden.

A component-based application requires the appropriate composition of the contracts at the different ports of the components. The ports must be properly connected so that the QoS level required by one must be matched by the QoS level provided by the other. This requires the selection of appropriate QoS contracts at each port. When QoS contracts are known statically, the developer or assembler can select the right concrete QoS contracts of each component and compose the whole application during design, implementation, or deployment time. But, for composing QoS contracts which depend on runtime resource conditions (e.g. network bandwidth, CPU) or quality attributes fixed dynamically, the selection of appropriate QoS contracts must be carried out at runtime through the process of contract negotiation. In our approach, the runtime environment, in particular the component container, performs the selection of the appropriate concrete QoS contracts for the components at runtime.

We tackle the QoS contract negotiation problem by first modeling it as a constraint satisfaction problem. Based on this, we propose an algorithm that selects concrete QoS contracts for components distributed in multiple containers. Some of the issues that we address are: (i) resource constraints at each node and the network, (ii) efficiency of the negotiation process, and (iii) over-constrained cases. We treat efficiency from two different aspects. The first is from the point of view of the selection of QoS contracts that heuristically optimizes the solution for meeting the negotiation goal (e.g. maximizing user's satisfaction). The complexity of the selection process increases exponentially with the number of components and the QoS contracts they provide. The second is from the standpoint of the number of inter-container message exchanges and the level of concurrency that can be achieved in the negotiation process.

The paper is structured as follows. Section 2 details our modeling of the contract negotiation as a Constraint Satisfaction Problem. In Section 3 we discuss the proposed approach by providing a heuristic algorithm for the negotiation, a discussion on over-constrained situations, and a comparison on three

architectures for realizing the algorithm. Section 4 demonstrates the presented ideas by taking an example scenario. The paper closes with an examination of related work, a summary, and outlook to future work.

2 QoS Contract Negotiation as a Constraint Satisfaction Problem (CSP)

Before formalizing the QoS contract negotiation as a CSP, we would describe first the notion of conformance [5] that should exist between QoS-profiles. A conformance between two QoS-profiles exists when the constraints in the QoS contract of the server component satisfies the constraints in the QoS contract of the client component. For e.g., the constraint $delay < 5$ conforms to the constraint $delay < 10$. When the components are distributed across containers, the relationship must take into account the influence of the network and containers between the interacting components as demonstrated in Section 4.

The objective of the QoS contract negotiation is the selection of QoS-profiles so that the composed application meets the user's QoS requirements and preferences. As a goal, the negotiation aims at finding a near-optimal solution (heuristic optimization) among the set of possible solutions.

A CSP consists of n variables $x_1, x_2, ..., x_n$ whose values are taken from finite, discrete domains $D_1, D_2, ..., D_n$ respectively, and a set of constraints on their values. In general, a constraint is defined by a predicate. That is, the constraint $p_k(x_{k1}, ..., x_{kj})$ is a predicate that is defined on the Cartesian product D_{k1} X ... X D_{kj}. This predicate is true iff the value assignment of these variables satisfies this constraint [20]. Solving a CSP is equivalent to finding an assignment of values to all variables such that all constraints are satisfied.

For the formalization, we take the variables to be the QoS-profiles of the collaborating components $C_1, C_2, ..., C_n$. There are n variables - $P_1, P_2, ..., P_n$ - each representing QoS-profiles of $C_1, C_2, ..., C_n$ respectively. The domain of each variable is the set of all QoS-profiles provided for a component. The constraints are classified as *conformance*, *user's* and *resource*. As explained earlier, a QoS-profile P_i defines the offered and required QoS contracts and the corresponding resource demand of a component, which we would designate as: $P_i.Offered$, $P_i.Required$, and $P_i.Resources$ respectively.

In the offered and required QoS contracts, one or more QoS characteristics (e.g. $delay$, $frameRate$) are constrained with values (e.g. $delay < 5$ or $frameRate = 15s^{-1}$). Suppose $d_1, d_2, ..., d_k$ are the QoS characteristics considered, the conformance constraint between P_i and P_j is valid if there is conformance between each corresponding QoS characteristics in P_i and P_j. We designate the conformance as ($P_i.Required.d_j => P_j.Offered.d_j$). This conformance constraint can apply to a single method of the interface implemented by the components or to the entire methods. This has not been distinguished for the purpose of clarity of the presentation.

As an example, for the application that involves three components (Fig. 1), the QoS contract negotiation is formalized as in Table 1, where all the

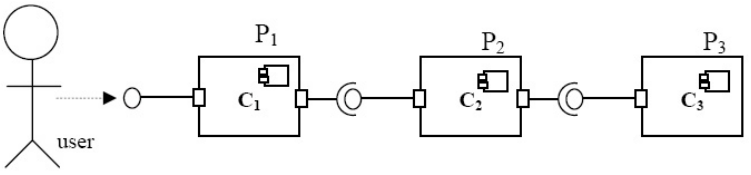


Fig. 1. Example Component-Based Application

Table 1. Component QoS contract negotiation as a constraint satisfaction problem

Variables	P_1, P_2, and P_3
Domains:	for each component, a set of QoS-Profiles are specified by the component developer
User's constraint	user's QoS Requirement on d_1 > P_1.Offered.d_1 … user's QoS Requirement on d_k > P_1.Offered.d_k
Conformance constraint	(P_1.Required.d_1 => P_2.Offered.d_1) … (P_1.Required.d_k => P_2.Offered.d_k) (P_2.Required.d_1 => P_3.Offered.d_1) … (P_2.Required.d_k => P_3.Offered.d_k)
Resource constraint	P_1.Resources + P_2.Resources + P_3.Resources ≤ $Resources_{avail}$

components are assumed to be deployed in a single container. For multiple containers, the conformance and resource constraints must be modified as follows. The influence of the network and the containers must be incorporated in the conformance constraint for those components connected across containers. For example, for *response time (RT)* property, the constraint can be modified as: ($P_i.Required.RT$ => $Pj.Offered.RT$ + *delay in network and containers*), where *delay in network and containers* is assumed to be a constant for the period the negotiation agreement is valid. In the resource constraint, for two containers case, instead of only one relation, three relations must be provided. These are for components deployed on the client, on the server, and for components connected across containers.

Resources considered in P_i could be CPU, memory, and network bandwidth. The resource constraint is based on the following assumption. Suppose there are n components - $C_1, C_2, ..., C_n$ - deployed in a container and these components need m different resource types. Let R_i be the resource demand of C_i, R_{all} is the resource demand of the n components, and R_{avail} is the available resource. That is,

$R_i = [r_{i1}, r_{i2}, ..., r_{im}]$ where r_{ij} is the j^{th} resource demand of C_i

$R_{avail} = [r_1^{avail}, r_2^{avail}, ..., r_m^{avail}]$

$R_{all} = R_1 + R_2 + ... + R_n$

$R_{all} = [r_{11} + r_{21} + ... + r_{n1}, r_{12} + r_{22} + ... + r_{n2}, ..., r_{1m} + r_{2m} + ... + r_{nm}]$ where the addition used is arithmetic

We require there are enough resources, and hence the resource constraint is satisfied, if:

$r_1^{avail} \geq r_{11} + r_{21} + ... + r_{n1}$,
$r_2^{avail} \geq r_{12} + r_{22} + ... + r_{n2}$,
...
$r_m^{avail} \geq r_{1m} + r_{2m} + ... + r_{nm}$

There are enough resources for the n components iff for each type of resource, the resource requirement is not larger than the corresponding available resource.

3 The Proposed Approach

To find a near-optimal solution for the resulting CSP (Table 1), in a *naive approach*, all possible configurations (assignments) must be considered. Then, QoS-profiles that give the highest utility and where all constraints are met are selected. This approach requires exhaustive searching of all possible configurations. In general, constraint satisfaction is NP-hard [20]. For applications with few possible configurations, the naive approach can be used. In other cases, *heuristics* must be applied to improve the efficiency of the naive approach. Next, we discuss how we have tailored general-purpose heuristic mechanisms and our own problem-specific heuristics.

3.1 Variable and Value Ordering

A variable and value ordering is a general purpose heuristics used to solve CSPs efficiently [12]. In this method, the variable to assign next is appropriately selected. Having selected a variable, the value to assign to must also be appropriately chosen. We have tailored this method as we seek a near-optimal solution.

The variables (QoS-profiles) are ordered for assignment by topologically sorting the network of cooperating components. We call the minimal element the front-end component (e.g. C1 in Fig. 1), which is also the one that interacts with the user. The assignment starts from this minimal element. At any point in time, the QoS property of the partially completed assignment can be taken as the *offered QoS contract* of the front-end. This is one advantage achieved by this ordering as we know right from the start and during successive assignments whether or not a partial assignment meets the user's constraint. The values of each variable, i.e. the QoS-profiles of each component, must be ordered from higher to lower quality. As contracts involve multiple QoS properties, the ordering is based on first the QoS property that is the most preferred by the user, then the next preferred, etc.

Using a centralized approach, we can find the near-optimal solution by considering all the components (in both containers) as a whole. Starting from the front-end component and its profile at the top of the list, we continue selecting QoS-profiles in the topologically ordered set. In doing so, if we cannot get

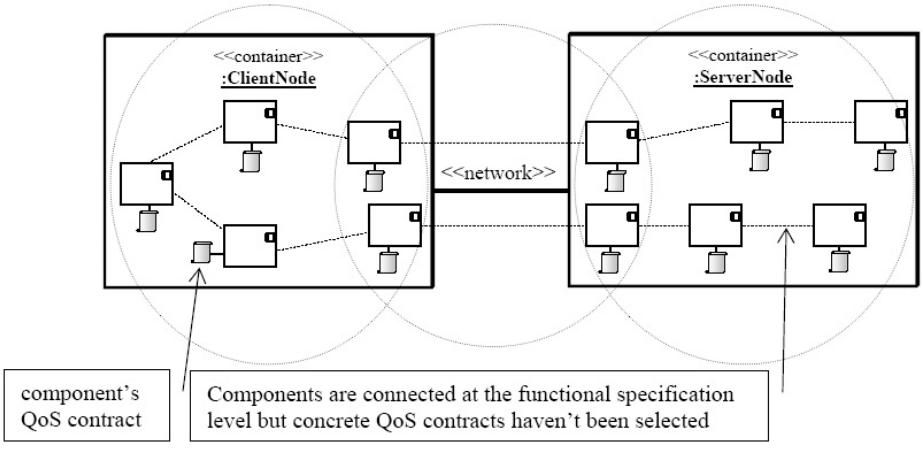


Fig. 2. Components deployed in a client and server Container and groupings based on resorce constraint

conformant profiles, then we have to backtrack. After exhausting all the components, we check if the resource constraints are met. If so, this will get us the near-optimal solution.

Because of the ordering of all profiles from higher to lower quality, the first solution obtained is the near-optimal one. However, the ordering may not necessarily be a heuristics for finding *any* acceptable solution (but this was not our aim!) as properties like smallest domain size, maximum degree, etc. help in speeding up search in CSPs. But the latter approach would require us to exhaustively look all possible solutions in order to get the one which is near-optimal.

3.2 Structure of the Problem

In the above, we tried to find the near-optimal solution by considering all the components as a whole. But, generally breaking a problem into sub-problems helps in finding the solution quickly. In the two container case, the problem can be divided into three sub-problems as shown by the categorization of components in Fig. 2. This subdivision is logical as separate resource constraints apply to each group. We refer the groups as: *components on the client*, *components connected across containers*, and *components on the server*.

An algorithm that combines the structuring of the problem with the variable and value ordering heuristics can be given as follows. For components on the client, the client container selects *appropriate* QoS-profiles. In the next steps, based on the already selected profiles, *appropriate* profiles for components across containers and components on the server are selected by the server container. This algorithm resembles the synchronous backtracking algorithm in [20].

There are two shortcomings of this algorithm: (i) it would lead to more backtrackings if the *bottleneck resource* were either the network bandwidth or server resources, and (ii) the method is sequential even though the problem has been

decomposed into sub-problems and lends itself for concurrent executions. We define a *bottleneck resource* as the first limiting resource in a step-by-step (from lower to higher quality) selection of QoS-profiles, which aims at finding a near-optimal solution. An algorithm that avoids the above stated shortcomings will have to first identify the location of the bottleneck resource and first selects QoS-profiles that maximally use this resource. The algorithm we propose (Listing 1.1), thus, determines the bottleneck resource first and refines the solution when more QoS-Profiles are available.

Listing 1.1. QoS contract negotiation algorithm for components deployed in client and server containers

```
enum ComponentGroup {On_Client, On_Server, Across_Containers}
/* On_Client, On_Server, and Across_Containers refer to components
 on the client, components on the server, and components connected
 across containers respectively */

void BuildSolution()
{
betterSolution = true;
while(betterSolution){
   if(FindConformantProfiles(On_Client)) {
      if(CheckResourceConstraints(On_Client)) {
         if(FindConformantProfiles(Across_Containers\On_Client)){
            if(CheckResourceConstraints(Across_Containers)) {
               if(FindConformantProfiles(On_Server\Across_Containers) {
                  if(CheckResourceConstraints(On_Server)) {
                     CommitSelectedProfiles();
                     continue;
                  }
                  else { // Bottleneck is Server
                     betterSolution = false;
                     SetPreviousSelection(On_Server);
                     DoFurtherNegotiations(On_Client&Across_Containers);
                  }
               }
               else { // no conformance for components_on_server
                  // backtrack to previous FindConformantProfiles()
                  }
            }
            else { // Bottleneck is Bandwidth
               betterSolution = false;
               SetPreviousSelection(Across_Containers);
               DoFurtherNegotiations(On_Client\Across_Containers);
               DoFurtherNegotiations(On_Server\Across_Containers);
            }
         }
         else { // no conformance for components_across_containers
            // backtrack to previous FindConformantProfiles()
         }
      }
      else { // Bottleneck is Client
         betterSolution = false;
         SetPreviousSelection(On_Client);
         DoFurtherNegotiations(On_Server&Across_Containers);
      }
   }
   else break; // no conformance for components_on_client
}
```

Listing 1.1 provides an algorithm that finds QoS-profiles that fulfill the user's, conformance, and resource constraints, if a solution exists and *No Solution* otherwise. To determine whether or not a solution exists, a list is used by the algorithm. Let the name of this list be *selectedProfiles*. After the execution of `BuildSolution()`, *selectedProfiles* contains the selected profiles if a solution exists, or it is empty for *No Solution*. *No Solution* occurs when the algorithm exits before executing `CommitSelectedProfiles()`. The categorization of components is shown as enumeration type `ComponentGroup`. Most of the functions invoked in Listing 1.1 take a type of `ComponentGroup` as input argument. `FindConformantProfiles(.)` finds QoS-profiles, which are conformant to one another for all the components specified in the input argument. At each iteration `FindConformantProfiles(.)` improves the solution by one step based on the specified QoS-profiles. `CheckResourceConstraint(.)` checks whether there is enough resource for the current selection. `DoFurtherNegotiation(.)` applies the same approach of finding the solution, for a subset of components while QoS-profiles have already been determined for other components. During intermediate steps of the algorithm, QoS-profiles are temporarily selected. We say, a possible solution is obtained when selections are made for all components. This selection is saved on *selectedProfiles* with `CommitSelectedProfiles()`. After temporarily selecting profiles and when it is found out that the partial solution would not lead to a solution, the temporary selection would be rolled back to the previous selection with `SetPreviousSelection(.)`.

A component may belong to two groups in `ComponentGroup`. For example, a component deployed on the client container and that also communicates across containers belongs to `On_Client` and `Across_Containers`. The notation \ in Listing 1.1 is read as "less" and & is read as "and".

3.3 Dealing with Over-Constrained Situations

The algorithm in Listing 1.1 has been modeled with hard constraints. The output is always expected to give a solution that meets all the constraints. *No Solution* will be provided, if at least one constraint is violated. This is inflexible as far as the user is concerned, who may prefer to get a solution, though a degraded one. In some instances, it may be difficult to find a solution that meets all constraints whatever the user's requirement may be. Under all these situations, the CSP is said to be over-constrained. Next, we discuss how to extend the algorithm in Listing 1.1 to handle some of the over-constrained cases.

One of the approaches used to deal with over-constrained CSPs is to extend the CSP framework into what are called *Partial Constraint Satisfaction Problems* [4]. Maximal constraint satisfaction algorithms, which seek a solution that satisfies as many constraints as possible, have been given in [4]. The algorithms compare potential partial solutions based on the number of constraints violated and choose the one that results in a minimum number of constraint violations. These are some general metrics and the nature of the problem at hand should dictate the strategy to be used in finding the partial solution.

In our case, we have three different types of constraints: user's, conformance, and resource. The conformance and user's constraints apply to each QoS property. It is difficult to compare violations of resource constraints with conformance constraints and hence simply counting the violated constraints is not applicable. In search of a partial solution, which is achieved by weakening the original problem, it is important to see which constraints to tolerate for violation. Allowing either resource or conformance constraints to be violated would make it difficult to predict the QoS property of the whole application. Currently, we are following the strategy where the user's constraint is systematically violated to reach to a solution. User's constraint relaxation helps in finding a solution when, (i) resource constraints cannot be met although a configuration that meets the other constraints exists, and (ii) the conformance constraints cannot be met when searching for a solution.

For the non over-constrained problem, the algorithm in Listing 1.1 improves a solution step-by-step by going one level up in the available QoS-profiles. The relaxation step in the over-constrained case uses a similar approach to get a near-best solution. It successively goes one step down on the list of QoS-profiles (which are sorted based on the user's preferences) of the front-end component to degrade the user's QoS requirements. The algorithm in Listing 1.1 is then applied to the weakened problem. If this step also produces *No Solution*, the user's QoS requirement is relaxed further and the above process is repeated. This step runs until a solution is obtained or the user's requirement can't be further relaxed.

The assumptions we have made concerning how user's requirements and preferences are captured are: a user can define the relative weights of the different QoS properties; and a user can define the minimum quality of the output below which it is unacceptable. This is done corresponding to each QoS property. A more robust user interface can be designed but it is beyond the scope of this paper. It should however be emphasized that it is essential not to make the user interface complex when trying to make it more robust. One possible user input mechanism we are currently studying to incorporate is the prioritization of a user towards the QoS properties. Out of the n QoS-dimensions considered in the application, the user may be interested in only m (where $m < n$) properties. This might imply, the negotiation can continue even if there are contract violations in the $n - m$ QoS properties.

3.4 Centralized Versus Distributed Solution

We have identified three approaches for implementing the algorithm in Listing 1.1: (i) centralized solution - only one of the containers makes the selection of QoS-profiles, (ii) distributed solution - both containers are responsible for the selection of appropriate QoS-profiles, and (iii) hybrid solution - combines centralized and distributed approaches.

In the centralized solution, the responsible container must have access to the configuration information of all deployed components and the resource availability at each container. The distributed solution doesn't require this as each

container is responsible for the selection of profiles of components it hosts and communicates with the other container to resolve constraints that concern both of them. In the hybrid solution, the bottleneck resource is first isolated using a centralized approach so as to minimize the number of interactions (message exchanges) between the distributed containers and then distributed approach is applied to harness concurrent executions.

Owing to the nature of our problem (which is not loosely coupled), a pure distributed solution results in large inter-container communication overhead and hence we do not consider it as an option. To make comparisons between the centralized and hybrid solutions, we divide the time of negotiations into two phases. Phase 1 is the initial negotiation process where contracts are to be established for the first time. Phase 2 is for re-negotiations that could take place after contracts have already been established. The number of inter-container communication and level of concurrency are taken as measures for the comparison.

For both the centralized and hybrid solutions, inter-container communication is required when: the QoS-profiles of interacting components and local resource conditions are exchanged just before the negotiation. The processor and other pertinent information of each node need to be exchanged together with the QoS-profiles. This is used to convert the resource demand of components that have been measured in a different environment.

In addition, in the centralized solution, inter-container communication is required when: (i) the established contracts are communicated finally. For the hybrid solution, inter-container communication is also required when: (i) the two containers agree to concurrently do negotiations locally, and selected QoS-Profiles that concern both are communicated, and (ii) each container confirms the completion of the independent negotiations and exchanges relevant established contracts.

The centralized solution performs better than the hybrid one in terms of minimizing the inter-container communication. On the other hand, the hybrid solution can take advantage of the concurrent executions of both containers and attain a better solution, especially if the required level of computation is high.

As contracts can be violated, monitoring of contracts is constantly performed. For instance, if a contract is established between `VideoPlayer` and `VideoServer` (see Section 4), the contract contains values of the parameters *frameRate* the `VideoPlayer` expects from the received stream and the *frameRate* the `VideoServer` provides. The contract monitoring checks whether the received stream is not in violation of the contract. When there is a contract violation, re-negotiations are usually performed to gracefully adapt the contract. In a different spectrum, once contracts are established, resources are monitored to see if more resources become available. When this happens, contracts can be upgraded for more quality output through re-negotiations.

In the contract re-negotiation phase, the centralized solution requires the exchange of information between the containers, when (i) any contract violation is detected, and (ii) during monitoring of available resources (which is a periodic activity). Nevertheless, for the hybrid solution, no such communication is required as long as the solution can be provided locally. The communication

would be required if the responsible container cannot perform the negotiations on its own and needs assistance from the other container.

The decision for applying either a centralized or hybrid solution can be made just before the start of the negotiation. The particular choice depends on the cost of inter-container communication and the required level of computation. If the former is expensive, a centralized solution would be better. Nevertheless, for a case that requires a lot of computation (e.g due to large number of components and QoS-profiles) and the environment enables relatively faster inter-container communication, the hybrid solution would be the more efficient approach. During run-time re-negotiations, the hybrid solution is preferred as it avoids unnecessary overheads when it is applicable to perform re-configurations locally.

3.5 Negotiation Ordering

Negotiation ordering is about negotiating one property before another, for example, security properties being negotiated before response time. In this paper, we follow the strategy of negotiating all properties in an atomic manner, i.e. without any order. This is natural as dependencies may exist among the properties. For example, for a QoS-Profile that specifies response time and security properties, the response time among others is influenced by the security attributes selected. If the *key length* in the encryption is large and some sophisticated algorithm is used, the *response time* will be higher. This makes it logical to negotiate response time and security properties in an atomic manner.

We are currently working on a strategy that follows order of negotiation between what we call *coarse-grained* and *fine-grained* properties. For example, negotiations on whether or not to do confidential communication must precede negotiation on particular security mechanisms. Or, negotiation on protocol or coding properties (*coarse-grained*) in a video streaming application must go before frame rate and resolution (*fine-grained*) negotiation. The rationale for this ordering is the fact that negotiation on certain properties makes sense if there is an agreement on other properties.

The procedure we can follow in the coarse-grained property negotiation between a client and server component is outlined as below. It assumes that each property is specified with multiple ordered values.

1. Take the first preference of the client component
2. Search for a *conformant* value in the corresponding property of the server component
3. If there is a matching, register the conformant pair in a temporary agreement list.
4. Proceed to the next value in the list of preference of the client component and go to step 2)
5. The procedure ends when all values are exhausted in the client component's preference list

At the end of the coarse-grained negotiation, the state of the temporary agreement list (Step 3 above) may: (i) be empty - the whole negotiation process stops,

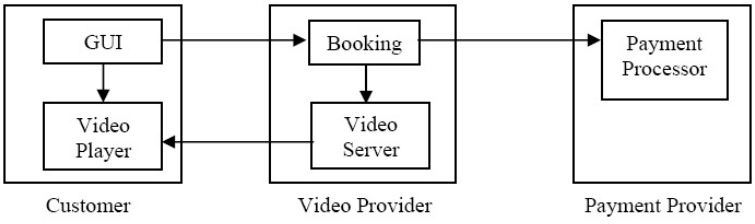


Fig. 3. Interaction of Components in a Video Streaming Application that includes Billing

or (ii) have a single value - the coarse-grained negotiation is concluded at this step, or (iii) have multiple values - final choices have to be made during the fine-grained negotiation phase.

For the fine-grained negotiation, we use the algorithm in Listing 1.1. Doing this two-level negotiation reduces the complexity of the search. This is also demonstrated in the Example Section. Incorporating strategies for negotiation ordering between two *fine-grained* properties (e.g. frame rate and resolution) or between two different QoS categories (e.g. security and timeliness) as may be required by users is the subject of our future research.

4 Example

An application scenario (Fig. 3) we discuss involves the interaction of a customer, video stream provider, and payment provider in the process of video streaming and payment transactions. Two use cases considered are *'order service'* and *'stream video'*. The *'order service'* is concerned with the handling of a user's request for a movie up to the payment process while *'stream video'* involves the streaming of the movie from the video provider to the customer.

The `Booking` component receives, through the `GUI`, information about the title of the movie, customer's data (e.g. name, ID), payment method, etc. `Booking` contacts the movie and customer databases to retrieve pertinent information about the movie (if it exists) like the amount of charge, the mode of payment, etc., and the customer's credit card information. `Booking` then sends the credit card number together with the charge to `PaymentProcessor` for the latter to process the payment. After successfully transacting the payment, the requested movie will be streamed from the `VideoServer` to the `VideoPlayer`. `VideoServer` reads pre-encoded media files from the server and streams video and audio to clients. It is capable of streaming at different frame rates, resolutions, media codings, and protocols. `VideoPlayer` reads the streamed media for playback at the customer's node.

As far as *'order service'* is concerned, customers may have security and performance requirements. They require credit card information be sent to `PaymentProcessor` confidentially. At the same time, when the payment transaction is made, it should not be slow. Some users may prefer stronger security mechanisms and tolerate the response time. `PaymentProcessor` and `Booking` are specified

Table 2. QoS-Profiles of GUI, Booking, and PaymentProcessor Implementations

GUI	Booking		PaymentProcessor
uses IOrder (response time in sec)	provides IOrder (response time in sec)	uses IPayment (key, algorithm, response time in sec)	provides IPayment (key, algorithm, response time in sec)
7	13	PP^-, RSA, 13	PP^-, RSA, 10
9	7	Shared, 3DES, 7	Shared, 3DES, 5
18	5	Shared, DES, 5	Shared, DES, 3

and implemented to give different levels of security and response time to entertain the various requirements. For *'stream video'*, the user's requirements are higher resolution and good frame update rate, where relative preferences on these attributes may vary depending on the video's content.

The collaborating components for *'order service'* include `GUI`, `Booking`, and `PaymentProcessor` while for *'stream video'*, `VideoPlayer` and `VideoServer`. In the former the front-end component is `GUI` while in the latter `VideoPlayer`.

Let us assume that `Booking` implements two interfaces: a uses interface `IPayment` (to get a service from `PaymentProcessor`) and a provides interface `IOrder` (to offer the required service to `GUI`). Likewise, `PaymentProcessor` and `GUI` implement a provides interface `IPayment` and a uses interface `IOrder` respectively. For the sake of readability, the profiles of these components are tabulated as in Table 2. The resource demands of each profile haven't been shown in Table 2. This is provided in Table 3. The concepts of specifications from [8] has been applied to incorporate security attributes in the QoS-profiles of components. In Table 2, PP^- is the payment provider's private key whereas *Shared* is a particular shared secret key between the video and payment providers.

`VideoServer` implements a provides interface `ICompVideo`. `VideoPlayer` implements two interfaces: a uses interface `ICompVideo` and a provides interface `IUnCompVideo`. The COMQUAD component model supports streams as special interface types [7] and allows to specify non-functional properties for them. We conducted an experiment to specify `VideoPlayer` and `VideoServer`. The `VideoPlayer` component was implemented using the JMF [14] framework and the `VideoServer` component abstracts the video media file, where the streams are pull data sources. The original media file has been encoded into many files with differing frame rates, resolutions, and encoding algorithm.

The PC where the `VideoPlayer` ran and the measurement was conducted has a configuration of: AMD Athlon(tm) XP 1600+, 1GB RAM, Windows 2000 Professional, Java 2 version 5. The PC was connected through a 100 Mbps LAN to the PC that hosted `VideoServer`. The purpose of the whole measurement was not ultimate accuracy but to have approximate data, which can enable us to validate our contract negotiation protocol and algorithm. Windows Performance Monitor has been used to measure the resource usage of the components. During the measurements, average bandwidth and CPU percentage time have been considered. The measured QoS-profiles of `VideoPlayer` and `VideoServer`, with properties UDP protocol and mp42 coding, is given in Table 3.

Table 3. QoS-Profiles of VideoPlayer and VideoServer Implementations

VideoPlayer		
provides IUnCompVideo (resolution, frame rate in s^{-1})	uses ICompVideo (resolution, frame rate in s^{-1})	Resource (CPU in %, bandwidth in Kbps, memory KB)
352x288, 30	352x288, 30	13.23, 2165, 31.6
352x288, 15	352x288, 15	8.90, 2146, 30
352x288, 10	352x288, 10	8.91, 2076, 30.2
352x288, 5	352x288, 5	5.90, 1852, 29.5
352x288, 1	352x288, 1	2.31, 1644, 29.2
176x144, 30	176x144, 30	0.97, 321, 25.6
176x144, 15	176x144, 15	0.90, 252, 18.8
176x144, 10	176x144, 10	0.62, 208, 24.4
176x144, 5	176x144, 5	0.39, 135, 24.0
176x144, 1	176x144, 1	0.11, 34, 24.4
128x96, 30	128x96, 30	0.51, 152, 26.1
128x96, 15	128x96, 15	0.38, 120, 25.1
128x96, 10	128x96, 10	0.64, 108, 24.5
128x96, 5	128x96, 5	0.19, 70, 24.2
128x96, 1	128x96, 1	0.04, 24, 22.9

VideoServer	
provides ICompVideo (resolution, frame rate in s^{-1})	Resource (bandwidth in Kbps)
352x288, 30	2165
352x288, 15	2146
352x288, 10	2076
352x288, 5	1852
352x288, 1	1644
176x144, 30	321
176x144, 15	252
176x144, 10	208
176x144, 5	135
176x144, 1	34
128x96, 30	152
128x96, 15	120
128x96, 10	108
128x96, 5	70
128x96, 1	24

As the measurements have been taken under light load conditions, it is assumed that the bandwidth requirement of `VideoServer` is taken to be the same as that for `VideoPlayer`. Moreover, the measured CPU requirements of `VideoServer` are too small (in the range of 0.1%). Hence, the CPU time has been left out from Table 3 as it is too small to have influence in our validation.

Suppose the user's QoS requirements and preferences for *'stream video'* are:

- $frameRate > 12fps$, $resolution = 176X144$
- first preference is frame rate, then resolution

Let the available resources are: Customer: CPU 80%; Video Provider: CPU 50%; and end-to-end bandwidth: 1Mbps. The available CPU times given are already converted to be consistent with the environment where the QoS-profiles have been measured.

Before running the algorithm of Listing 1.1, the QoS-profiles of `VideoPlayer` and `VideoServer` must be sorted according to the user's preference, i.e., first on frame rate and second in resolution. The QoS-profiles in Table 3 are sorted first on resolution and then in frame rate. The main steps in the negotiation are executed as follows.

`BuildSolution()` (Listing 1.1) is invoked to iteratively select heuristically optimized QoS-profiles. For a front-end component, `FindConformantProfiles()` uses both conformance and user's constraints to find conformant profiles.

- 1^{st} Iteration
  - `FindConformantProfiles(On_Client)` gets a QoS-profile for `VideoPlayer` with the offered QoS contract: $15fps$, $176X144$

- `CheckResourceConstraint(On_Client)` is successful.
- `FindConformantProfiles(Across_Containers\On_Client)` will select a QoS-profile for `VideoServer` with the offered QoS contract: $15fps$, $176X144$
- `CheckResourceConstraint(Across_Containers)` is successful.
- `FindConformantProfiles(On_Server\Across_Containers)` will not add any additional selected profiles as there are no components on the server other than `VideoServer`.
- `CheckResourceConstraint(On_Server)` is successful in this iteration.

– 2^{nd} Iteration
- `FindConformantProfiles(On_Client)` gets a new valid QoS-profile for `VideoPlayer` with the offered QoS contract: $15fps$, $352X288$
- `CheckResourceConstraint(On_Client)` is successful.
- `FindConformantProfiles(Across_Containers\On_Client)` will select a QoS-profile for `VideoServer` with the offered QoS contract: $15fps$, $352X288$
- `CheckResourceConstraint(Across_Containers)` is NOT successful. The temporarily selected QoS-profiles in this iteration (first and third steps) are invalidated. The appropriate configurations, hence, are those selected in the previous iteration, i.e. 1^{st} iteration.

The bottleneck resource has been determined to be the network bandwidth in the 2^{nd} iteration. Negotiations, however, would continue in the server and client containers with the `DoFurtherNegotiations(.)`. As `VideoPlayer` and `VideoServer` are the only components on the client and server respectively, further negotiations in the two containers wouldn't produce new profiles.

The selected profiles for `VideoPlayer` and `VideoServer` are the ones with the offered QoS contract $176X144, 30fps$. Although the profiles with $176X144, 15fps$ satisfied all constraints, an improved solution was sought as the negotiation's goal was the maximization of the user's satisfaction.

Let's consider now the case of an unsuccessful negotiation, which arises when trying to fulfill user's QoS requirements while resource and/or conformance constraints can't be met. Assume the user's requirements and preferences are as previous and the available resources are: Customer: CPU 80%; Video Provider: CPU 50%; end-to-end bandwidth: 200Kbps.

The 1^{st} iteration of Listing 1.1 can't be completely executed as `CheckResourceConstraint(Across_Containers)` returns an unsuccessful result. To handle this over-constrained condition, the user's requirement will be relaxed in succession. This step depends on the QoS-profiles of `VideoPlayer` (note that the available QoS-profiles are sorted according to the user's preferences). In the first iteration of the relaxation step, the chosen degraded requirement is: $frameRate = 15fps$ and $resolution = 128X96$. At this step, the negotiation turns out to be successful. If the user's preference were first resolution and then frame rate, the relaxation step would have tried the sequence $176X144, 10fps$; $176X144, 5fps$; etc. and the negotiation would have been successful at $176X144, 5fps$. If the

user's requirement had been given in a range (like frame rate 5 - 15 fps or resolution 128X96 - 800X600), then this information would have been used in the relaxation step not to further relax the user's requirement below the given minimum.

Table 3 shows QoS-profiles of `VideoPlayer` and `VideoServer` for a UDP protocol and mp42 encoding. Suppose `VideoPlayer` is specified with RTP/UDP and UDP protocols and mp42, mpg4, and h263 coding; and `VideoServer` specified with RTP/UDP, UDP, and TCP protocols and mp43, mp42, mpg4, and h264 codings. For all combinations of coding and protocol, QoS-profiles similar to Table 3 have to be specified. The coarse-grained property negotiation between `VideoPlayer` and `VideoServer` (see Section 3.5) culminates in the selection of: (i) mp42 and mpg4 coding, and (ii) RTP/UDP and UDP protocol. In both properties, multiple values have been selected and the final decision is made by the fine-grained negotiation, which is demonstrated above by following Listing 1.1 step-by-step.

For *'order service'*, assume the user has a higher preference to security than response time. Before making the contract negotiation for the three components (`GUI`, `Booking`, and `PaymentProcessor`), the QoS-profiles must be ordered first on security properties and then on response time, as it is done in Table 2. The algorithm in Listing 1.1 is applied to select profiles that provide the near-optimal solution. Conformance on security properties between two interacting components exists when the constraints on all the parameters (key type, key length, and algorithm) match exactly. For matching offer and expectation on response time, the conformance relationship must take into account the transmission delay between the customer's workstation and video provider's server (for `GUI` and `Booking`) and between the video provider's server and the payment provider's server (for `Booking` and `Payment`).

Our algorithm selects appropriate profiles by improving from lower to higher quality to maximally satisfy the user's requirements and preferences. For QoS-profiles that contain both security and response time properties (Table 2), if the user has a higher preference to security, the step-by-step increment leads to better security (e.g. from $Shared, DES$ to PP^{-}, RSA), but this affects the response time negatively. In the case where conflicting properties exist the algorithm maximizes the satisfaction of the most preferred property.

5 Related Work

The work in [6] treats contract negotiation only in a single component container and has not pursued the case of distributed contract negotiation. Their approach ties the container-based negotiation to a real-time operating system called DROPS. But, in our case, the negotiation is not dependent on a particular platform. Furthermore, our algorithm treats the over-constrained case.

In [10] QoS contract negotiation is applied when two components are explicitly connected via their ports. In the negotiation, the client component contacts the server component by providing its requirement; the server responds with a

list of concrete contract offers; and the client finally decides and chooses one of the offers. The contract negotiation phase is started if the components want to negotiate a certain QoS contract. The container of the client component decides on this. The container knows whether the hosted component wants to negotiate a certain QoS contract or not. Our approach differs in that negotiation precedes every service invocation of the application. The service invocation may involve more than two components and hence the negotiation takes place among all the collaborating components. Furthermore, we have described the various contract negotiation mechanisms like negotiation goal, efficiency of the negotiation process, and over-constrained situations in a generic manner.

QuA [13] aims at defining an abstract component architecture, including the semantics for general QoS specifications. The proof of concept is provided by implementing an open framework for platform managed QoS. In QuA's *QoS-driven Service Planning* has similarities to our concept of contract negotiation. Complexity issues, however, haven't been accounted for in the service planning.

6 Conclusions and Outlook

A QoS contract negotiation is applied to select appropriate QoS contracts between collaborating components. Important challenges to be addressed are possible resource constraints at each node and the network, the efficiency of the negotiation process, and the selection of a heuristically optimized solution. Component QoS contracts are specified in QoS-profiles that contain the required and provided non-functional properties as well as resource demand. Due to its dynamic nature, the QoS contract is specified with multiple QoS-profiles.

We tackled the problem first by modeling the negotiation process as a constraint satisfaction problem. Based on this, we proposed an algorithm that selects concrete QoS contracts for the distributed components that meet the stated challenges and mechanisms for relaxing constraints when the contract negotiation couldn't produce a solution. We also found out that a hybrid solution, that combines centralized and decentralized approaches, performs well for realizing the proposed algorithm.

We are currently working on incorporating the case of multiple clients in our negotiation model. In this paper, the negotiation goal we assumed was only the maximum satisfaction of one client (the user). But, in the multiple client scenario, we want to include, among others, the service provider's needs as a parameter in the negotiation goal. We are planning to introduce *negotiation policy constraints*, to our existing model, which enables us to capture the different negotiation goals.

References

1. J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems.* PhD thesis, University of Oslo, 2001.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

3. B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Springer, 2005.
4. E. C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*, pages 278–283, 1989.
5. S. Frolund and J. Koistinen. Quality-of-service specification in distributed object systems. *IOP/BCS Distributed Systems Engineering Journal*, December 1998.
6. S. Göbel, C. Pohl, R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. The COMQUAD component container architecture. In J. Magee, C. Szyperski, and J. Bosch, editors, *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, pages 315–318, Oslo, Norway, June 2004. IEEE.
7. S. Göbel, C. Pohl, S. Röttger, and S. Zschaler. The COMQUAD Component Model—Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects. In *3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, 22–26 Mar. 2004.
8. K. M. Khan and J. Han. Composing security-aware software. *IEEE Software*, pages 34–41, January/February 2002.
9. Object Management Group. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms: Revised submission. OMG Document, May 2003. URL http://www.omg.org/docs/realtime/03-05-02.pdf.
10. T. Ritter, M. Born, T. Unterschutz, and T. Weis. A QoS metamodel and its realization in a CORBA component infrastructure. In *proceedings of the Hawaii International Conference on System Sciences*, 2003.
11. S. Röttger and S. Zschaler. CQML$^+$: Enhancements to CQML. In J.-M. Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, June 2003.
12. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach; 2nd edition.* Prentice Hall, December 2002.
13. R. Staehli, F. Eliassen, and S. Amundsen. Designing adaptive middleware for reuse. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 189–194, New York, NY, USA, 2004. ACM Press.
14. Sun Microsystems. Java Media Framework API Guide, Nov. 2001.
15. COMQUAD Project – Components with Quantitative Properties and Adaptivity. http://comquad.org/.
16. A. Ulbrich, T. Weis, K. Geihs, and C. Becker. DotQoS – a QoS extension for .NET remoting. In *International Workshop on Quality of Service (IWQoS)*, volume 2707 of *Lecture Notes in Computer Science (LNCS)*, pages 363–380, Monterey, CA, 2003.
17. G. J. Vecellio, W. M. Thomas, and R. M. Sanders. Containers for predictable behavior of component-based software. In *The 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, 2002.
18. N. Wang, C. D. Gill, D. C. Schmidt, A. Gokhale, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, March 2003.
19. T. Weis, N. Plouzeau, K. Geihs, A.-M. Sassen, and J.-M. Jézéquel. QCCS: Quality controlled component-based software development. *Kluwer International Series in Engineering and Computer Science*, 2003.
20. M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

RIMoCoW, a Reconciliation Infrastructure for CORBA Component-Based Applications in Mobile Environments

Lydialle Chateigner, Sophie Chabridon, and Guy Bernard

GET/INT, CNRS UMR SAMOVAR, 9 rue Charles Fourier, 91011 Évry, France
{Lydialle.Chateigner, Sophie.Chabridon, Guy.Bernard}@int-evry.fr

Abstract. Initially targeted at supporting individual work, personal devices are more and more used for collaborative activities in mobile environments. In this context, service continuity raises the problem of data availability during disconnections. This relies on optimistic replication with the deployment of a local copy of the necessary application components, leading to potential divergence of the distributed copies. Consequently, the main challenge consists in designing reconciliation mechanisms in order to synchronize copies at reconnection time. In this paper, we present RIMoCoW (Reconciliation Infrastructure for Mobile Collaborative Work) a reconciliation infrastructure designed and implemented for CORBA component-based applications in a mobile collaborative work environment. RIMoCoW follows a service-based approach and integrates SOCT4Mob (Serialization of Concurrent Operations by Transposition in Mobility) a synchronization algorithm we have extended for the mobile context. In addition, we propose recovery and sequencer related policies that fit to mobile requirements. Finally, RIMoCoW is implemented using an emergency management application. The performance results show that our infrastructure can reconcile in roughly 30 minutes more than 16.000 victims medical files filled by 8.000 rescuers, which corresponds to a level of 6 on a 6-degree European scale for technological accidents.

Keywords: RIMoCoW, Mobility, cooperative work, reconciliation of copies, CORBA component-based applications, SOCT4Mob.

1 Introduction

Collaborative work enables a group of scattered actors to achieve a common aim through data sharing. While recent developments have made collaborative work possible remotely, there are currently few solutions for mobile users. With the increase of their capacity and the introduction of new services, personal mobile devices originally aimed at supporting personal work, can now support collaborative work. In the context of mobile collaborative work, data consistency is the first initial requirement that oriented our work. Shared data are replicated to provide a working continuity while being disconnected. Because of concurrent

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1668–1685, 2006.
© Springer-Verlag Berlin Heidelberg 2006

modifications, copies may diverge and lead to conflicting updates. That is why upon reconnection, divergences must be dealt with to achieve consistency.

Within the above context, our contribution lies in the integration of our reconciliation platform, namely RIMoCoW (Reconciliation Infrastructure for Mobile Collaborative Work), into a component-based extensible middleware in order to maintain the consistency of shared and distributed applications. This infrastructure follows a service-oriented approach which reflects the splitting paradigm between purely applicative aspects and extra-functional ones. The reconciliation service of RIMoCoW, a new extra-functional service, integrates SOCT4Mob, a reconciliation algorithm which is adapted to a distributed mobile environment. This algorithm states that two independent operations are concurrent if they are performed on two copies deployed on two distinct sites. A conflict results from the execution of two concurrent operations with incompatible aims. The choice of SOCT4Mob was motivated by the fact that this approach resolves conflicts automatically without aborting or undoing any operation because each applicative case is predefined and handled thanks to forward transposition functions [18]. This paradigm allows to transform, for instance to modify parameters value, an incoming operation with respect to the local operation executed previously in order to converge eventually to the same value at all sites. Moreover, SOCT4Mob recommends operation transfer instead of state transfer [17], offering more flexibility for conflict resolution knowing the whole suite of operations that led to it.

The remainder of this paper is organized as follows. Section 2 presents the architecture and the functionnalities of each component of RIMoCoW. Then, section 3 offers the specification of the distributed Sequencer Service we propose and describes the two policies it implements. Section 4 focuses on SOCT4Mob and its related *Recovery* policy. Experimental results of synchronization and integration times computations are provided in section 5. Section 6 presents the related work on existing reconciliation systems. Finally, section 7 concludes this paper and proposes several research directions.

2 RIMoCoW : A Reconciliation Infrastructure

The RIMoCoW (Reconciliation Infrastructure for Mobile Collaborative Work) platform that we propose, holds up users disconnected work while taking care of CORBA-components copies reconciliation. It is integrated into the open source component-oriented middleware OpenCCM [16] to allow a transparent adaptability of applications in spite of connectivity changes. This integration is made possible using an Extensible Container Model (ECM) [5] which offers a complete and clearly specified framework to integrate new extra-functional services in the CCM container in a portable and suitable manner. Furthermore, RIMoCoW relies on a service-oriented approach which offers a well-defined separation between cooperating functionalities within the platform. It is modelled and implemented using the Fractal [8] component model. The advantages of Fractal are that it is hierarchical and allows not only service building but can also be applied to middleware.

RIMoCoW architecture is presented on Fig. 1. RIMoCoW can be seen as a logical composition of five middleware services, with the Reconciliation service as the central part interacting with four external services, Cache, Connectivity Detection, Sequencer, and Group Communication, that we present below. We assume that RIMoCoW and the necessary components - necessary meaning required on mobile terminal for allowing disconnected work [13] - of user applications are replicated and deployed on each node of the system. Therefore, the client part of user applications corresponds to the Client Component, receiving directly user operations, while the Copy Component is a proxy of a shared component application deployed on the mobile terminal, and processing application operations.

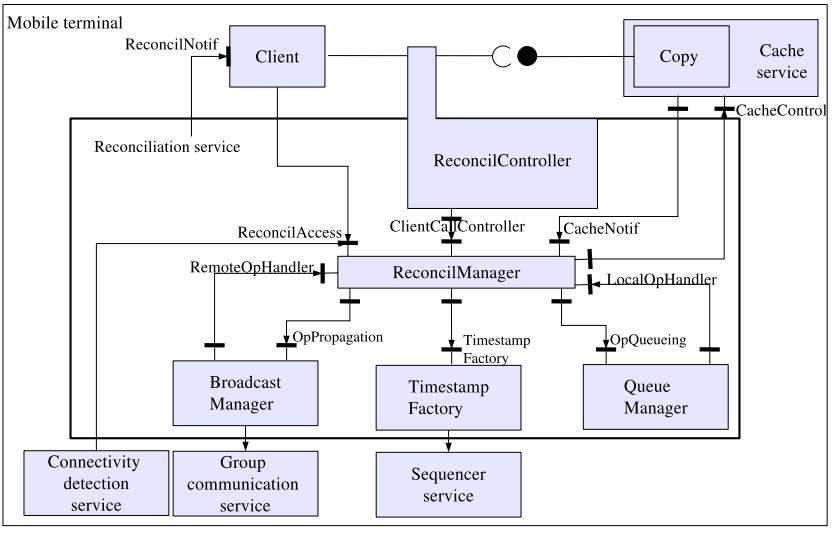


Fig. 1. RIMoCoW Architecture

Connectivity Detection Service. This service monitors the locally available physical resources such as battery power or connection quality [2]. It notifies the reconciliation service when the connectivity mode changes to allow it for dynamic adaptation.

Cache Service. This service asserts the replication of the components necessary for disconnected work in the software cache on the terminal before network cut [13]. It defines component deployment and replacement policies.

Sequencer Service. This service returns on request an unique ticket to each operation so as to globally order all operations of the system. It is replicated on each site which may disconnect, therefore a synchronization mechanism among replicas to maintain a continuous global order is also provided. This service is detailed in section 3.

Group Communication Service. This service carries the messages exchanged between participants of the group. It relies on a membership mechanism, a reliable and ordered broadcast mechanism and a consistent view manager based on the members notifications when joining or leaving the group. Our implementation has been done using the JavaGroups toolkit [10].

The Reconciliation service being a composite, we now present each of its constituents.

ReconcilController Component. The *ReconcilController* component operates between the *Client* component and the *Reconciliation Service* and acts at the client side by intercepting all outgoing requests and forwarding them to the *ReconcilManager* component.

ReconcilManager Component. The *ReconcilManager* component is the entry point of the service and thus co-ordinates the reconciliation process relying on all the other sub-components. Triggered by the *Connectivity Detection* Service through the *ReconcilAccess* interface, it activates the reconciliation when the connection is possible or prepare for logging local work otherwise.

QueueManager Component. When the *ReconcilManager* component receives an application operation, the corresponding XML command [4] is forwarded to the *QueueManager* component in order to be logged. Three queues are associated with each component, namely *local*, *receipt* and *history*. The first one contains operations generated by the client and not yet timestamped. The second one stores operations remotely timestamped and broadcasted on the network. The last one represents the totally-ordered sequence of local and remote operations.

BroadcastManager Component. The main function of the *BroadcastManager* component is to ensure the communication between sites by message passing using reliable asynchronous channels. This component can be invoked by the *ReconcilManager* entity via the *OpPropagation* interface to send, broadcast or receive information such as an XML command as well as a message.

We define hereafter the different kinds of messages which can be encapsulated in the *send*, *broadcast* and *receive* primitives. We first detail the three messages related to the sequencer service (cf. section 3). `la` represents the logical address of the site sending the message and `id` refers to the identifier of a request.

- **RequestTicket** `(la,cn,tv)` to ask for a ticket whose value is `tv` to timestamp an operation associated to a component with the name `cn`.
- **RequestUpdate** `(la,id)` to allow the reconnecting site to request from the connected replicas the update of its sequencer.
- **AnswerUpdate** `(la,csv,id)` to allow the connected site to update a sequencer which reconnects, on the reception of the **RequestUpdate** message. `csv` is a vector which reflects the state of each sequencer associated with each *Copy* component.

The following messages are used in the recovery policy of SOCT4Mob (cf. section 4). `la` represents the local address of the sender site of the message and `hs` refers to the state of history of each *Copy* component.

- **RequestHistory** `(la,hs)` to trigger the reconciliation process.
- **AnswerHistory** `(la,hs)` to forward information about the state of sites which have received a `RequestHistory` message.
- **RequestDiff** `(la,ldStart,ldEnd)` to recover operations not received during the disconnected work between *ldStart* and *ldEnd* bounds.

TimestampFactory Component. SOCT4Mob is a ticket-based algorithm. It requires a local operation to be timestamped before it is broadcasted. Moreover, a remote one is integrated into the history according to its ticket. SOCT4 [18], the predecessor of SOCT4Mob, relies on an unique sequencer to achieve a global unique order. Nevertheless, in a mobile context, the site which implements the sequencer may potentially disconnect and become out of reach of the other working sites. The consequence is to freeze the collaboration during this lapse of time. To overcome this limitation, the *TimestampFactory* component relies on an underlying sequencer service to timestamp local operations as described in the next section.

3 Specification of the Sequencer Service

This section presents the specification of the distributed sequencer service for which a replica is deployed on each site of the system. Our work is original because the sequencer service is built to be suitable for collaborative work in a mobile environment. That allows sites to get sequence numbers that can be used to label successive operations. The result is to achieve a total order on operations exchanged on the network despite the disconnection of several replicas and without needing additional timing assumptions over an asynchronous distributed system. The sequencer service has to satisfy the following main properties:

- (i) Unicity: a sequencer replica must deliver a sequence number that has not already been assigned by the sequencer,
- (ii) Consecutiveness: the sequence of numbers, starting from zero, assigned by a sequencer has no "hole",
- (iii) Liveness: the sequencer service is always live, meaning that at any time at least one replica of the sequencer service has to remain connected and available.

We adopt a collaborative distributed point of view, like in the peer-to-peer model, where a site, member of a work group, can be a client just as well as a server. Therefore, the sequencer service is replicated on each site of the system and a sequencer handles client requests coming from its own site. Communication between members is asynchronous, reliable and made possible thanks to the primitives described in section 2. The sequencer service may be invoked on

two types of events: a request for a ticket or a request for a sequencer replica update when a site reconnects. The following paragraph provides respectively the *Timestamping* policy and the *Sequencer Synchronization* policy pseudo-code. *la* refers to the local address and *ra* denotes the remote address.

Timestamping Policy Description. To timestamp an operation, a site invokes the *getTimestamp()* procedure on its sequencer replica, which then broadcasts the *RequestTicket* message to the sequencer service (see Fig. 2). This procedure is then suspended thanks to the *monitor.wait* instruction. The *receiveRequestTicket* procedure describes the way the sequencer replicas answer. It is based on the total global order property provided by the group communication service. Finally, the request is reactivated by the *monitor.notify* instruction.

```
getTimestamp() {
1  broadcast(RequestTicket(la, getTicket() + 1 )) ;
2  requestedTicket = currentTicket() + 1 ;
3  monitor.wait() ;
4  requestedTicket = null ;
5  return getTicket() ;
6 }
```

```
receiveRequestTicket( ra, ticket) {
1  if (ticket > currentTicket() ) {
2        if (ra = = la) {
3           setTicket(ticket)) ;
4           monitor.notify() ;
5        } else {
6           setTicket(ticket)) ;
7           if (requestedTicket = = ticket) {
8     broadcast(RequestTicket(la, getTicket() + 1) ;
9              requestedTicket =  getTicket() + 1
10          }}}}
```

Fig. 2. Timestamping Policy

Sequencer Synchronization Policy Description When a disconnected site reconnects, its sequencer state may be obsolete. So, the *Sequencer Synchronization* policy is started thanks to the *update* method broadcasting the *RequestUpdate* message (cf. Fig. 3). Then, the sequencer is frozen by the *monitor.wait* instruction. The *receiveRequestUpdate* procedure allows up-to-date sites to send the *AnswerUpdate* message containing the last delivered ticket value. Thus, the

```
update () {
1  broadcast(RequestUpdate(la, currentRequestId));
2 sendRequestId = id;
3 monitor.wait();
4 update = true;
5 currentRequestId++;
6}
```

```
receiveAnswerUpdate(ra, ticket, id) {
1   if (id = = currentRequestId) {
2           setTicket(max(ticket, currentTicket));
3           monitor.notify();
4   }
5}
```

```
receiveRequestUpdate(ra, id) {
1  if (update)
2        send(ra, AnswerUpdate(la, currentTicket, id));
3 }
```

```
onDisconnection() {
1   update = false;
}
```

Fig. 3. Sequencer Synchronization Policy

reconnecting sequencer is updated with the new value *via* the *setTicket* method. Finally, the request is reactivated by the *monitor.notify* instruction.

In this section, we have presented the specification of the sequencer service. Our design adopts a service replication scheme which is well suited to mobile environments. This optimistic approach enables cooperation continuity and provides some level of fault tolerance and robustness despite replica failures or disconnections, as long as there is at least one active replica.

4 SOCT4Mob, a Reconciliation Algorithm for Mobility

This section deals with the SOCT4Mob algorithm which represents the *Reconciliation* policy used within RIMoCoW. Its predecessor SOCT4 [18] relies on application-dependent operation transformations which define for each couple formed of a remote and a local operations the resultant operation. To understand how the Operational Transformation approach maintains replica consistency, let us consider that a user A receives an operation op that was previously executed by some user B on a replica of the shared entity. If user A integrates directly op on its replica by executing it as it is, this might lead to an inconsistency, depending on the history of the previously executed operations. Therefore, user A may execute a variant of op, denoted by op' (called $transformation$ or $transposition$ of op) that intuitively intends to achieve the same effect as op and that preserves consistency. op' must be written by the application developer at implementation time. Transformational operations are difficult to write as very strict conditions have to be respected so that convergence is guaranteed [9]. With SOCT4Mob, we have made operational transformations as independent as possible from the application by modelling data as an XML tree. Such a tree can be manipulated by only a few canonical operations such as creating a node, adding an attribute to a node, changing the value of an attribute [4]. This relieves the application developer from writing specific operational transformations with the counterpart of having to provide a mapping from application operations to operations on a XML tree.

Besides, SOCT4 has been used for collaborative work in synchronous and asynchronous mode [3] where uncoupled work is performed in the presence of a permanent connection. It means that even though the site is not synchronized and updated, it remains connected and can receive messages exchanged on the network. In our context, isolated work occurs when the network is not physically available. As a result, the mobile terminal can neither receive remote modifications nor broadcast its locally performed operations. Furthermore, existing group communication toolkits do not offer the storage of operations not received because of a disconnection. That is why SOCT4Mob calls for a the *Recovery* policy to cope with mobile environments.

The pseudo-code of the *Recovery* policy is given in Fig. 4 and 5. It is event-based and is simplified to take only one component into account. Messages exchanged during the recovery by the reconnecting and connected sites are described in section 2. Let Ld be the last delivered ticket of the involved component, BS the

```
synchronized startRecovery () {
1 broadcast(RequestHistory(la,ld));
2}

receiveAnswerHistory(ra, ld) {
1 if (localLd < ld) {
2       table.addElement(ra, ld);
3       startDownload(answerTable);
4   }
5}
```

```
startDownload(table) {
1 while (download possible) {
2       (bs, ldStart, ldEnd=
        computeBestDownload(answerTable);
3       send(bs, RequestDiff(la, ldStart, ldEnd));
4       table.downloading(bs,ldStart, ldEnd);
5   }
6 }
```

Fig. 4. Recovery Policy at reconnecting client side

```
receiveRequestHistory (ra, ld) {
1 if (localLd < ld)
2   broadcast(RequestHistory(la, localLd));
3 else
4   if (localLd > ld)
5       send(ra, AnswerHistory(la, localLd));
6}
```

```
receiveRequestDiff(ra ldStart, ldEnd) {
1 for (i = ldStart; i <= ldEnd; i++) {
2       send(ra, HistoryQueue(i));
3   }
4}
```

Fig. 5. Recovery Policy at answering sites side

logical address of the best site to contact, *LdStart* and *LdEnd* the bounds for the download and *la* and *ra* respectively the local and remote address.

The *Recovery* policy acts upon reconnection. It allows a reconnecting site to get remote operations that were performed on the other sites while it was disconnected, so that this reconnecting site can be updated. Our proposition exploits the explicit consultation of groups members and is triggered by the *startRecovery()* method of the *ReconcilManager* component. A timeout mechanism allows possible site failures or network delays to be detected.

The advantages of our solution are many. First of all, the reconciliation of the reconnecting and existing sites is achieved. Moreover, the *Recovery* policy does not require a dedicated site as in [15], but is performed by members of the collaborative group allowing to balance the load inside the group. A simple clock mechanism provides failure detection and a local answer table ensures fault tolerance. All these functionalities allow us to say that our proposition is well-suited to nomadic context requirements.

5 Implementation and Evaluation

In this section, RIMoCoW is evaluated, first within a general context, and then in the specific context of the AMPROS project.

5.1 Evaluation in the General Context

We have implemented *Reconciliation* and *Sequencer* services as described in sections 2, 3 and 4. In order to evaluate the efficiency of the service, we have

performed some experiments to measure synchronization and integration times. These measurements were carried out by varying the number of participants in the working group and the number of XML atomic operations generated by members.

These performance tests were run on a set of 7 computers, 1 laptop (Mobile Intel 1.80 GHz Pentium 4, 512M RAM) and 6 PC (Intel 1.80 GHz Pentium 4, 512M RAM), wired with a 10 Mps LAN network. All computers have almost similar capacities i.e. GNU/Linux RedHat 7.3 powered by a 1.80 GHz Mobile Pentium with 512 Mb RAM. The laptop has been selected to play the operation generator role. The procedure was the following. At first, all the sites started in disconnected mode. Then, one of them generated a maximum of 100.000 operations. Afterwards, the other 6 sites were connected successively and finally, the generator site was connected which triggers the synchronization process.

Synchronization Time Results. Firstly, we focus on the amount of time required to synchronize the sites involved. Synchronization time represents the time necessary for a site to get updated by only taking into account remote operations. In other words, the local queue is empty, thus the forward transformation functions are not invoked. Our goal with these tests is to evaluate the performance of our sequencer service and the SOCT4Mob algorithm, except its *integration* procedure.

Figure 6 shows the variation of the synchronization time according to the number of operations generated and the size of the working group. The synchronization of 1.000 operations within a group of 2 members lasts only 12s, while with 10.000 operations in a group of 7 members the synchronization time takes 200s (about 3 mn). The results show that the synchronization time increases in a linear way according to the number of operations and the number of members involved for values lower than 10.000 operations. However this time remains absolutely acceptable for a new user who joins the cooperative group or for a user reconnecting after leaving a disconnected area without having performed operations during the disconnected period.

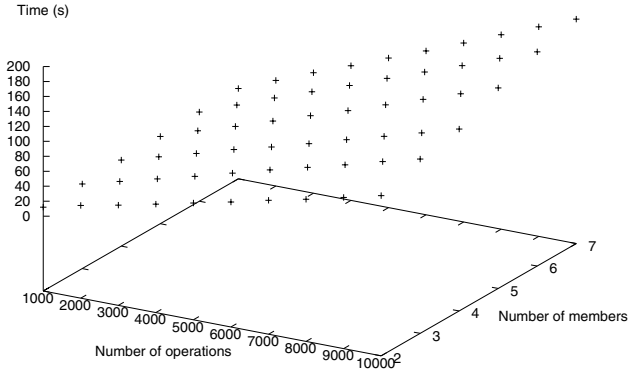


Fig. 6. Synchronization Time (in s)

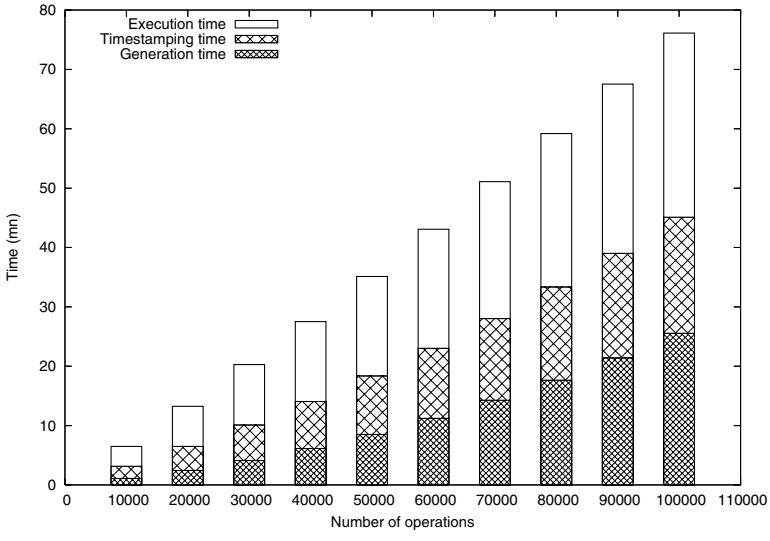


Fig. 7. Breakdown of Synchronization Time on generator site

Decomposition of Synchronization Time. In order to determine precisely which factors have an influence over the synchronization time, we have split up the synchronization time into different parts. Figure 7 illustrates the breakdown of time at the site generating operations. The *Generation* time corresponds to the time needed for the execution of the operations. The *Timestamping* time indicated the time useful for the progress of the *Timestamping* policy described in section 3. The *Execution* time reflects the execution of the SOCT4Mob algorithm without transpositions.

On the receiver side, the synchronization time can be split into 3 distinct parts as shown in Fig. 8. The first one, the members contact time, represents the time necessary just after the reconnection to launch the *Recovery* policy and the *Sequencer Synchronization* policy of the sequencer service. The sequencer update time is the time needed for the *Sequencer Synchronization* policy, according to the generator site sequencer. Finally, the update time is related to the recovery of remote operations and their repercussion on the local *Copy* component.

Synchronization Time Decomposition Analysis. Apart from the members contact time which is constant, the other times are strongly correlated with the number of operations, particularly the *Generation* time. Moreover, the *Timestamping* and the *Sequencer Update* times, related to the *Timestamping* and the *Sequencer Synchronization* policies, consume mainly network bandwidth. Thus, we can conclude the overhead generated for a greater number of operations is due to the communication time through the network. Indeed, JavaGroups emulates the broadcast by the point by point multicast. Hence, the growth of the group size leads to a multiplication of messages exchanged on the network. Other constituent of times, i.e. the *Execution* and the *Update* times,

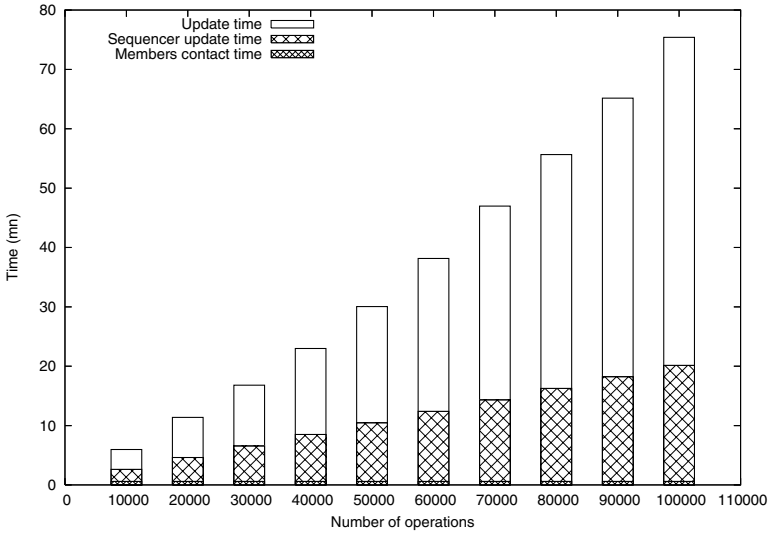


Fig. 8. Breakdown of Synchronization Time on a remote site

correspond to Corba invocation of received operations through the *Corba Dynamic Invocation Interface* (DII) which are avid for CPU time.

These series of tests demonstrate a simple case in which a user equipped with a mobile device joins the group for the first time. At reconnection, only an update is needed to be consistent with the rest of the group because no operation has been performed on the terminal during the disconnected work.

Integration Time Results. In these second series of tests, we allow all group members to generate operations during disconnection time. We aim to evaluate the efficiency of the SOCT4Mob *integration* procedure by measuring the *Integration* time of a remote operation locally. It represents the main part of the reconciliation time; in other words, it is the time required to perform the SOCT4Mob forward transpositions (Cf. section 4). It consists in transforming each remote received operations by taking the local ones into account before their local execution. Thus, we conducted a series of experiments using a simple application on a laptop GNU/Linux RedHat 7.3 (Mobile Intel 1.80 GHz Pentium, 512 Mb RAM). In addition, we implemented a simulator that artificially fills up the receipt queue of the site with operations according to the number of members in the collaborative group. In this case, each site performs the same number of operations from 100 to 600 during the disconnected period.

Integration Time Analysis. For a group of 7 members, this time varies between 1 mn 57 s for the integration of 700 operations and 53 mns 23 s for 4.200 operations, as shown in Fig. 9. The main difference with the first tests concerns the number of operations that it is possible to integrate within 1

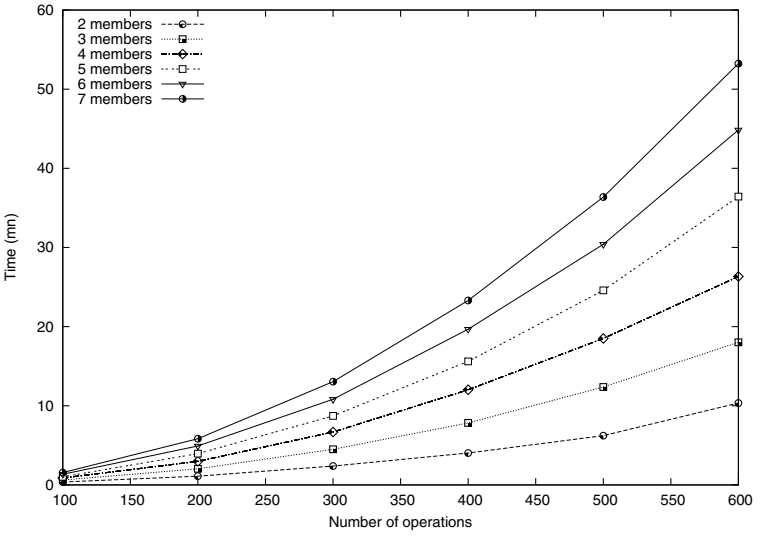


Fig. 9. Integration Time

hour. The limit here is only 4.200 operations instead of 100.000 operations for the synchronization time. This parameter is reduced because of the combinatorial explosion generated by the SOCT4Mob *integration* procedure. In the worst case, 4.320.000 transformations are necessary to integrate 3.600 operations within a group of 7 members. This figure comes from the following equation : $Nb_{forwardTranspositionFonctions} = Nb_{localOp} \times \sum_{S=1}^{S=Nb_{remoteSites}} Op_S \times 2$ where we consider a pessimistic case with all sites generate an equivalent number of 600 operations. The local site has 600 operations to be reconciled with the $6 \times 600 = 3.600$ operations received from the other 6 sites. The coefficient 2 comes from the fact that SOCT4Mob implies a two way search of the queues. These results show that more the number of operations performed during the disconnected time is important more the execution of SOCT4Mob algorithm requires a consequent time. So, the integration time is correlated to the number of operations as well as to the size of the group. Hence, this algorithm is rather adapted to applications which accept a small number of operations. This requirement is met by the target AMPROS application because the update of a medical file involves only 5 XML primitives. Thus, a very few of operations are performed on each site which implies a limited number of forward transpositions. Consequently, it contributes to reduce the reconciliation time.

5.2 Evaluation in the Specific Context of the AMPROS Project

Our work is performed in the context of the AMPROS project [1] whose main purpose is to provide a proactive middleware platform for applications such as emergency aid and crisis management. We illustrate our approach with the example of a medical emergency management application which represents a

realistic use case of our work. In this section, we describe a simple scenario where the reconciliation service is essential to preserve data consistency.

In a disaster area, paramedics and first-aid workers are deployed in two main zones, namely the collecting and the sorting areas. In the collecting area, rescuers equipped with mobile devices collect victims and ensure their follow-up by creating a medical file containing their age, sex and seriousness of injury. Their observances are stored on their mobile equipment. Then, the victim is transported to the sorting area where a physician makes a detailed diagnosis and sorts the victim according to the seriousness of the injury. The diagnosis and prescription are registered in the medical file of the victim. All in all, the 5 relevant operations performed on the medical file are "add a new victim", "add the sex information for a victim", "add the age information for a victim", "add injury seriousness information for a victim", "diagnose a victim". Finally, at the local headquarters, the victim manager can get a global view of the situation and evaluate the final level of the disaster. Some divergence may occur if the rescuer or the physician suffers from network disconnections, due to the crossing of an uncovered zone for instance. Consequently, modifications performed on their terminal will not be broadcasted through the system causing divergence among patient medical file replicas. Our reconciliation service is responsible for providing synchronization to maintain a high level of consistency in the system. A complete scenario can be found in [4].

[6] deals with a European scale for technological crashes, such as explosion or contamination. This scale consists of 6 levels going from 1 to 6 and allows a synthetic estimation of an accident gravity. For each level, the number of involved victims and rescuers required is given. However, these figures include physicians, paramedics, ambulance staff, firemen, policemen, hospital administrative staff, etc. Thus, there are uncertainties on the number of medical rescuers equipped with a mobile device. We assess that 1 rescuer equipped with a mobile equipment is enough for 2 victims.

The main goal here is to measure the integration time in the context of emergency situations management. The hypotheses are the following: 8.000 rescuers are equipped with a mobile terminal and they can perform at least 5 XML operations in order to create and fill up each victim medical file. Figure 10 results show that RIMoCoW is able to integrate 16.000 medical files in roughly 30 minutes i.e. 9s of performation for each one. This time is widely below the time needed to diagnose an injured victim, create, fill-up and broadcast the related medical file. Finally, these figures mean RIMoCoW will never be overloaded so the disaster level 6 involving more than 1.250 victims is easily managed.

5.3 Discussion Regarding Evaluation and Lessons Learned

As surveyed in the literature, SOCT4Mob is the unique suitable algorithm for an automatic conflicts resolution in our specific context. This explains it can not be compared with alternatives. The presented evaluations focused on the synchronization and integration times. Even though the testbed for performance seems to have little to do with the crisis management scenario described, we claim

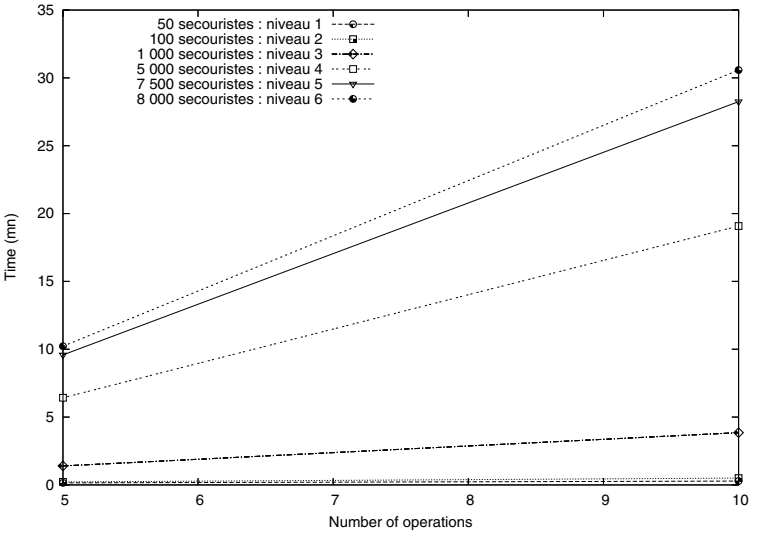


Fig. 10. Integration Time in Medical Emergency Management

the characteristics of the tests environment we have chosen can be projected into the wireless environment.

As we did not have enough mobile devices equipped with wireless communication cards available, tests were performed with a wired set of laptop-class terminals. We have chosen a set of equipment of average power in terms of CPU speed in order to remain near to mobile devices capacity. Concerning hand-held devices memory size limitation, RIMoCoW takes advantage of the *Cache* service which integrates deployment and replacement policies according to the terminal cache size. As we assessed, each rescuer takes care of two victims, consequently, only two medical files of victims that are nursed by a given rescuer are just in time loaded on her device. The size of each medical file is about 2 kb i.e. 4 kb for two files. In addition, we have to take into account operation logging in the different reconciliation service queues i.e. local, receipt and history queues. For one medical file, only five canonical XML operations are needed to create the file with an associated ID, add age, sex, state of gravity information and a diagnosis. Hence, during the rescuer disconnected work period, only ten primitives will be stored in each queue i.e. 12 kb for the three queues. As a result, we can conclude that in spite of memory capacity limitation, light-weight devices can support our testbed hypothesis. Finally, the personal mobility feature is tested in a wired 10 Mbps LAN network. This value is lower than Wifi network bandwidth, which is estimated at 54 Mbps. Consequently, the wired network we use induced a transfer time which is larger than within a Wifi network. As a result, the time results we provide are overestimated in comparison to results which can be obtained in a realistic wireless context. Thus, we consider the applicability of the emergency management scenario within our testbed environment.

We now identify some lessons we have learned from the conducted experiments. First of all, the *Integration* time remains the weak point of the SOCT4Mob algorithm because of the combinatorial explosion generated by the execution of all transformational operations on each site. Hence, the case of a very big collaborative team which produces a lot of operations on a shared application during a long period of disconnection may be problematical due to the large time required for reconciliation. In the situation of emergency crisis management, such a system is not viable.

Nevertheless, RIMoCoW is compliant with the peer-to-peer approach. Thus, it allows the parallelism of the reconciliation process on several sites because each site owns the whole suite of operations. With the distribution of the SOCT4Mob algorithm execution, it takes advantage of the possible parallelization to minimize the global update time. This concept improves SOCT4 vision which is based on a single reference site performing the reconciliation of sites in a sequential manner. Despite the progress provided by RIMoCoW, an optimization of SOCT4Mob algorithm must be planned.

Finally, to reduce notably the reconciliation cost, the design of the target application architecture may be refined. Indeed, at reconnection time, the reconciliation process is launched for each component deployed on the terminal. Thus, the replication of one applicative component gathering together all victims medical files from all rescuers equipments is inconceivable owing to the substantial integration time which will be required to reconcile one by one each medical file, even locally unaltered ones. Emergency crisis management may not tolerate this waste of time. That is why, a just in time replication of only the necessary medical files is recommended to minimize the reconciliation time.

6 Related Work

Several research efforts have focused on finding relevant solutions to manage divergence between data replicas and are compliant to the mobile context. These existing solutions were designed for various fields, such as distributed file systems, database, reflective middleware and cooperative work.

Coda [12], which issues from AFS, deals with distributed file systems. The consistency between files is achieved thanks to a "callback promise" mechanism. In addition, files marked as "inconsistent" become out of reach and may be restored either automatically or manually. However, this solution is client-server oriented. In our work, we consider rather a peer-to-peer model, where sites may collaborate together to achieve a convergent state. Bayou [7] is mainly targeted at collaborative database systems. The convergence is guaranteed thanks to an epidemic diffusion of updates. Bayou servers send out writes thanks to an anti-entropic protocol providing reconciliation. This protocol ensures that databases' copies converge and reach the same state if no new modification is performed. It means servers receive all write requests and schedule them in a consistent manner. However, operations scheduling is arbitrary and does not exploit operations semantics.

The IceCube project [11] offers reconciliation based on the semantic specificities of the applications and the user's intention. During the convergence step, the system merges log files to minimize conflicts. Nevertheless, this solution brings up the combinatorial explosion problem caused by the exploration of all possible schedules, which may not be compatible with the low capacities of mobiles devices.

XMIDDLE [14] is a data-sharing middleware for mobile computing and provides the structuring of data thanks to XML trees. The reconciliation protocol is composed of two parts. The first one is based on general techniques of comparison and merge of XML trees. The second allows handling of reconciliation parameters to resolve conflicts specific to the application. To cope with divergences, the application provides meta-data which define system behavior *a priori*. XMIDDLE benefits from these rules thanks to the reflexivity concept. Although XMIDDLE adapts dynamically, this approach lies in an application-specific way to resolve divergence. In our work, we advocate a reconciliation solution which is as independent as possible from the application.

An important work based on the transformational approach has been performed for reconciliation as surveyed in [18]. The authors of [15] have proposed a generic formal prover to guarantee the accuracy of operational transformations. Our concern is more general and provides reconciliation as an extra-functional middleware service. Furthermore, we focus on extending the scope of SOCT4 to component-based applications in mobile environments. In addition, a SOCT4 adaptation for mobility has been provided in [18], but this proposition remains rather static and centralized and has not been evaluated. The contribution, described in this paper, overcomes these limitations by providing a dynamic, distributed and mobile solution.

7 Conclusion and Open Issues

Replication constitutes a major paradigm in the mobile collaborative field. It allows one to keep working in spite of connectivity changes by deploying a local copy of application components. In this context, the challenge of our work is to provide a suitable system for divergent data reconciliation. To achieve this goal, we have adopted a service-oriented approach to build a new middleware infrastructure, named RIMoCoW.

This paper focuses on the evaluation of RIMoCoW. Thus, we have conducted series of performance tests to measure synchronization and integration times in a general and a medical emergency management contexts. The reconciliation service allows copies of medical files to be kept up-to-date, thus providing a consistent global view of the system. According to the results obtained, the reconciliation of 16.000 medical files is feasible within only 31 minutes for a group of 8.000 rescuers. This shows the capacity of our system to manage a concrete crisis situation of extent 6, where about 1.200 victims must be taken care of by about 7.500 rescuers. Hence, we consider that the SOCT4Mob algorithm is well-suited to distributed collaborative work where disconnection times are short or only a few XML operations are generated during the isolated work period. This

number of operations may vary according to the number of members involved and the integration time which seems reasonable for the client.

As future work, we plan to reduce the integration time by limiting the number of computations needed for transforming operations. We are exploring an optimized XML tree representation of the application, in order to improve the SOCT4Mob *integration* method. Indeed, SOCT4Mob advocates the tree-crossing of the whole list of local operations for each remote operation. We now intend to perform the forward transposition only on dependent operations i.e. on an XML sub-tree. This approach will sizeably reduce the combinatorial explosion.

Concerning performance tests, we used so far wired communication between the equipments. The next step consists in introducing mobile devices, such as PDAs in order to measure the synchronization and integration time with a local wireless network, thanks to WIFI and Bluetooth technologies.

References

1. AMPROS Project Home Page. `http://www-inf.int-evry.fr/AMPROS/`, 2005.
2. M. Bhatti and D. Conan. Fault-tolerance in Mobile Environments: A Partition Detection System. In *Proc. of 2nd Int. Workshop on Frontiers of Information Technology*, Islamabad, Pakistan, Dec. 2004.
3. A. Bouazza and P. Molli. Unifying coupled and uncoupled collaborative work in virtual teams. In *ACM CSCW workshop on collaborative editing systems*, Philadelphia, Pennsylvania, USA, Dec. 2000.
4. L. Chateigner, S. Chabridon, and G. Bernard. A Generic Reconciliation Service for Divergence Management in Mobile Environments. In *International Association of Science and Technology for Development on Software Engineering*, Innsbruck, Austria, Feb. 2005.
5. COACH IST-2001-34445 European Project. Specification of the Container Architecture and OMG IDL3/CIDL Compilers. In *WP2 Deliverable 2.1*, http://coach.objectweb.org/results/COACH-WP2-THC-001o-D21.zip, Sept. 2003.
6. C. Dautun. Crisis Management Organisation : Rescue Forces facing a Crisis of very great Width. Master Thesis (in French), École des Mines, Alès, France, Sept. 2004.
7. A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proc. of the IEEE Work. on Mobile Comp. Syst. and Appli.*, Santa Cruz, CA, 1994.
8. Fractal Home Page. `http://fractal.objectweb.org/`.
9. A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theoretical Computer Science, Elsevier*, 351(2):167–183, Feb. 2006.
10. JGroups Home page. `http://www.jgroups.org/`.
11. A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island (USA), 26-29 Aug. 2001.
12. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *Thirteenth ACM Symposium on operating Systems Principles*, vol. 25, pages 213-225, Feb. 1991.

13. N. Kouici, D. Conan, and G. Bernard. Caching Components for Disconnection Management in Mobile Environments. In *Proc. of International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, Oct. 2004.
14. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE : A Data-Sharing Middleware for Mobile Computing, April 2002.
15. P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the Transformational Approch to Build a Safe and Generic Data Synchronizer. In *Proc. Int. ACM SIGGROUP Conf. on Supporting Group Work*, Florida, USA, Nov. 2003.
16. OpenCCM Home page. `http://www.objectweb.org/openccm/index. html`.
17. Y. Saito and M. Shapiro. Optimistic replication. In *ACM Computing Surveys, vol. 37(1), pp. 42-81*, Mar. 2005.
18. N. Vidot. Copies consistency in distributed collaborative Environments. PhD Thesis (in French), Montpellier-2 University, France, Sept. 2002.

A Component-Based Planning Framework for Adaptive Systems

Mourad Alia[1], Geir Horn[2], Frank Eliassen[1,3], Mohammad Ullah Khan[5], Rolf Fricke[4], and Roland Reichle[5]

[1] Simula Research Lab, Martin Linges v 17, Fornebu, P.O. Box 134, 1325 Lysaker, Norway
mouradal@simula.no
[2] SINTEF ICT, P.O. Box 124, 0314 Blindern, Oslo, Norway
Geir.Horn@sintef.no
[3] IFI, University of Norway
frank@ifi.uio.no
[4] Condat AG, Berlin, Germany
Rolf.Fricke@condat.de
[5] University of Kassel, Germany
{khan, reichle}@vs.uni-kassel.de

Abstract. Recently, many researchers have focused on designing generic and reusable middlewares to overcome the complexity in building adaptive systems. There is a general agreement that the openness provided by component-based approaches coupled with reflection mechanisms is the minimum prerequisites for supporting dynamic reconfigurations. However, this is not sufficient to implement the heart of the adaptation loop namely the decision making on the required reconfiguration that adapts the system in a given context. In this regard, this paper proposes a planning framework that subsumes and automates the adaptation decision-making in reflective component-based adaptive systems. The salient feature of this framework is to model the variability of the adaptive system as a set of variation points at which alternative component compositions and implementations can be selected to form an application configuration. The selection of a feasible configuration in a given context is based on the concept of component wise utility functions that estimates the user benefit of including a specific implementation alternative at a variation point in the composition. We show that the selection problem can be modelled as a multi constraint shortest path that can be found in polynomial time. Our approach is validated through a real world example implementing adaptive scenarios in the domain of mobile computing.

1 Introduction

With the proliferation of mobility, pervasiveness and globalization of computing, the typical operating environment for software is increasingly characterized by heterogeneity and unpredictable dynamic variation both in computing and communication resources and in user needs. In this setting adaptivity becomes crucial and there is a need for better support for constructing adaptive systems. Commonly, the adaptivity is represented by an adaptation control loop that reflects three main functions: (i) detecting significant changes including variations in computational resources, operating

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1686–1704, 2006.
© Springer-Verlag Berlin Heidelberg 2006

environment contexts, and user needs, (ii) planning for the suitable reconfiguration that adapts the system, (iii) and finally executing the reconfiguration steps. This control loop is complex and costly to implement and therefore many research efforts in this domain have dealt with it by concentrating on designing general and reusable adaptive systems and middlewares. These research efforts generally agree that a component-based approach coupled with reflection mechanisms provide openness is the minimum prerequisites for supporting dynamic reconfigurations [1,2,3]. However, this openness is not sufficient to implement one of the complex steps of the adaptation loop control, namely, the adaptation planning which aims at making decisions on the reconfigurations to be performed in order to maintain the usefulness of the application.

Indeed, implementing such an adaptation decision process may be very complex and has not received much attention from the adaptive systems design research community [4]. The planning process is therefore a crucial component of adaptive architectures. The purpose of this paper is to present a planning framework that *encompasses and automates the adaptation decision-making* and provides an *efficient* adaptation policy engine that dynamically manages and performs the required adaptivity for large scale component-based applications. An adaptation policy embedded into the planning framework aims at driving the adaptation by incorporating different contexts including user and application specific semantic knowledge and intelligence, combined with the low-level monitoring of the execution environment. In our previous work, we identified some principles of this planning framework by concentrating particularly on the design [5] and on QoS-based adaptive mobile services [6]. In this paper, we consider reflective component-based adaptive systems architectures and focus on the analysis of the efficiency and the scalability of the planning process by suggesting a new planning algorithm with improved scalability and efficiency properties.

The main feature of the planning framework is to model the different variation points of an adaptive system and then dynamically search and select the best component composition implementation that is most suitable for the current context. It assumes the existence of minimal runtime reflective hierarchical component models such as typically Fractal [7] and OpenCom [8]. The planning framework exploits component composition deployment feature in order to support adaptivity. The selection of a feasible configuration among component implementation alternatives is based on the concept of component wise utility functions that estimates the user benefit of including a specific implementation alternative at a variation point in the composition. We show that the selection problem can be modelled as a multi constraint shortest path that can be found in polynomial time.

The remainder of this paper is structured as follows: the overall planning concepts and approach are presented in section 2. Section 3 formalises the planning problem, analyses its complexity and proposes an algorithm that deals with the scalability issue. Section 4 presents an example that illustrates the usage of the planing framework to adapt a real adaptive mobile application developed in a mobile adaptive middleware MADAM[1] [9]. Section 5 evaluates our proposal both qualitatively and quantitatively. Finally, section 6 presents some relevant related work before concluding in section 7.

[1] MADAM is a European IST Project. Deliverables can be consulted here: http://www.ist-madam.org/consortium.html

2 Planning Framework: Concepts and Terminology

The planning framework is usable with applications that are designed using software components and component frameworks. It assumes a minimal reflective component model, such as Fractal [7], and OpenCom [8] which is used to model and implement component frameworks representing the adaptive applications. This conceptual model (see Fig. 1) is characterized at least by the following properties: *encapsulation*, *structural hierarchical composition*, and *dynamicity and reconfiguration*. By encapsulation, we mean that components interact through explicit bindings only between well-defined access points called ports. A port is a collection of required and/or provided interfaces. The hierarchical composition is one of the main characteristics of this component model where components can contain recursively other components. This recursion ends up with atomic components that may reify hardware resources or directly encapsulate plain software objects (e.g. Java objects). These components are dynamic in sense that they can be created and destroyed at run-time. They also support dynamic reconfiguration routines such as starting, stopping, (un)binding and updating their states. Thus, component frameworks constructed from this component model are predisposed to dynamic reconfiguration and therefore for dynamic adaptation.

In the following, we present the main concepts used by our planning framework namely: *variability*, *component properties*, *utility functions* on component Compositions, and *planning*.

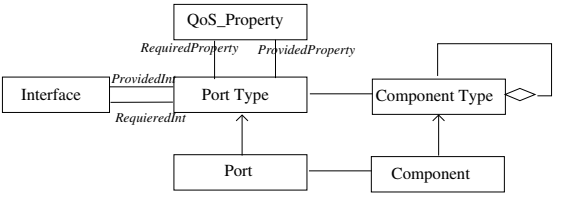


Fig. 1. Hierarchical component model

2.1 Variability

Using the above component model adaptive applications are designed to model their variability in different contexts. This variability is captured using variation points, which designate specific places in a component framework architecture where one or multiple, logically alternative component implementations (termed variants) can be plugged in dynamically. Each variant has a component type that defines its signature and externally observable behavior. A variant can be plugged in at a variation point if the type of the variant conforms qwith type specifed for the variation point. These variation points are used to specify structural and algorithmic variations. So far, this compositional variability has been used to express different adaptations in different domains. For example, [10] presents a video streaming compositional application scenario which specifies different variation points to allow for several alternative video encoding algorithms, client signaling protocols and movie version displaying modes. The case study used in [11] also illustrates somewhat the variability notion through a

review of documentary movie application which supports varying implementations of three composed variation points which are video playing, text editing and browsing. In the context of MADAM project [9,5], this variability is used to overcome resources constraints with regards to the user needs to construct mobile adaptive applications. For this purpose, many mobile scenarios covering different adaptation types (i.e user interface adaptation, disconnections, data streaming, etc.) have been defined [9]. One of these applications, the service technician scenario is taken as an example in this paper and is presented in detail in section 4. Different applications variants consume different amount of resources and provide different quality to the user. The selection of the suitable application variants depends on the different contexts and especially the user preferences and the available resources. In addition to compositional adaptation, component parameters tuning is also supported in order to capture more fine-grained adaptations by modifying programs variables such as buffer size, device parameters, etc.

2.2 Component Properties

To discriminate between alternatives components implementations, components are annotated with non-functional metadata called properties as shown in Fig. 1. The term "property" is used as a generalization of QoS characteristics and can be specified according to a given QoS model [6]. The principle is that instead of forcing the developer to make assumptions about the context in which his components are deployed, we allow him to express the relationship between components and the underlying run-time environment through property descriptions. Properties are associated with component ports and are used to qualify the services provided or required at ports. Properties may be expressed as simple constants, as functions modeling dependencies on other properties of the component itself, or more commonly of composed components.

To support the modelling of such dependencies, we associate property predictor functions with variants. The predictor of a variant predicts the properties of the variant in the current context. For instance, this can be used to express that a specific component implementation requires more than 500 Kbytes of memory or how response time depends on available network bandwidth.

2.3 Component Composition Utility Function

Based on the different context information and the user preferences, a utility function is used in the same way as in micro-economics to quantify the user happiness.

It computes the utility of an application variant as a function of the predicted properties of the different variants and the current context. The aim is to select those variants, which together maximize the total utility of the application while satisfying the resource constraints provided by the underlying environment.

By composing functionally application components we compose and predict also component compositions behavior. We assume that the overall utility of a composition of components can be calculated by summing up the different utilities provided by the different composed components. Depending on the needed granularity, the developer can also assign one utility function for a composition of components considered as a composite without considering the individual utilities of the sub-component but rather their properties and property predictors.

2.4 Planning Process

The planning process proceeds in two phases. The first phase is the discovery and assembly of variants that yields the application variant with highest utility. We refer to this process as resolving variation points. The second phase is the generation of the reconfiguration steps to achieve the adaptation by comparing the newly selected application variant with the current one. These reconfiguration steps make it possible to safely transform and evolve the current running application configuration to the new one. To perform such architectural comparison, we use the same approach as in [12].

In the following, we concentrate only on the first phase, resolving variation points. Since there may exist several implementation of a component type, the resolution of variation points may result in several alternative application variants. The utility function is applied to each of these alternative variants to decide which application variant provides the best utility to the user. In the following section, we analyze the cost effectiveness of the problem of resolving variation points.

3 Algorithms for Adapting the Application Composition

Limiting overheads in resolving variation points is crucial for the usability of the planning framework for the following reasons:

- The planning framework may be used to implement the adaptation control loop in dynamic systems such as mobile computing and everyday systems where exaggerated planning overheads may not be tolerated [13].
- Also, the adaptation process may be executed in a constrained mobile device and therefore the optimisation becomes indispensable.
- Furthermore, our framework combined with components is targeting complex and large-scale systems components that lead to a large alternatives search space. This is particularly the case when there is an extensive use of component parameterization that can lead to a large set of variants differing only marginally in properties. Thus, the planning framework should support optimisations that makes it cost-effective for large scale adaptive systems.

3.1 Problem Formalisation

As the number of variants to consider for an application grows combinatorially with the number of variation points and is in general NP-hard. The formalism of the problem will be introduced via an illustrative example. Say that one wants to construct an application $\mathbf{a}$ consisting of three variation points $\mathbf{a} = [c_{1,j}, c_{2,k}, c_{3,l}]^T$ where each of them can be chosen from three subsets of variants:

$$c_{1,j} \in \mathbb{C}_1 = [c_{1,1}, c_{1,2}]$$
$$c_{2,k} \in \mathbb{C}_2 = [c_{2,1}, c_{2,2}, c_{2,3}]$$
$$c_{3,l} \in \mathbb{C}_3 = [c_{3,1}, c_{3,2}]$$

In other words there need not be the same number of variants in each subset and a variant from one subset can not replace a variant from another subset.

Each variant has a utility value. Typically this is computed at run time as a scalar mapping of the operational parameters for the component. This *utility function* is defined *a priori* by the application programmer for each component. Hence, upon application construction the utility values can be evaluated for each component giving subsets of utility values for $\mathbb{C}_1$, $\mathbb{C}_2$ and $\mathbb{C}_3$ respectively

$$\begin{aligned}\mathbb{U}_1 &= \{u_{1,1}, u_{1,2}\}\\ \mathbb{U}_2 &= \{u_{2,1}, u_{2,2}, u_{2,3}\}\\ \mathbb{U}_3 &= \{u_{3,1}, u_{3,2}\}\end{aligned}$$

It is assumed that the utility of the application $\mathbf{a}$ is the sum of the individual utility values of its components. The goal is to find a component composition that maximises the application utility.

Define binary indicator variables $x_{i,j} \in \{0, 1\}$ such that $x_{i,j} = 1$ if $c_{i,j} \in \mathbf{a}$ and 0 otherwise. For each variation point i, only one indicator is set to 1. The utility of the application can then be written as

$$U(\mathbf{a}) = \sum_{i=1}^{|\mathbf{a}|} \sum_{j=1}^{|\mathbb{C}_i|} u_{i,j} x_{i,j} \tag{1}$$

where $|\mathbf{a}|$ is the number of components in the application, in this example $|\mathbf{a}| = 3$, and $|\mathbb{C}_i|$ is the cardinality of, or number of elements in, component subset i.

3.2 Unconstrained Example

Without any restrictions on how the components can be chosen, it is trivial to maximise the application utility (1). The sum is maximised by selecting the component in every component subset with the highest utility, thus $x_{i,j} = 1$ if $u_{i,j} = \sup_j \mathbb{U}_i$ for a pair of indices i and j.

Obviously the best computational algorithm is to keep the set of utility values sorted such that it is easy to pick out the component corresponding to the largest utility value. For small component sets of say $|\mathbb{C}_i| \leq 20$ applying simple insertion sort of newly computed utility values into linked linear lists that have low structural overhead will give the best performance. One might expect to compare a newly computed utility value with half of the already stored utility values to find the right position in the list for the new value. Hence the algorithm is $O(|\mathbb{C}_i|)$. For larger systems one should use a self-balancing binary tree, of which the one known as the best is the *splay tree* [14]. This kind of structure reduces the complexity of the insertion to $O(\log|\mathbb{C}_i|)$.

3.3 Constrained Version: Greedy and Brute Force Algorithms

One does not normally have the luxury of selecting the optimal components from a subset. This is because the system has a certain amount of resources and each component will consume some of these resources. In the considered example it does not make sense to construct an application where the first two components take all the available resources since the application also needs the third component. Similar to the way the

utility values were defined above, the amount of resources needed by the components in the component subsets $\mathbb{C}_1$, $\mathbb{C}_2$ and $\mathbb{C}_3$ can be defined as

$$\mathbb{R}_1 = \{r_{1,1}, r_{1,2}\}$$
$$\mathbb{R}_2 = \{r_{2,1}, r_{2,2}, r_{2,3}\}$$
$$\mathbb{R}_3 = \{r_{3,1}, r_{3,2}\}$$

Then the problem is to maximise (1) subject to the resource constraint

$$R(\mathbf{a}) = \sum_{i=1}^{|\mathbf{a}|} \sum_{j=1}^{|\mathbb{C}_i|} r_{i,j} x_{i,j} \leq R \tag{2}$$

where R is the amount of resources available for the application. There are several brute force algorithms that can be applied:

- *Minimum resource*: The idea is that for each component class one will select the least resource consuming variant to be included into the composition. Obviously this may satisfy the resource constraint; however it will most likely give the worst performing application with low utility and probably not satisfy the user's demands.
- *Maximum resource*: This is the opposite of the previous approach in that one will select the variant that consumes the maximum resources. This will likely enable the best performing application to be constructed; however there is a significant risk that the resources will be exhausted before any feasible composition has been made.
- *Maximum utility*: The strategy is to select the variant offering the highest user utility fitting within the remaining resources. The target is again to maximize the utility for the user, however the risk remains that all resources are consumed before a feasible application has been constructed.
- *Random selection*: Recognizing the shortcomings of the above strategies, it might be better just to select a variant at random. The argument is that over many adaptations this leads to a composition that consumes an average amount of resources and offers an average utility to the user. Note that this is not a greedy algorithm.
- *Maximum efficiency algorithm*: Let the *efficiency* of a variant be defined as the ratio of its utility to its resource consumption, $\xi_{i,j} = u_{i,j}/r_{i,j}$. Then take the component from each class with the highest efficiency. Note that this approach tries to maximise the combined utility, however there is no guarantee that it will be possible to complete the application composition before the total resources are exhausted.

There is empirical evidence that if the level of offered resources is low compared to the total resources demanded by all variants (the sum of the resource requirements of all selected variants), the minimum resource strategy will perform the best. However, if the resource limit exceeds about 50% of the total resources of all variants, the maximum utility strategy will be the better one [15].

None of these strategies is optimal, and most do not even guarantee to find a feasible solution. One naïve approach would be to improve on an infeasible solution by picking one component of the current composition and replacing it with a variant not

yet included in the composition. Then one will first have to answer how to pick the component to exchange, and then how to pick the variant to replace this component. Many heuristics have been tried to answer these two questions; however there is no clear evidence that it will really improve on the solution found by the greedy min-max alternatives above. If a composition is at all possible, it will obviously be found by the minimum resource strategy as a last resort to solve the problem.

3.4 Constrained Version: Shortest Path Solution

The constrained problem can be solved with a shortest path algorithm on a directed graph [16]. To illustrate the method assume numerical the integer utility values $u_{i,j} \leq 10$ in our example:

$$\mathbb{U}_1 = \{3, 10\}$$
$$\mathbb{U}_2 = \{3, 2, 8\}$$
$$\mathbb{U}_3 = \{9, 1\}$$

The resources consumed by the components are given as

$$\mathbb{R}_1 = \{3, 7\}$$
$$\mathbb{R}_2 = \{3, 1, 5\}$$
$$\mathbb{R}_3 = \{8, 1\}$$

and let the total amount of resources available be $R = 15$. The vertices of the digraph is placed in a co-ordinate system where the abscissa is the component group index and the ordinate values is the total amount of resources consumed in that vertex. As an example, consider the vertex corresponding to the partial composition $c_{1,1}, c_{2,3}$. This vertex will be placed in the co-ordinate system at $(2, r_{1,1} + r_{2,3}) = (2, 3 + 5) = (2, 8)$. Note that the first column of vertices will have $|\mathbb{C}_1|$ vertices, the second column will have $|\mathbb{C}_1| \times |\mathbb{C}_2|$, then $|\mathbb{C}_1| \times |\mathbb{C}_2| \times |\mathbb{C}_3|$ et cetera. Obviously all vertices with ordinate value larger than $R = 15$ need not be included.

There will be a directed edge from a vertex (i, r) to all vertices $(i + 1, r')$ and the weight of this edge will be the negative utility value of the new component included in vertex $(i + 1, r')$. The vertex corresponding to $c_{1,1}, c_{2,3}$ saw the inclusion of $c_{2,3}$ to the application starting from $c_{1,1}$. Hence, the weight of the directed edge between the vertex corresponding to $c_{1,1}$ to the vertex corresponding to $c_{1,1}, c_{2,3}$ will be $-u_{2,3} = -8$. The corresponding digraph of the example at hand is shown in Fig. 2.

The above method can be generalised in the following way: Start with the $v_{0,0} = (0, 0)$ vertex. Let $R(v_{i,j}) = j$ be the resource consumption, the ordinate value, of vertex $v_{i,j}$. Then add vertices column by column such that $v_{i,k} = (i, R(v_{i-1,j}) + r_{i,k})$ where the index $k = 1, \ldots, |\mathbb{C}_i|$ runs over all variant in the set $\mathbb{C}_i$ for the every vertex j in the column $i - 1$. Then add an outgoing link from $v_{i-1,j}$ to each of the $k = 1, \ldots, |\mathbb{C}_i|$ vertices the configuration can be extended with and assign this edge the weight $-u_{i,k}$. Finally, all edges going from the last column to the terminal vertex are assigned weights zero.

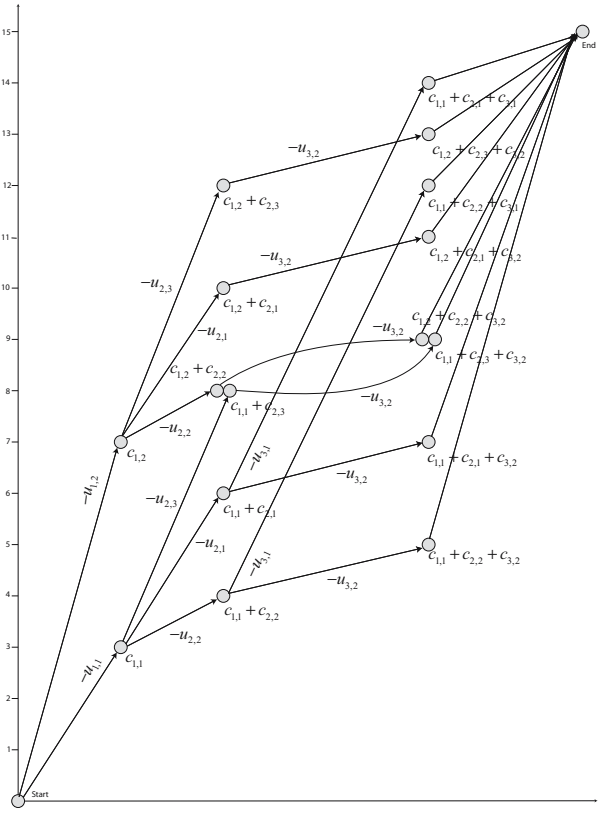


Fig. 2. The digraph of the numerical example showing how shortest path methods can be used to find the optimal configuration. Note that the vertices corresponding to the two partial configurations $[c_{1,1}, c_{2,2}]$ and $[c_{1,2}, c_{2,3}]$ are shown as two vertices for clarity. They should be mapped to the same vertex, and consequently there should be only one vertex after extending these configurations with $c_{3,2}$.

The constructed digraph will obviously be cycle free since all edges leads from a lower indexed column to the vertices in the next column and no edge goes towards a lower indexed column. This allows us to use the polynomial time Bellman-Ford algorithm [17] given in Figure 3. It is a label correcting algorithm assigning a distance to each vertex in the digraph representing its distance from $v_{0,0}$. In our multi-stage digraph it can be simplified as follows: Start with column 1 and set all distance for a vertex $v_{1,m}$ equal to the minimum weight of an edge leading from $v_{0,0}$ to $v_{1,m}$. Note that there can be multiple inbound edges possible in the digraph since there can be multiple component configurations represented by the same vertex. Then continue this process column by column until all vertices have their distances set. Potentially this will give as many distance updates for every vertex in the digraph as it has inbound edges. Thus, the algorithm has an upper bound on its execution time equal to the number of vertices times the number of edges in the digraph.

Input

$\mathbb{V}_k$ Vertices in column k of the digraph

$\mathbb{E}$ Edges of the digraph whose weights are the negative utility values for the new variant of the vertex the edge points towards. The edge label is the variant identifier.

Initialisation

1: Distance labels $d(v_{0,0}, v_{k,m})$ set to infinity for all distances except for the distance from the starting node to the starting node set to zero.

2: **begin**

3: **for** $k = 1$ **to** $k = |\mathbf{a}|$

4: **for each** $v_{k,m} \in \mathbb{V}_k$

5: **for each** Inbound edge $\boldsymbol{e}(v_{k-1,i}, v_{k,m})$ to vertex $v_{k,m}$

6: **if** $d(v_{0,0}, v_{k,m}) > d(v_{0,0}, v_{k-1,i}) + \text{Weight}(\boldsymbol{e}(v_{k-1,i}, v_{k,m}))$ **then**

7: $d(v_{0,0}, v_{k,m}) = d(v_{0,0}, v_{k-1,i}) + \text{Weight}(\boldsymbol{e}(v_{k-1,i}, v_{k,m}))$

8: **if** $d(v_{0,0}, v_{k,m})$ is the the least of all distances in $\mathbb{V}_k$ **then** add the label of the edge $\boldsymbol{e}(v_{k-1,i}, v_{k,m})$ as the composition component at stage k

10: **end**

Fig. 3. The modified Bellman-Ford algorithm to work with the simplified multi-stage digraph of this problem

3.5 Discussion

With abundant resources there will be few nodes in the graph exceeding the threshold R given by (2), thus the graph may contain almost all the

$$\prod_{i=1}^{|\mathbf{a}|} |\mathbb{C}_i| \tag{3}$$

different applications possible. With this combinatorial explosion one might rightfully ask what has been gained by the algorithm.

This approach makes sense since the resource consumption is a property of the involved variants, and hence unlikely to change with the configuration changes. However, any context change requiring the application to adapt boils down to a change in the resource level R. In other words, the digraph itself does not change, but the cut-off ceiling on the digraph changes. With this observation it might be possible to compute the digraph only once and store it for use when an adaptation is needed.

On the other hand, the utility of each component will depend on the resources available under a given context. Thus, the weights of the edges in the digraph might change although the structure of the digraph remains the same. Planning a new component configuration is then a matter of finding the shortest path in this digraph with updated weights on its edges.

Furthermore, if some of the variation points become useless given the available resources its utility becomes zero. Then the algorithm can be optimised to prune first all nodes pointed to by an edge of weight zero. This will exclude all application variants

reachable through this zero edge. This carries some resemblance with the branch-and-bound [18] techniques used in combinatorial[2] optimisation [20]. However, the idea in standard branch-and-bound approaches is to derive upper and lower bounds on the objective function (1) to guide the pruning of the solution space. Furthermore the solution graph is build incrementally as opposed to here where the digraph can be kept between adaptations.

The proposed algorithm also resembles dynamic programming [17] in that a partial solution is extended iteratively until the final optimal solution is found. The main difference is that the proposed algorithm is not trying to increase the current value of the objective function (1), but rather the partially constructed application is extended in each iteration by taking the next group of variation points into consideration.

4 Application

The objective of this section is twofold. In the first part, it gives the big picture of how the planning framework is integrated and used in a typical context-aware adaptive middleware. This is validated in the context of the MADAM project [9]. In the second part, it illustrates how to apply the planning approach as to adapt a real existing mobile application.

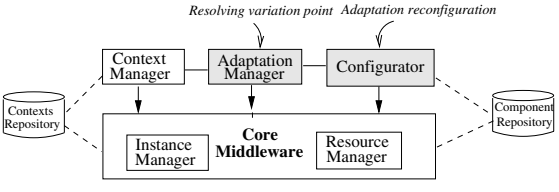


Fig. 4. Planning-based middleware architecture

4.1 Planning in MADAM Middleware

Our planning framework is incorporated within the MADAM middleware architecture [21] that follows (see Fig. 4) the IBM autonomic element architecture[3] applied to components and component frameworks. The central idea behind this architecture is the separation between the configuration mechanisms featured by the underlying platform represented by the core middleware component (resources reification and component instances life cycle control: instantiate, remove, bind, etc) and the adaptation components (Adaptation Manager, Context Manager and Configurator) which encapsulate the adaptation strategies that are plugged on top of the core. The context manager component is responsible for collecting, storing and aggregating and deriving new context information to be used by the Adaptation Manager. Basically, the planning process is

[2] The optimisation is combinatorial because of the integer constraint on the indicator variables $x_{i,j}$. Without this constraint, the problem could have been solved efficiently by approaches like the simplex or interior-point methods known from linear programming [19].

[3] An Architectural Blueprint for Autonomic Computing: http://www.ibm.com/autonomic

represented by the two components the Adaptation Manager which subsumes the resolution of variation points process and the Configurator which subsumes the second step of the planning process which consists of deriving and orchestrating the necessary configuration steps that adapts the application through the core API. Literally, the Context Manager notifies the Adaptation Manager which selects and generates a suitable application variant and ask the Configurator to reconfigure the application to the new one.

4.2 Adapting Service Technician Application

The service technician application is based on an existing business solution[4] and currently implemented as part of the MADAM middleware use cases. It is a solution for service technicians who inspects and maintains of buildings, equipment and vehicles using mobile devices. They can be applied in various working situations, ranging from administrative work in a quiet and connected office environment, through traveling between technical installations in rugged industrial environments with varying or even no network coverage. The devices support planning tasks, loading applications, performing analysis, registering information, tagging, controlling and repairing of equipment. The original implementation of this application did not support adaptation and therefore had many limitations. The usage of the mobile devices was limited, because the offered functions were often not suited for the specific conditions of the working environment. The user interfaces were not adequate for his functions because applications needed for the current task were not available, not enough bandwidth resulted in an inefficient transfer of data and applications did not consider the need for hands-free during some tasks. To overcome these shortcomings, the solution has been improved with the advanced scenes described in the following:

Scene 1 - Morning at home: The service technician checks his assignments for the day before he leaves home, using his company work schedule on his handheld device. There is little memory because of his running video player. In this situation, a thin client configuration (i.e. a light front end on the handheld and complex processing delegated to the server) is chosen for the work schedule application.

Scene 2 - Leaving home: When the service technician shuts down the video player to leave home a context change is raised: the free memory on the handheld becomes high. In order to increase application response time the work schedule is reconfigured with a thick client and caching of data to save power and to become less vulnerable to network instability.

Scene 3 - Driving: He enters his car to go to the company building where a faulty installation is located. While driving he decides to retrieve more information using a hands-free user interface. He selects tools for guiding the inspection assignment and initiates their downloading. The device is now connected through GPRS to the service technician company server. As the cost is high and the capacity of the network low, the downloading of the tools is postponed.

Scene 4 - Arriving at the customer: Arriving at the company site he gets access to WLAN. As this network cannot be regarded as trusted, a VPN tunnel configuration is established. Now he can perform the postponed download of tools.

[4] Business solution from the Condat AG, Berlin.

Scene 5 - Measurement: He starts the measurement of the large buildings ventilation system. In order to reduce battery consumption he switches to a stand-alone mode.

Scene 6 - Measurement analysis: He moves back to his office where he can use a more powerful stationary PC to perform the measurement analysis. The work session is automatically moved from the handheld to the PC and the analysis tool is started.

These scenes make in evidence the adaptive behaviour of the application to allow an efficient contextual usage of the mobile device. This leads to the variability-based architecture shown in figure 5 that expresses the different contextual configurations. The component framework of the application is composed of 7 variation points where each one of 6 of them can have two different alternatives variants. Therefore, one can have a maximum of 64 (1*2*2*2*2*2*2 = 64) application variants.

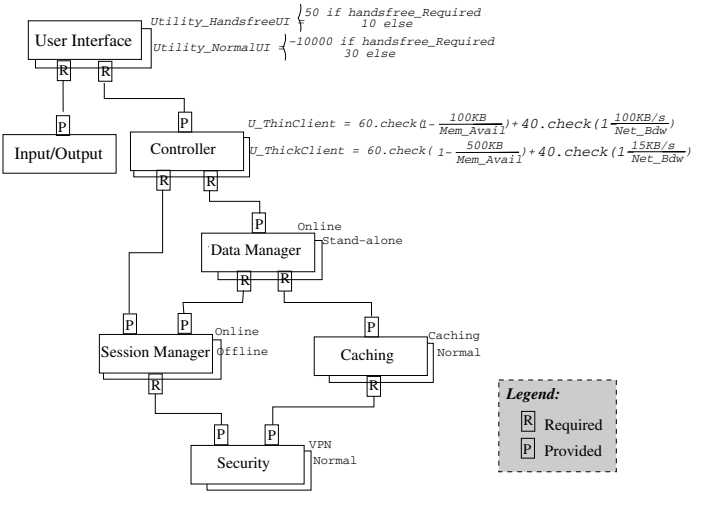


Fig. 5. Application variants configurations

Each of the involved component variants is quantified by a utility function that expresses the service technician user happiness. Fig. 5 shows an example of two utility functions for the variants of the two variation points User Interface (hands-free mode and normal mode variants) and Controller (Thin and Thick Client Controller variants). The utility functions for the two variants of the User Interface are very simple. If hands-free mode is required, the normal User Interface component is not appropriate and not useful, indicated by the value -10000. The high negative value is also intended to guarantee that no configuration using this component can have the maximum utility , which is calculated as a weighted sum of the utilities of the collaborating components. If hands-free mode is not required the hands-free User Interface also works fine, but as it provides only limited functionality it has a lower utility compared to the normal mode User Interface. This is indicated by the values 10 and 30. [5] The utility functions for the two variants of the Controller are more complex, as the context dependencies Memory

[5] Please note: we assume that a utility of 30 means appropriate and 50 means very well suited.

and Network have to be considered. Lower values result in a high negative value of the utility function, as there are not enough resources available for the component. This is guaranteed by the check function. As it has to be considered that other components in the configuration are also supposed to consume memory or network bandwidth, a considerable value for the utility is only achieved if the available memory or network bandwidth is a multiple of the required values. The constants 60 and 40 are gains or weights for the calculated influence of the memory and network dependencies. From the utility function it is obvious, that the Thin Client component is favored in an execution context with low memory and high network bandwidth, whereas in a context with low network bandwidth and a high amount of available memory the Thick Client is preferred.

Experiments on resolving variation points process related to this example are presented in the next section as part of the quantitative evaluation.

5 Evaluation

This section evaluates our framework qualitatively and quantitatively and highlights weaknesse with suggestions for possible improvements and extensions.

5.1 Qualitative Evaluation

- *Compositional general planning*: The planning framework is a compositional-based decision-making that can be applied for hierarchical run-time component model such as Fractal and OpenCom in a modular way. It copes with a wide range of different adaptation domains such as performance, disconnections management by caching or replication, user interface, etc [9]. This is possible since we suppose that applications are designed from scratch using component frameworks and the planning generate the whole application composition configuration that fits the adaptation.
- *Context-driven planning*: Context information is taken as input by the planning framework. The adaptive application is connected to contexts through its components properties in order to reason and select best adaptation configuration alternative. Furthermore, the example presented in section 4 shows how the planning framework can be coupled with a context management infrastructure within a mobile adaptive middleware.
- *Complexity to the developer* : The major developer task consists of variability-based design of the application by providing alternative component frameworks and atomic components for component types, property predictors for the alternative component implementations. New variants of the application or any of its constituent variation points may be published at any time after the application has initially been taken into use that will be automatically discovered when a new planning process of the application is initiated. Adaptation polices are published as utility functions that depend only on the non-functional properties of the different application variants and not on their implementation. Furthermore, the user still reasoning in term of component composition leading the planning to discover and derive the reconfiguration tasks. This is in contrast to action-based policies (see

section 6) that requires detailed knowledge and specification of the low level reconfigurations actions for every adaptation type. A disadvantage of utility function, however, is that they are non-trivial to define for the end-user. Rather we believe a system architect should define these functions for a given application where the user can influence preferences by assigning values to some parameters characterizing the function. Note that defining utility function is also known to be a hard task even in the domain of micro-economics.

5.2 Quantitative Evaluation

The algorithm for resolving variation points, presented in section 3 (see Fig. 3), is extensively tested both on Laptop and PDA. The selection time is measured against the number of possible application composition ranging from a few to several thousand variants. The increase in the selection time is observed and it is analyzed against the scalability issue. For the measurement, we have adopted the example application presented in section 4 as the base. However, more component variants are plugged in for each of the 7 components shown in Fig. 5. We have assigned 5 variants for InputOutput, 6 for UserInterface, 7 for Security and 2 variants each for SessionManager, DataManager, Controller and Caching component in order to consider the scalability issue. This produces a maximum of 3360 (5*6*7*2*2*2*2 = 3360) variants. Each of the component variants has a minimum resource requirement and a utility function. In our simulation we have considered the effect of changes in two resources, namely, available memory and network bandwidth along with the request for hands free user interface, which presents a user need.

Different combinations of these inputs (available memory, network bandwidth and request for hands free mode) produce the number of feasible application variants. These variants construct a digraph like that in Fig. 2 for the current simulated context. Fig. 6 shows the simulation result as two graphs - where the selection time is plotted against the number of application variants - both for the IBM Laptop with Centrino 1.5 GHz processor and 1 GB RAM, Windows XP SP2 using JDK1.3 and an HP iPaq 6340, 168 MHz, 64MB RAM, Windows Mobile 2003 using Crme 4.00. It is to note here that the figure presents the time required only for selecting the best fit application variant according to the presented algorithm. It does not represent the total adaptation time that also considers the time required to build the graph and the time required to apply the adaptation by reconfiguration.

During the first stage of the MADAM project (See [9] for more details), the brute force algorithm was adopted. The measurements employing that algorithm were taken on a Laptop with Pentium III 1066 MHz processor, 256MB RAM, Windows XP SP2 using JDK1.3, while having a HP iPaq of the same configuration. An application having only 20 variants was tested and the average selection times were found as 50 ms and 330 ms for the laptop and the PDA respectively. From figure 6 , we can see a significant improvement in the selection time compared to that required applying the brute force algorithm. For both the Laptop and the PDA, the selection time increases in a polynomial way with the increase in the number of application variants, which is in concordance with our expectation. For the laptop, we can support almost 900 application variants within a 50ms selection time. The figures are considerably improved for

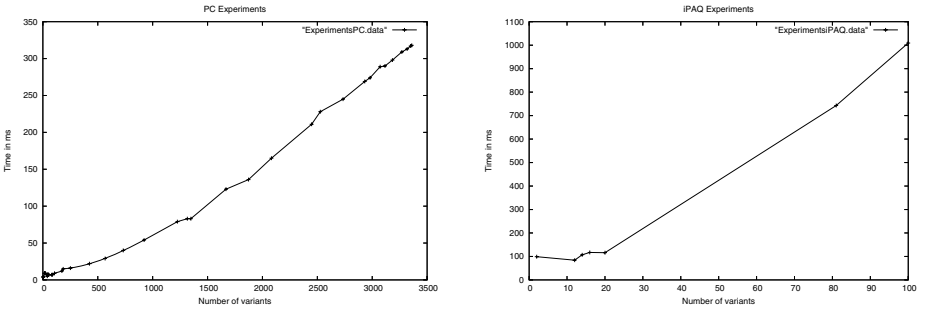


Fig. 6. Resolving variation points scalability

the PDA as well; requiring only 116ms for 20 variants. In order to test the scalability problem, we can see that for laptop, it takes only little over 300ms for 3360 variants and the figure is around 1 minute for the PDA. In Fig. 6, we have plotted the simulation result for up to 100 variants for the PDA. Therefore, for laptop the proposed algorithm solves the scalability problem resulting from multiple applications in good extent. For PDAs, if there were several thousand possible variants, then the algorithm might not be able to provide a fast enough solution. However, from our experience with the mobile devices, we usually don't have applications with many variants and therefore the algorithm is very much applicable. In any case, it improves the earlier development quite significantly.

Since utilities are calculated at run-time, the selection time will slightly vary with the complexity of the utility functions of the visited variants. This is reflected through some graph fluctuations. Furthermore, note that in the current implementation, one doesn't consider paths buffering at the node level which may improve the results. Considering such technical heuristics will probably affect device memory resource which is not recommended.

6 Related Work

We approached the state of the art by presenting planning approaches used in some adaptive policy-based middlewares [22,23,11,1] and relate to researches that addresses generally the problematic of planning in autonomic systems and in particular those using AI planning techniques [4,24].

As one of our objectives, Rainbow [22] addresses also the problem of constructing general adaptive systems by following an component-based approach. However, its planning is action-based where the adaptation policies are represented as conditions-actions patterns like besides many other projects [23,2]. While this kind of policy is natural and simple to declare, it presents some drawbacks compared to our utility function-based policy approach as also pointed in [25]. Firstly, it provides less transparency to the developer which by reasoning in terms of reconfiguration (i.e. actions) instead of architectural design (i.e. variability). Secondly, it handles only a finite number of actions whereas the utility function approach allows choosing the best adaptation

among a huge possibilities. Finally, with the multiplicity of the actions, this approach can be complex to manage by the user. However, note that our planning framework can be adopted by the Rainbow middleware.

CARISMA [1], a mobile computing middleware targets adaptivity in presence of multiple users instead of one user as in our approach. The planning consists on choosing among a handful of predefined policies so as to resolve conflicts and to maximize an objective function of multiple users using microeconomics techniques.

[11] addresses the problem of automatic and dynamic configuration of resource aware services in ubiquitous computing environments. The proposed approach involves also the selection of the application services composition alternatives. However, it does not consider the design issue of adaptive systems and focus only on resources constraints adaptation type. The proposed algorithm for dynamic composition of resource-aware services seems interesting and it would be a nice exercise to compare this market-based algorithm with ours. However, this is not trivial because the measurements given in [11] are in term of number of visited application variants whereas our algorithm is measured in term of time and it has no sense to measure it in term of visited variants.

In AI planning community, there is a particular interest in solving the planning problem in general [4,24]. Commonly in the AI community, a plan corresponds to a sequence of tasks that transforms the system from the current state to the desired one. In our approach this sequence corresponds to the second phase of planning resulting from the comparison of the old and the new generated configuration. For example [24] uses AI planning approach in pervasive computing by using PDDL language to model tasks, and STRIPS-planning as a plolicy engine that generates the plan. While this work demonstrates the feasibility of the approach the scalability stills an open issue and more investigation is needed to see how our planning framework can be projected efficiently on such AI planning models.

7 Conclusion

To go a step beyond the current architectural approaches for handling the complexity of constructing adaptive systems, we propose a planning framework which subsumes and automates the decision-making on the adaptation. The planning framework aims to be coupled with the software component and component framework architectural paradigms, while offering a domain-independent planning process that exploits component compositions to adapt the system with respect to the defined policy. The main planning process consisting of resolving variation points involves dynamic discovery of application variants and the selection of the one that best matches the operational environment and provides the best user utility. While the analytical model shows that this process is NP-hard, a methodology and a multi-constraint optimal path-like algorithm are proposed to deal with the scalability. We demonstrate our approach through a real industrial case study that illustrates how to use the planning framework in order to implement the adaptivity in a typical policy-based middleware. The results obtained through the demonstration are quite satisfactory and we believe that the proposed methodology and algorithm will come as a good solution to the scalability problem.

Currently, we are porting our planning approach to some other component models and technologies and particularly to OSGi bundles [6]. In the short term, we intend to distribute the planning framework to target particularly decentralized and autonomous environments. Algorithms and on component placement and collaboration strategies will be exploited for architecting such decentralized decision making much more as in adaptive peer to peer and grid computing.

Acknowledgements

This work is part of the MADAM project and would like to thank all involved partners.

References

1. Carpa, L., Emmerich, W., al: Carisma: Context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering **29** (2003) 929–945
2. David, P.C., Ledoux, T.: Towards a framework for self-adaptive component-based applications. In: Proceedings of Distributed Applications and Interoperable Systems (DAIS'2003), Paris, France (2003)
3. Zachariadis1, S., Mascolo, C., Emmerich, W.: Satin: a component model for mobile self-organization. In: Proceedings of CoopIS, DOA and ODBASE. (2004)
4. Kephart, J.O.: Research challenges of autonomic computing. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 15–22
5. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjrven, E.: Beyond design time: using architecture models for runtime adaptability. IEEE Software (2006)
6. Amundsen, S.L., Lund, K., Eliassen, F.: Utilising alternative application configurations in context- and QoS-aware mobile middleware. In: 6th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'06), June 13-16, Bologna, Italy, Lecture Notes in Computer Science, Vol. 4025, Springer Verlag (2006) 228–241
7. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Proceedings of the International Workshop on Component-Oriented Programming (WCOP02) Ecoop. (2002)
8. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A component model for building systems software. (In: Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge MA, USA)
9. Madam Consortium: (Mobility and ADaptation enAbling Middleware. Delivrable are open here: http://www.ist-madam.org/consortium.html)
10. Amundsen, S., Lund, K., Halvorsen, P., Griwodz, C., Eliassen, F.: Scenario Description -Video Streaming in the Mobile Domain http://vefur.simula.no:8888/QuA/uploads/2/techVScenA1.1.pdf. Technical report (2005)
11. Poladian, V., Sousa, J., Garlan, D., Shaw, M.: Dynamic configuration of resource-aware services. In: Proceedings of the 26th International Conference on Software Engineering (ICSE). (2004)
12. Dashofy, E.M., der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: Proceedings of the first workshop on Self-healing systems, WOSS. (2002) 21–26

[6] The OSGi Service Platform - Dynamic services for networked devices: http://www.osgi.org

13. Hallsteinsen, S., Stav, E., Floch, J.: Self-adaptation for everyday systems. In: proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, WOSS'04, New York, NY, USA, ACM Press (2004) 69–74
14. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the Association for Computing Machinery **32** (1985) 652–686
15. Andersen, H., Brseth, A.M., Nytun, J.H.: The knapsack problem: Main project in computer technology. Vestfold University College: http://www-bib.hive.no/tekster/studoppg/ri/2003/andersen.pdf (2003)
16. Schrijver, A.: A Course in Combinatorial Optimization. University of Amsterdam (2004)
17. Bellman, R.: On a routing problem. Quarterly of Applied Mathematics **16** (1958) 87–90
18. Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28** (1960) 497–520
19. Vanderbei, R.J.: Linear Programming: Foundations and Extensions. International Series in Operations Research Management Science (Frederick S. Hillier). Springer-Verlag, Berlin (2001)
20. Cook, W.J., Cunningham, W.H., Pulleyblank, W.R., Schrijver, A.: Combinatorial Optimization. Wiley-Interscience Series in Discrete Mathematics and Optimization (R. L. Graham, J. K. Lenstra and R. E. Tarjan. John Wiley & Sons, New York (1998)
21. Alia, M., Eliassen, F., Hallsteinsen, S., Stav, E.: Madam: Towards a flexible planning-based middleware. In: proceedings of the ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), May 21-22, 2006, Shanghai, China, ACM Press (2006)
22. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37** (2004) 46–54
23. Huang, A.C., Steenkiste, P.: Building services using service-specific knowledge. In: proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, July 2005. (2006)
24. Arshad, N., Heimbigner, D., Wolf, A.L.: Deployment and dynamic reconfiguration planning for distributed software systems. In: proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA, IEEE Computer Society (2003) 39
25. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: proceedings utility functions in autonomic systems. In: In proceedings of the First International Conference on Autonomic Computing (ICAC'04. (2004)

A Case for Event-Driven Distributed Objects

Aliandro Lima, Walfredo Cirne, Francisco Brasileiro, and Daniel Fireman

Laboratório de Sistemas Distribuídos
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
58109-970, Campina Grande, PB, Brazil
{aliandro, walfredo, fubica, fireman}@dsc.ufcg.edu.br

Abstract. Much work has been done in order to make the development of distributed systems as close as sensible to the development of centralized systems. As a result, there are today good distributed object solutions that closely resemble centralized programming. However, this very attempt to mimic centralized programming implies that distributed objects create the illusion that threads traverse the whole distributed application. This brings all the problems related to multi-thread programming, including the need to reason about the thread behavior of the *whole* application, which gets amplified by the large scale and inherent non-determinism of distributed systems. Moreover, distributed objects present other troubles when the application is not pure client-server, i.e., when the client has other things to do besides waiting for the server.

As an alternative, there are a number of message-based non-blocking communication solutions. Unfortunately, these solutions were not designed to directly address the above mentioned issue of multi-threading over the whole distributed application. In addition: (i) these solutions are not as well integrated to the programming language as distributed objects, and (ii) most of them do not provide a well-defined embedded failure detection mechanism, something that is crucial for the development of many distributed systems, and that is well solved by distributed objects (as they couple method invocation and failure detection).

We here propose and evaluate an improvement for such a status-quo, named JIC (Java Internet Communication). JIC is an event-driven middleware that relies on a non-blocking communication model, yet providing close semantics to the object-oriented paradigm. JIC is designed to combine the best characteristics of distributed objects and message-based solutions. For instance, JIC defines precise scope for the application's threads, promotes non-blocking communication, provides a failure detection service that is simple to use with precise semantics, and has performance comparable to Java RMI. Furthermore, JIC is designed to be firewall and NAT friendly, greatly helping the deployment of JIC-based applications across multiple administrative domains.

1 Introduction

In the last decades, much work has been done in order to make the development of distributed systems easier. Indeed, a great number of those efforts relied on

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1705–1721, 2006.
© Springer-Verlag Berlin Heidelberg 2006

papering over the distinction between centralized and distributed programming. Actually, if distributed programming were as simple as centralized programming, distributed system developers could take full advantage of the object-oriented paradigm and write their programs without worrying about object location.

However, programming a distributed system can not be as simple as programming a centralized system. There are fundamental characteristics that are not present in a local environment, like *partial failure* in absence of a central manager, *mandatory concurrency* and *partial connectivity* (the last one due to the increasing use of firewalls and NATs). Therefore, unifying the programming models for local and distributed objects is doomed to failure if the particular characteristics of distributed systems are not considered [1]. This means that such a unification, in the best case, would make local programming as complex as distributed programming.

Such a realization allowed for great progress with distributed objects [2] as some solutions can cope with the differences between distributed and centralized programming, while integrating well to the programming language (e.g., Java RMI [3, 4] and CORBA [5]). These solutions provide middlewares that hide the basic building blocks of processes and message passing, providing a distributed programming model as close as sensible to the object-oriented paradigm. This is done via a blocking communication paradigm, in which a *client* invokes a *server* and blocks until receiving a response or detecting a failure.

Note that the blocking *client* thread gives the illusion that the thread goes beyond the local address space, traversing the distributed application. In reality, each invocation is remotely performed by a new thread. Because of that, the *server* application is forced to be multi-threaded even if it could otherwise process remote requests sequentially. Much worse, since the new thread can be thought of as an extension of the invoking thread, the programmer must reason about the thread behavior of the whole distributed system as to assure that thread-related problems do not occur. In particular, one cannot combine two correct components and know if the result is still correct without examining the two codes [6]. The problem arises because one has no control over how a third part code acquire locks, and thus the composition of components may lead to distributed deadlocks. In short, if centralized multi-thread programming is already complex and failure-prone [7], distributed multi-thread programming only makes matters worse.

Moreover, for applications that are not "pure" client-server ones, i.e., when the client has other things to do but to wait for the server (e.g., interactive, peer-to-peer and performance-constrainted [8] applications), the blocking nature of distributed objects invocation does not fit well. The most obvious way to circumvent this problem seems to be creating an extra thread to block on each invocation, releasing the original thread to continue working.

However, this solution has a series of drawbacks. First, creating a new thread for each new invocation may lead to a thread explosion in the *client* if it is performing a great number of simultaneous invocations. Moreover, if the *server* needs to callback the *client*, communication initiation should still be

possible in both directions (from *client* to *server* and vice-versa), which is typically a problem due to firewalls and NATs. Finally, since the application thread does not block, the failure detection mechanism must be augmented. Note that, in distributed objects, the failure detection mechanism is embedded in the remote method invocation. If the invocation is done through a new thread, there should be a mechanism to notify the original thread when a failure occurs.

A possible way to deal with the problems of distributed objects is to avoid them altogether, and instead, use message-based communication solutions. There are a number of message-based communication alternatives available, ranging from sockets to sophisticated Event-Based Middleware (EBM) [9, 10] and Message-Oriented Middleware (MOM) [11,12], passing by parallel programming support as PVM [13] and MPI [14]. These solutions do not block waiting for results, and thus some of the problems described above simply do not exist (thread explosion, for example). On the down side, they do not integrate with the programming language as nicely as distributed objects.

In 2005, we faced the need for a communication solution that could solve the problems discussed, but yet remaining lightweight and working well in environments composed by multiple administrative domains (read firewalls and NATs are present) where failures are the common. Such a need came from the OurGrid project, a free-to-join peer-to-peer computational grid in which research labs donate their spare computation resources to other labs [15]. OurGrid was based on Java RMI and although it is in production since December 2004, the problems of using distributed objects were making the system brittle and hampering its evolution [16].

It was clear that the communication solution we needed was message-based. However, we could not find an existing message-based solution which was a good match for OurGrid needs. The problem was that existing solutions were designed with other goals in mind. Sockets simply aim to expose the transport layer to application programmer. EBMs and MOMs try to decouple the components of a distributed system via the *publish-subscribe* paradigm, often augmented by support services, like store and forward, messaging routing and transaction management, done by integration containers. PVM and MPI, on their turn, are designed to facilitate collective operations, hence facilitating the development of parallel applications. For example, none of them has good support for firewalls and NATs traversing, or aims to scope out the threads of a distributed application.

JIC (Java Internet Communication) then appeared as a solution for our needs in OurGrid. And, since we had the chance to develop a novel solution, we made it well integrated with the programming language (in the case, Java), challenging the belief that message-based solutions must rely on send/receive primitives. The key point of keeping a good integration to the programming language is to allow for the type-checking system to work also for the interaction among distributed components.

As we show in details throughout the paper, JIC (i) is event-driven (i.e., a message-based solution on which messages are consumed by threads that process them in an event loop); (ii) relies on a non-blocking communication model, yet providing close semantics to the object-oriented paradigm; (iii) creates explicit scope for threads, called *access points*; (iv) is NAT and firewall friendly; (v) provides a failure detection service that is simple to use and has precise semantics; and (vi) has performance comparable to that of Java RMI.

We are currently working on migrating OurGrid [15] from RMI to JIC. This on-going experience suggests that JIC can be a very interesting solution for applications that are not well served by distributed objects or other event-based solutions. We conjecture that JIC should serve well most applications that remain being written at the socket level, like most peer-to-peer applications.

The remaining of this paper is organized as follows: the next section introduces the JIC solution, highlighting how the discussed problems are addressed and explaining the semantics that JIC provides to the programmer that develops a distributed application using it. We also explain how this semantics is achieved. After that, in section 3 we present the results obtained from a performance and software engineering evaluation over JIC. Then, we provide some discussion about related work in section 4 and finally conclude in section 5.

2 The JIC Solution

JIC's main goal is to provide non-blocking access to Java objects located in a distributed system in a way that threads can have a well defined scope. As such, we require that JIC objects only carry void-returning methods, i.e., methods that provide no return. Thus, the client object can promptly proceed to the next line of code. The server object will have the target method eventually invoked.

Clearly, JIC objects are different from Java objects. Besides the required void type, the methods of JIC objects throw no exceptions. A little more subtle (and more important) difference is that the thread from the client (invoker) object never goes into the server (invoked) object. Invocation only marks the invoked method to be executed with the server own thread (actually, with the access point thread, as we shall see briefly). Nevertheless, JIC provides automatic object marshaling (saving the programmer from creating tedious "message" classes) and enables the Java type-checking to work. Type-checking greatly helps to capture bugs in early development stages and is this characteristic that makes us consider JIC well-integrated to the language. Note that the automatic object marshaling and type checking are possible due to the use of *stub* objects, which in JIC are automatically generated in run-time.

A JIC object must implement the **EventProcessor** interface, which denotes that the object can be exported by JIC (as **Remote** denotes an RMI object), and contains a few methods that must be implemented by any JIC object. The code fragment below shows an excerpt of the OurGrid EventProcessor that performs job submission.

```
public interface MyGrid extends EventProcessor{
        public void addJob(JobSpecification jobSpec);
}

public class MyGridImpl extends SimpleEventProcessor
                                    implements MyGrid{
        public void addJob(JobSpecification jobSpec){
           ...
        }
}
```

Notice that the **MyGrid** interface extends **EventProcessor**. In addition, the **MyGridImpl** class implements the **EventProcessor** interface by extending the **SimpleEventProcessor** class. **SimpleEventProcessor** is a default implementation for the **EventProcessor** methods and provides a specialized semantic for remote objects, overriding the default local semantic from `equals`, `toString` and `hashCode` methods. It was developed to avoid the burden of programming that for each application EventProcessor. Hence, by extending **SimpleEventProcessor**, it is possible to inherit the semantics provided by JIC.

2.1 Connectivity

JIC is built on top of Jabber [17, 18], which provides a good substratum for firewall- and NAT-friendly messaging passing. As such, this is the infrastructure used by the EventProcessors objects so as to communicate. Jabber is a set of streaming XML protocols, mostly used for instant messaging, that enables entities to exchange messages close to real time. Its architecture is something similar to the e-mail architecture. If two entities want to communicate, they must create an account in a Jabber server, receiving a Jabber Identification (JID). An entity that has a JID and connects to a Jabber server is named a *Jabber client*. JIDs are based on DNS and recognized URI schemes, in a manner that they have the same form of an e-mail address, as shown below:

```
node@domain/resource
```

In the JID above, `domain` is a Fully Qualified Domain Name (FQDN) for the Jabber server, `node` is the client that connects to the server and `resource` is a session-specific object that belongs to a client, such as a device or location. Therefore, a message sent from a Jabber client to another, first goes to the server of the sending client, which forwards it to the server of the recipient client, which finally relays it to the recipient client.

Note that the use of Jabber server as relay greatly reduces the problems posed by the use of firewalls and NATs. The problems with firewalls and NATs arise because they limit the connectivity between peers, typically making possible to start a connection from only one direction. Since every EventProcessor communicates through a Jabber server, one needs to open just a *single* port per

administrative domain, no matter how many objects are exported by an application. This port is exactly the one that makes the Jabber server accessible by other Jabber servers. Indeed, if the instant messaging port (5222) is opened by default, Jabber communication can be done without any additional administrative effort.

2.2 Access Points

Since JIC uses Jabber as a NAT and firewall friendly communication substratum, JIC method invocations are sent to Jabber clients. Each Jabber client used by JIC is known as an *AccessPoint*. An access point is the entry point for a set of JIC objects. In order to receive remote invocations, a JIC object must be exported by some access point. The following code fragment creates an access point and exports OurGrid job submission object in the access point **peer**.

```
...
MyGrid mygrid = new MyGridImpl();
String myGridName = "mygrid";
...
String accessPointName = "john@lsd.ufcg.edu.br/myGridAP";
AccessPoint accessPoint = new AccessPoint(accessPointName);
accesspoint.start();
accesspoint.bind(myGridName, mygrid);
```

Each access point contains a queue of invocations to be performed on the objects it holds, as well as a set of threads that perform such invocations, as depicted in Figure 1. That is, any invocation to a method m that belongs to object *obj* that is exported via the access point *ap*, is marshaled by the JIC runtime, transferred to *ap* using Jabber, and stored at *ap*'s invocation queue. The threads that belong to the access point *ap* remain in an eternal loop: get invocation from the queue and perform it. Note that the invocation queue is processed in FIFO order. Therefore, if *ap* was created with t threads, and there are $t + u$ not yet completed invocations of *ap*'s objects, the last u invocations remain in the queue waiting for the t threads to finish with their invocations. In particular, if *ap* has a single thread (i.e., $t = 1$), the JIC objects exported by *ap* do not need to worry about synchronization and thread safety.

Note that an access point is a self-contained component that sets boundaries for its threads. It is exactly such a feature that makes threads more tractable in JIC-based applications. In JIC, threads do not travel over the whole distributed application. Instead, each thread remains confined within its own access point. This render thread usage composable and modular in JIC. One does not need to understand and reason about the thread behavior of the whole application. This can be done one access point at a time. Note, however, that if two access points are running on the same JVM, it is currently up to the programmer to ensure that the threads from one access point do not traverse over objects of the other access point. We are currently working on automatic ways to instrument the code and use monitoring techniques to detect violations on this [19].

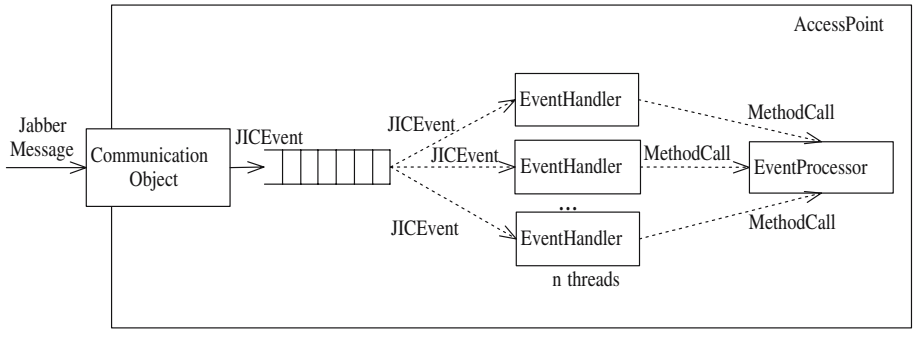


Fig. 1. Access point overview

2.3 Services and Objects

JIC objects are said to provide a *service*, i.e., a computational task invoked by another program. We differentiate between services and objects because a given JIC object may go down and have to be replaced by another object in the future. In fact, this commonly happens in practice due to a number of reasons: power failure, reboots to install patches, software rejuvenation, etc. The *service* is the permanent entity that fulfills a given role in the distributed system (e.g., the name service for a domain). The *object* is the Java object that at some point in time is in charge of a service. At any moment in time, a service is provided by (at most) one object.

The service id is formed by the access point id together with the service name (which is given by the application) and is a globally unique identifier. The object id is formed by the service id together with an incarnation number. JIC objects can choose their incarnation number. This is meant to be used when the service is implemented as to save the state of the object that provides it, as to recover this state when a new object takes over the service. In such a situation, moving from one object that provides the server to another is just seen as a temporary service unavailability.

When the object that implements that service is stateless, however, it will have no history of the past interaction with remote objects and, therefore, these remote objects must know the object they were interacting with no longer exists. In this case, the incarnation number can be randomly generated when the object is being exported. In such a manner, when the service reboots, it receives a new incarnation, denoting that there is a new object providing the service and there is no state about past interaction with this service.

In short, the differentiation between services and objects is needed because a remote object may want to know that a service it is using has just rebooted. If such a service is stateless, the remote object likely wants to take actions, such as resubmit pending requests. This way, JIC lookup is done on a service, but it returns an object. JIC communication is always among objects.

2.4 JIC Semantics

Understanding the *precise* semantics of the communication infrastructure is a key for developing robust applications. JIC is no different. One must understand the semantics provided by JIC to develop an application using it. By doing that, it is possible to understand how the communication among objects happens and in which situations the JIC infrastructure will detect a failure.

Consider a scenario in which there are two JIC objects A and B, exported within access points AP_A and AP_B, respectively, intended to communicate to each other. Access points AP_A and AP_B can be the same, distinct access points in the same address space, or in different address spaces, possibly in distinct machines. Communication is done through remote method invocation, such that an invocation is marshaled into a message and sent to the receiver side, where it will be placed in the access point queue to be consumed in FIFO order. Every parameter but EventProcessors is passed by value. EventProcessors, since are remote objects, are passed by reference. Given that, we have some definitions:

Let S_A and S_B be the Jabber servers used by the access points AP_A and AP_B, which contains the objects A and B, respectively. A *path* between two JIC objects A and B is composed by all Jabber elements that are involved in the communication of these objects. Thus, we can say that **there is a path** between A and B if and only if S_A and S_B are up and all the communication links are active, i.e., they are able to communicate. Similarly, we say that **there is no path** between A and B if S_A or S_B is down, or even if at least one communication link is not active. If there is a path between A and B, we say that A is connected to B, which we denote $connected_A(B)$.

Obviously, a path between two objects is of little use if both objects are not *available*. We say that an object is **available** if the object is exported via an access point. In the same way, we say that an object is **unavailable** if the object is not exported via an access point. Of course, if the access point is down, the object is unavailable. For an object A, the notation of its state is represented as $available(A)$ if the object is available and $\neg\, available(A)$ otherwise.

Now, we can define the *reachability* relation between JIC objects, that will be of great importance for understanding JIC semantics.

$$reachable_A(B) \models available(A) \wedge available(B) \wedge connected_A(B)$$

Given the above definitions, JIC provides precise semantics to the programmer that writes a distributed application. Considering that an EventProcessor A has registered interest in the failure of an EventProcessor B and vice versa, JIC guarantees that:

1. if B stays reachable from A, then every sent message from A to B is eventually received by B. Messages between two JIC Objects are delivered in a FIFO order;
2. if B becomes unreachable from A, then A will eventually be notified that B is unreachable. In this case, we say that A suspects B.

3. if there is at least one message lost from A to B, A will eventually be notified that B is unreachable. In addition to it, B will also be notified that A is unreachable, i.e., A suspects B and B suspects A.

2.5 Assuring the Semantics

Note that "A suspects B" does not guarantee that B has become unavailable. It is possible that a message from A to B has been dropped in the network, or B is too slow in responding the failure detection messages from A. This means that the programmer must reason about the application knowing that either the invocations are successful or the caller is going to be notified of a failure. However, being notified does not assure that a failure happened.

In order to avoid any inconsistent behavior, and guarantee that the above semantics hold, JIC has a failure detection mechanism that implements a probabilistic model described in [20]. Our implementation provides a flexible framework in which it is possible to balance the trade-off between detection time and wrong suspicions. Therefore, JIC access point provides a high level failure detection interface that makes possible for an object to be interested in an object failure or a service recovery.

Once A registers interest in the failure of B, the local AP_A failure detection mechanism starts monitoring B. This monitoring is done in a pull model [21], in which the AP_A failure detector periodically request information about the availability of B. The local AP_B receives these requests and responds whether B is alive or not. If there is no response up to a threshold (that can be application specific), AP_A will suspect that B has failed.

Nevertheless, there is a situation in which B stays available and reachable from A, but does not receive a message due to a brief network failure. In that case, the AP_B failure detector was able to respond AP_A monitoring messages, although B has failed on receiving an application message. Indeed, AP_A needs to detect such a failure, but cannot do it only by using the failure detection messages. Because of that, we implemented a handshake protocol similar to TCP three-way handshake [22] that establishes a connection between two access points before objects start to communicate. This handshake is responsible for defining sequence numbers for messages between two access points. This means that every application message carries such a sequence number in a way that it is possible for the other side to identify if a message was lost (Since messages between two access points are delivered in a FIFO order, a sequence number greater than the expected denotes a lost message). In this situation, the access point that identified the message lost (AP_B) releases the connection (also using a 3-way protocol) and notifies B that A is unreachable. The same happens for AP_A, after finishing the release protocol. Then, both A and B are notified that the other is not reachable, as stated in the JIC semantics. It is important to notice that the application does not need to know which message has been lost. It only needs to be notified that an object of interest has failed.

Note that, although we are using TCP between the access points and the Jabber servers, we can not simply rely on the TCP three-way handshake protocol.

This happens because we do not have an end-to-end TCP connection, but one point-to-point connection between each access point and its server, and between the server themselves.

3 Evaluation

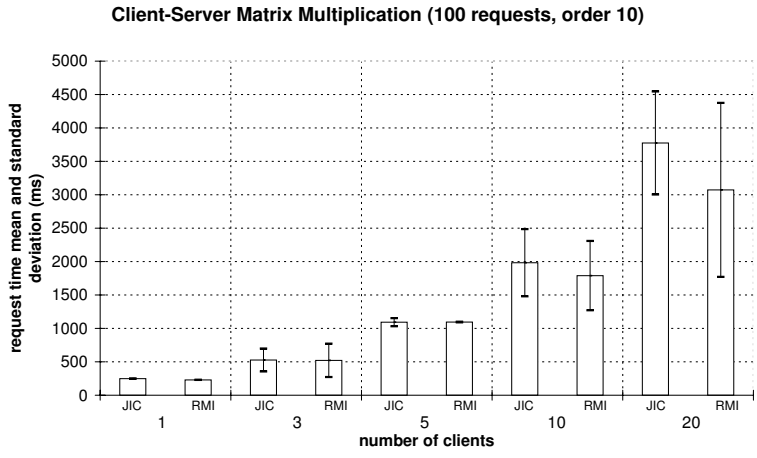
In this section, we evaluate the use of JIC against Java RMI, which is a representative well succeeded example of distributed object middleware. The experiment focuses on performance, but we also provide a preliminary assessment of the software engineering benefits of setting scope for threads.

The performance evaluation consisted on running an application that multiplies matrices. Basically, the application consists on reading two matrices from the disk (randomly reading numbers from a very large file), multiplying them, and calculating the greatest sum of elements from a column in the resulting matrix. For each request, this procedure is done twice, such that, in the first time, it is done with concurrency control (there is a critical region protected by a java synchronized block). The greatest value between the two executions is returned as result. We believe that such application is representative of real workloads as it combines processing within and without critical regions, as well as disk access with processing.

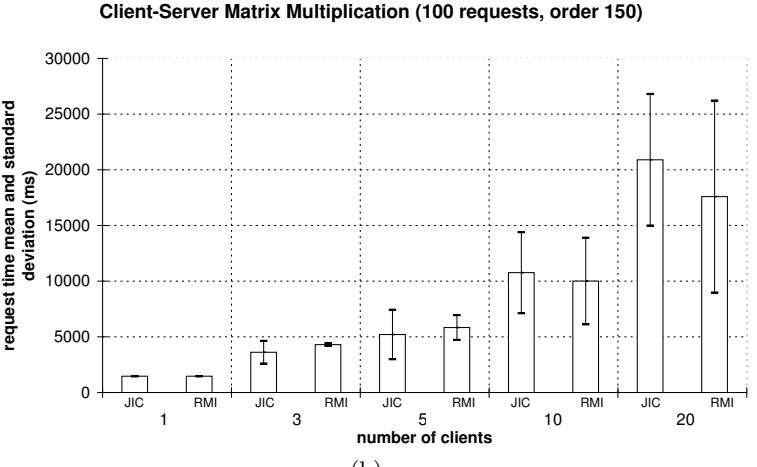
The experiment was executed in a local area network using machines with about the same configuration. Timing measurements were performed in machines with no users, such that they are not influenced by other processes. Moreover, the experiments were conducted in periods of light network traffic, as to reduce communication contention.

We divided the experiment in two scenarios: *client-server* and *peer-to-peer* applications. In the former, clients performed requests on a single server, asking it to multiply matrices of a specified order. After processing, the server returned a result, which delimits the end of a request. In the latter type of application, there were multiple peers running. Peers produce requests, which are processed locally and forwarded to other three random peers, a process that is repeated until each request reaches the entire set of peers. After receiving the response of the forwarded requests, each peer compares it with the local result, selecting the greatest value as the request result. If a peer receives the same request twice, it does not process the request again, but only replies with the already calculated result.

For each kind of application, we executed scenarios with two distinct matrix orders. One that is smaller enough to be processed in few milliseconds (matrix order 10) and other that was processed in seconds (matrix order 150). For each execution, every client/peer performed 100 requests. The metric that was measured was the *request processing time average* for a single request. Figure 2(a) and 2(b) show the results we obtained for client-server scenario. The x axis represents the number of clients that simultaneously accessed the single server. In the y axis, we have the request processing time mean considering all the clients and the standard deviation for that metric, represented as an error bar.



(a)

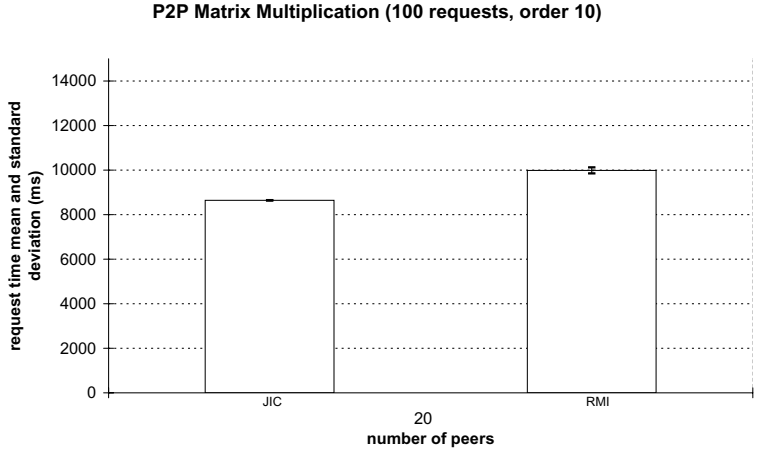


(b)

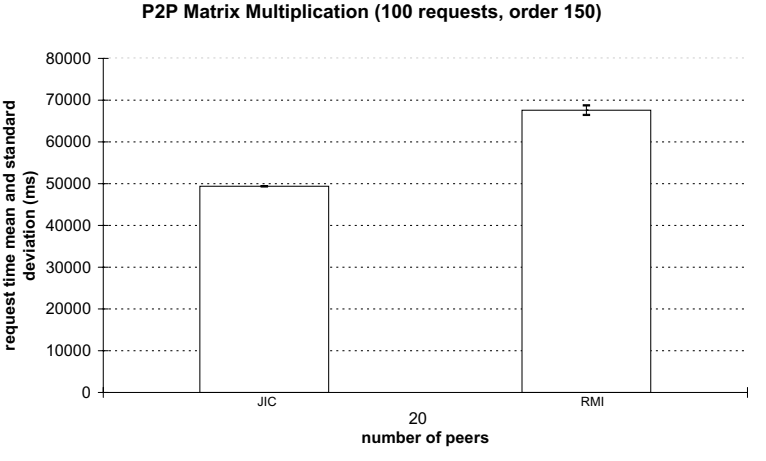
Fig. 2. (a) Client-server application for a small matrix and (b) for a big matrix

We can see that JIC performance is comparable to Java RMI. The greater times sometimes observed in JIC can be justified by the use of the Jabber server, which acts as a relay for every message between clients and the server. However, we consider that this is a good trade-off, since the use of the Jabber server makes JIC a firewall and NAT friendly solution and, besides that, we did not need to worry about concurrency in JIC, as the server access point was monothreaded. In RMI, the server has as many threads as the number of simultaneous clients, so that the server must be thread-safe.

Figure 3(a) and 3(b) shows the same measurement for the peer-to-peer scenario. This scenario was executed for a network of 20 peers. The results, however, show that JIC average request times are smaller for both matrix sizes. We believe



(a)



(b)

Fig. 3. (a) Peer-to-peer application for a small matrix and (b) for a big matrix

this happens due to the blocking nature of RMI application. In RMI, each request is locally processed, then forwarded to a random peer, and only forwarded to the second peer after the receiving of the result from the first random peer. (Likewise, the forwarding to the third random peer is only done after the second peer returns) As such, for the whole system of n peers, there are at most n^2 threads, but $n^2 - n$ are blocked waiting for a response of some other peer. JIC requests although dispatched in parallel, are processed by a single thread in each peer, but the peers do not suffer the overhead of thread management.

Although we have not finished porting OurGrid to JIC, we have already changed most of its interfaces to explicitly implement an event-driven architecture in which threads are modularized in the same way this is done by JIC.

Communication is still RMI, but after a remote method is invoked, it creates an object denoting the invocation, and places it in an invocation queue. Invocation queues are consumed by threads that never leave the "access point". All business logic is within "access points", behind the invocation queue. RMI methods only serve to relay invocations to the queue. We only retained the traditional RMI usage for the components that are commonly installed in different administrative domains, on which the need for callbacks would generate great administrative burden to open firewall ports and/or set-up reverse NATs.

Even considering that the application of the event-driven architecture did cover all code, and that much of the existing code is tedious and will disappear with JIC (i.e., the "invocation classes" and all RMI code that exists before the invocation queue), the results were very positive. We were able to release a long delayed release of OurGrid (version 2.2), which was "stuck" in a fix-a-bug-introduce-another loop. Maintenance time was drastically reduced. Programmer felt much more confident about their code. As a numerical evidence, the number of synchronized blocks in the code have decreased 36,8%, the efferent coupling [23] has decreased 4.6%, the lack of cohesion of methods (LCOM*) [23] has decreased 26.6%. We refer the reader to [16] for a detailed description of our experience of introducing the event-driven remote invocation in OurGrid.

By finishing the migration of OurGrid to JIC, we expect to further improve the code. Expressing an event-driven architecture, JIC is much more natural than RMI methods that explicitly place "invocation objects" on an event queue. A lot of OurGrid code is just going to disappear, further simplifying the software and increasing its understandability.

4 Related Work

The Common Object Request Broker Architecture (CORBA) [5] provides two main non-blocking communication models: (i) the Oneway invocation; and, (ii) the Deferred Synchronous model. In [24], Schmidt *et al.* shows that Oneway calls guarantee neither non-blocking semantics nor reliable delivery. In the Deferred Synchronous model, after making a request, the client can either poll to see if the target object has returned a response, or it can perform a separate blocking call to wait for the response. The deferred synchronous model is inefficient due to its reliance on the Dynamic Invocation Interface (DII), which allocates memory and copies data excessively [24].

To address these problems, the CORBA Messaging specification [25] introduces the Asynchronous Method Invocation (AMI) [26], which specifies two similar non-blocking communication models. In the Polling model, the invocation returns a Poller object. The client can use the Poller methods to obtain the status of the request and the value of the reply from the server (in a blocking way or not). In the Callback model, a callback entity named *ReplyHandler* is passed as a parameter when a client invokes a server and the server responses are redirected by the client ORB to the ReplyHandler. However, few ORBs fully implement this specification.

There are works that aim to built remote invocation mechanisms in Java, like RMI. In [27] and [28], the authors propose to extend the model used in RMI to provide non-blocking communication. The solution consists on the automatic creation of threads to deal with each remote call. Despite the simplicity and good integration with the language these solutions have the same drawbacks of RMI, since they use the same communication layer and do not prevent the thread explosion that can occur at the client side if many simultaneous invocations are performed.

Message-oriented middleware (MOM), such as IBM's MQSeries [29], are simple and provide low coupling between their entities. Basically, MOMs are composed by three types of entities: (i) the *event service*, that is the entity responsible for the management of the events, notifying the interested entities that some event has happened; (ii) the *interest objects*, that publish events in the event service; and, (iii) the *receptor objects*, that subscribe their interest in the occurrence of events of a given type (publish/subscribe model). Despite the well-defined API to exchange messages, MOMs are not fully (and transparently) integrated to programming languages.

MOMs architecture provide mechanisms that allow suppliers to reliably transmit messages asynchronously through the use of special entities named containers. The containers (or servers) route and process messages on behalf of application processes. It uses a store and forward strategy. It means that if a consumer happens to be unavailable due to scheduled downtime, a site crash, or a network partition, the router will attempt to deliver the message periodically until the consumer becomes available.

Other problem with the MOMs is that vendors have performed proprietary extensions to the model. Because of that, applications built using systems from different vendors may not be able to exchange messages. To circumvent this problem, the user needs to use (or built) gateways that deal with the particularities of each vendor implementation. Another drawback is that MOM's non-blocking mechanisms are too heavyweight for many high-performance and real-time applications. Moreover, the message-oriented invocation mechanisms of MOM systems can be harder to program correctly due to the lack of strong type checking.

5 Conclusions

JIC is a communication solution that supports a notion of distributed objects that can be invoked without blocking the invoking object. This approach leads to an event-driven software architecture, in which threads are scoped within components of the distributed application, called access points. Delimiting thread scope (instead of letting threads to traverse the whole distributed application) greatly simplifies the software development. In particular, scoping out threads makes for composable components.

JIC also addresses other difficulties of using distributed objects on scenarios in which clients (invokers) have something else to do besides waiting for the server

(invoked) to respond, such as thread explosion and the need to decouple failure detection from method invocation. JIC is built over Jabber, which provides native message forwarding, much easing the deployment of JIC-based applications over firewall and NATs. Finally, all of this is done without introducing a performance penalty (when compared with RMI). It is also worth noting that JIC is open source (available at www.ourgrid.org).

We are currently porting OurGrid, a free-to-join grid [15], to JIC. The initial impression of the OurGrid programmers was that developing a JIC application requires more effort than its RMI equivalent. This is due to the need of explicitly managing response events. In JIC, the programmer must separate which events are new requests from those that are responses for a request initiated by the component itself. In RMI, this is automatically done. The response is the method return value. However, after a while, the programmers realize that the initial greater effort to develop JIC code pays off. The code seems to be harder to develop in the beginning, but it is easier to get right, just as a code that is developed with automated tests. Understanding the code behavior and debugging it are much simpler. We credit most of this simplicity due to the well-defined thread scoping. However, we also believe that JIC leads to a good distributed programming discipline. By making the event handling explicit, the programmer is invited to think how the code should react to any event (including failures on remote objects), which can happen with any configuration of the component state. While seeming a lot of work, this is truly necessary in a distributed system. RMI programmers might only realize this after lots of tests and bug occurrences. In short, we believe that RMI is deceively simple. JIC just exposes the real complexity of distributed programming in a way that helps the programmer to deal with it.

One exciting possibility JIC opens up is the possibility of obtaining parallelism without writing multi-thread code. JIC access points can be single-threads. If an application has enough of them, this may be enough to explore the application parallelism. Even if this is not the case, one can start with single thread per access point, profile the application, and only invest in multi-threading the bottleneck access points. We are currently investigating these possibilities.

On the down side, JIC is language dependent. However, we do not see any obstacle to remove this limitation by using XML to code messages in order to represent method calls in an interoperable way. Some argue that this could require manual coding of the XML to achieve true interoperability [30]. But that seems the price to pay for true multi-language interoperability.

Acknowledgments

We would like to thank Ayla Dantas, Marcell Manfrim and Katia Saikoski for the important comments and suggestions to this paper. We also thank the anonymous reviewers for the insightful comments and questions. This work has been developed in collaboration with HP Brazil R&D.

References

1. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: *A Note on Distributed Computing.* Technical Report SMLI TR-94-29, Sun Microsystems Labs (1994)
2. Coulouris, G., Dollimore, J.: *Distributed systems: concepts and design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005)
3. Grosso, W.: *Java RMI.* O'Reilly & Associates, Inc., Sebastopol, CA, USA (2002)
4. RMI: *Java Remote Method Invocation web site.* At `java.sun.com/products/jdk/rmi/` (2006)
5. CORBA: *Common Object Request Broker Architecture Core Specification.* At `www.omg.org/technology/documents/formal/corba_iiop.htm` (2006)
6. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue **3** (2005) 54–62
7. Lee, E.A.: The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley (2006)
8. Schmidt, D., Gokhale, A., Harrison, T., Parulkar, G.: A high-performance endsystem architecture for real-time CORBA. IEEE Communications Magazine **4** (1997)
9. Pietzuch, P.R.: Event-based middleware: A new paradigm for wide-area distributed systems? In: 6th CaberNet Radicals Workshop. (2002)
10. Starovic, G., Cahill, V., Tangney, B.: An event based object model for distributed programming. In: OOIS (Object-Oriented Information Systems) '95, London, Springer-Verlag (1995) 72–86
11. DS-online: *Distributed Systems Online web site. Message Oriented Middleware section.* At `dsonline.computer.org/middleware/intro_MOM.html` (2006)
12. Monson-Haefel, R., Chappell, D.: Java Message Service. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2000)
13. Sunderam, V.S.: PVM: a framework for parallel distributed computing. Concurrency, Practice and Experience **2** (1990) 315–340
14. Forum, M.P.I.: MPI: A message-passing interface standard. Technical Report UT-CS-94-230 (1994)
15. Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., Mowbray, M.: Labs of the world, unite!!! (Accepted for publication in Journal of Grid Computing. Available at `walfredo.dsc.ufcg.edu.br/papers/Labs of the World Unite v19.pdf`)
16. Dantas, A., Cirne, W., Saikoski, K.: Using aop to bring a project back in shape: The ourgrid case. Journal of the Brazilian Computer Society **11** (2006) 21–35
17. JSF: Jabber software foundation web site. At `www.jabber.org/` (2006)
18. XMPP: Extensible messaging and presence protocol specification. At `www.xmpp.org/specs/` (2006)
19. Deng, X., Dwyer, M.B., Hatcliff, J., Mizuno, M.: Invariant-based specification, synthesis, and verification of synchronization in concurrent programs (2002)
20. Hayashibara, N., Defago, X., Yared, R., Katayama, T.: The phi accrual failure detector. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS04), IEEE Computer Society (2004) 66 –78
21. Felber, P., Défago, X., Guerraoui, R., Oser, P.: Failure detectors as first class objects. In: Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland (1999) 132–141
22. Tomlinson, R.S.: Selecting sequence numbers. In: Proceedings of the 1975 ACM SIGCOMM/SIGOPS workshop on Interprocess communications, New York, NY, USA, ACM Press (1975) 11–23

23. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice Hall (1995)
24. Schmidt, D., Vinoski, S.: Object interconnections: Programming asynchronous method invocations with corba messaging (1999)
25. Object-Management-Group: Corba messaging specification, omg document orbos/98- 05-05 ed. (1998)
26. Vinoski, S.: New features for corba 3.0. Commun. ACM **41** (1998) 44–52
27. Raje, R., William, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism for java. In: ACM Workshop on Java for Science and Engineering Computation, ACM Press (1997)
28. Falkner, K.E.K., Coddington, P.D., Oudshoorn, M.J.: Implementing Asynchronous Remote Method Invocation in Java. Technical Report DHPC-072 (1999)
29. MQSeries: IBM MQSeries web site. At `www.ibm.com/software/mqseries` (2006)
30. Loughran, S., Smith, E.: *Rethinking the Java SOAP Stack*. Technical Report HPL-2005-83, Hewlett-Packard Bristol Laboratories (2005)

MoCoA: Customisable Middleware for Context-Aware Mobile Applications

Aline Senart, Raymond Cunningham, Mélanie Bouroche, Neil O'Connor, Vinny Reynolds, and Vinny Cahill

Distributed Systems Group, Department of Computer Science, Trinity College Dublin, Ireland
first.last@cs.tcd.ie

Abstract. Many programming models have been proposed to facilitate the development of context-aware applications. However, previous work does not offer support for building customised systems and has largely been targeted at a single application domain. In this paper, we describe MoCoA, a flexible middleware framework that permits the rapid development of context-aware applications and supports deployment scenarios ranging from augmented artefacts to city-wide smart-space applications. Crucially, MoCoA supports a small set of programming abstractions that are suitable for building a wide range of context-aware applications for deployment in a fixed or (ad hoc) mobile environment. For each of these abstractions, MoCoA provides a set of implementations via a library of components. We present three applications of the MoCoA framework that demonstrate both the use of the programming abstractions and the flexibility of the framework.

1 Introduction

Enabled by recent and expected developments in new sensor technologies, by wireless networking, and by miniaturisation of computational devices, context-aware mobile computing has become a reality. We can now envision a class of applications involving large collections of collaborating mobile devices that operate by sensing their surrounding environment and adapting their behaviour accordingly without human intervention [1]. These applications will typically address areas such as environmental monitoring, independent living, intelligent transportation systems, mobile robotics, and city-wide smart spaces.

Even though many programming models, supported by associated middleware, have been proposed to facilitate the development of context-aware applications, to date, none offers the flexibility required by the application space. They are typically only appropriate for a single application domain or deployment scenario, e.g., smart spaces [2,3], tracking of valuable goods [4,5] or robotics [6,7]. Moreover, they fail to handle some or all of the non-functional requirements expected of emerging context-aware applications such as timeliness, reliability, mobility and scalability.

The main contribution of this paper is the identification of a small set of abstractions that can be used to support a wide range of context-aware (mobile)

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1722–1738, 2006.
© Springer-Verlag Berlin Heidelberg 2006

applications. These abstractions have been implemented in MoCoA, a flexible middleware framework for building context-aware applications for deployment in fixed or (ad hoc) mobile environments. We describe how context-aware applications can be modeled with these abstractions and describe their implementation in a library of commonly-required and optional components that permits the rapid development of applications. Examples of components include specific sensors and actuators and high-level services such as context acquisition, event-based communication, and intelligent reasoning. These components can be assembled and specialised by developers to form customised middleware that fits application requirements exactly. The use of replaceable components and well-defined interfaces improves code reuse and allows the MoCoA framework to address multiple combinations of non-functional requirements, and therefore to be suitable for a wide range of applications.

To demonstrate the use of the programming abstractions and the flexibility of MoCoA, we have designed a number of representative context-aware applications that exhibit a variety of different non-functional requirements. Among these, we describe in this paper a sensor-augmented sofa with application in independent living, an autonomous mobile robot, and a simulated urban traffic control system. These experiments in assembling and customising components from our library serve to validate our approach.

The remainder of this paper is structured as follows. In Section 2, we describe three applications used in the paper to demonstrate the potential of MoCoA and introduce the key requirements of context-aware (mobile) applications. In Section 3, we present the programming abstractions and components supported by MoCoA. Then, in Section 4, we describe and discuss the use of these abstractions and of MoCoA in the application scenarios. Finally, Section 5 and Section 6 present related work and our conclusions respectively.

2 Application Requirements

Customisable middleware supporting common programming abstractions for the envisaged generation of context-aware applications is a key requirement for supporting a range of deployment scenarios. As discussed below, such middleware need to support non-functional requirements in order to cope with large-scale distributed applications, composed of heterogeneous and mobile devices interacting closely with a changing environment. This section presents examples of representative applications exhibiting non-functional requirements and derives requirements that context-aware applications have on middleware.

2.1 Application Scenarios

We introduce a few scenarios that are representative of expected classes of sensor-driven context-aware applications and indicate their main requirements.

The Sentient Sofa. Falls present a serious health risk. Among older adults, falls are the leading cause of injury deaths and the most common cause of injuries

and hospital admissions for trauma [8]. [9] argues that "technological devices such as alarm systems that are activated when patients try to get out of bed or move unassisted may be useful".

Motivated by these observations, we have designed the "sentient sofa" as a prototype of a bed for elderly or disabled people that raises an alarm if the occupant leaves the bed, for example, as a result of a fall. This scenario is representative of the *augmented artefact* class of context-aware application, where sensor-enabled systems are embedded in everyday environments, and addresses one of the most important application domains for this technology in the future, independent living.

The sentient sofa can identify the person lying in the bed, monitor her/his movements and autonomously decide to alert carers in case of unexpected behaviour. To achieve this, the sofa is equipped with a fixed number of load sensors enabling the mass currently on it to be measured. This assisted living application is relatively small scale and doesn't require mobility support.

Sentient Traffic Lights. Traffic congestion is an increasing problem in urban areas but is alleviated to some extent by urban traffic control (UTC) systems that control and coordinate traffic signal timings in order to reduce journey times and delays [10,11]. Future UTC systems may be able to exploit a wide variety of fine-grained sensor data, e.g., GPS position data from individual vehicles, to inform attempts to optimise signal timings.

To explore these possibilities, we have designed a city-scale (simulated) UTC system in which traffic light controllers use sensor data from local vehicles to inform their decision making [12]. In this scenario, the traffic light controllers also exchange views of their environment describing the traffic conditions sensed locally through sensor data fusion. By incorporating these views into their reasoning, traffic light controllers can establish convergent views of the congestion level, which can in turn inform their local decisions as to which signal phase is currently most appropriate. Positive (or negative) feedback from the environment (e.g., a decrease or increase in the local congestion level) reinforces (or discourages) the selection of such phases and enables good strategies to be learned over time.

This scenario is representative of the class of *city-wide smart space* applications, that are typically large-scale and composed of many geographically dispersed entities. Traffic light controllers in this application are not mobile, but they need to communicate with a variable collection of anonymous mobile vehicles as well as a fixed collection of nearby junction controllers. A completely decentralised learning technique is required so that consensus can be established on the optimal phases to choose at a junction.

Sentient Robots. An autonomous mobile robot is a mechanical device that performs its tasks under real-time conditions, in a previously unknown environment, without human supervision. Potential applications include dull or dirty tasks like floor cleaning, transportation and surveillance, and tasks that are dangerous for humans, such as military missions [13] and emergency rescue [14].

To represent this class of application, we have designed simple sentient robots that navigate autonomously towards some destination, while avoiding obstacles

and obeying traffic lights they encounter en route. In order to complete their mission, the sentient robots are equipped with various sensors, as illustrated in Figure 1. The outputs of these sensors are fused together in order to determine the robots' context with respect to obstacles and traffic lights and thereby infer what are the appropriate actions to take in order to avoid collisions. In this scenario, timely communication is required in order to ensure that the robots can react in time, for example, in response to a change of the traffic light colour.

Fig. 1. The sensor-augmented robot

This scenario is typical of *robotics* applications, where mobility and safety constraints are key requirements. As robots are mobile, they spontaneously communicate with anonymous neighbours that they discover in their vicinity, e.g., traffic lights or other robots. Furthermore, strong real-time constraints on communication, decision-making and reaction have typically to be guaranteed in order to ensure safety for this type of application.

2.2 Fundamental Requirements

From the scenarios studied previously, we can establish a list of requirements that middleware for (mobile) context-aware applications should support.

- **Spontaneous interaction:** Applications will be composed of a dynamically changing population of interacting devices. Hence, unanticipated interaction between nearby devices has to be supported, enabling a device to dynamically establish connections to other devices within its current vicinity.
- **Geographical dispersion:** Unlike current embedded systems, future context-aware applications will integrate components that are scattered over buildings, countries, and continents.
- **Mobility:** Parts of an application will possess the ability to move geographically (by virtue of being carried by a mobile device) or between hosts possibly on different networks, while remaining in continuous operation.
- **Large-scale:** Applications may be composed of millions of interacting hardware and software components and will be able to expand indefinitely.
- **Autonomy:** Typical context-aware applications will operate independently of human control and will be able of acting in a decentralised fashion according to their own knowledge.

- **Inference and learning:** Applications will typically have to cope with changing conditions during their lifetime. Not only must they be designed to reason about these changes according to their factual knowledge, but they must also gain knowledge through experience, e.g., by learning appropriate behaviour through trial-and-error interactions with their environment.
- **Time criticality:** These applications will interact with the physical environment within a given time interval, and will have to cope with its pace, regardless of adverse conditions due to scale and technology shortcomings.
- **Safety criticality:** As such applications are based on mobile devices embedded in everyday environments, they must obey strong safety requirements to interact with human users, whose well-being will frequently rely on them.

3 MoCoA: Middleware for Context-Aware Applications

In this section, we describe the programming abstractions that are supported by MoCoA for the development of context-aware mobile applications. For each of the abstractions, we propose optional components that are available in MoCoA's library and that can be composed and customised to build the applications.

3.1 Sentient Objects, Sensors and Actuators

In the MoCoA framework, applications are structured using the ***sentient object*** abstraction [15,16]. Sentient objects are mobile intelligent entities, that extract, interpret and use context information obtained from sensors, other sentient objects, and their computational infrastructure, to drive their behaviour. The granularity of a sentient object is not constrained: a robot, a component of a robot, or a traffic light controller are all potential examples of sentient objects.

In this model, a ***sensor*** is seen as an entity that produces software events in reaction to a real-world stimulus and is an abstraction of some physical device. To act upon their environment, sentient objects use ***actuators.*** An actuator is any component that consumes software events and reacts by attempting to change the state of the real-world via some physical device. To facilitate mobility as well as loose coupling between dynamically varying collections of anonymous mobile devices, a sentient object is both a producer and a consumer of asynchronous events. As explained above and illustrated in Figure 2, a sentient object can also receive events from its infrastructure. Notice also that sentient objects may communicate directly by means of events or indirectly via the environment.

The behaviour of all sentient objects follows the same pattern. First, a sentient object may receive input from a variety of sources (sensors, sentient objects and infrastructure) that needs to be integrated before being used in determining the overall context of the sentient object. As an example, our robots are equipped with ultrasonic, orientation, and location sensors. The outputs of these sensors are fused together with input from nearby traffic lights in order to determine the robots' context with respect to obstacles, traffic lights, and their destination.

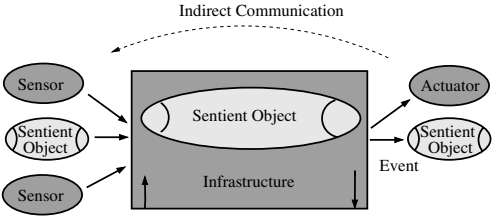


Fig. 2. An example of a sentient object

Context recognition determines the current context, based on fused data and past history. For example, that someone has left their bed is inferred from historical information that someone was in the bed and new sensor input indicating that they are no longer present.

Sentient objects are then expected to change their behaviour or act upon their environment based on some rules. This implies some form of intelligent reasoning captured in an inference component. Rules may be predefined or learned over time. In the robot scenario, the inference component reasons on a set of predefined rules, in order to take actions and/or to derive a higher-level context. For example, the detection of an obstacle results in a transition to the obstacle avoidance context, followed by a braking process. In the UTC scenario, rules are used to choose the appropriate signal phase given the current congestion level of the system and are learned based on feedback from the environment.

3.2 Events, Event Channels, and Proximities

While the basic concept of event-based communication is simple, there are a number of difficult issues to be tackled in large-scale and geographically dispersed applications in which large volumes of events are raised. In MoCoA, we support two abstractions to control event propagation: proximities and event channels.

Filtering ensure that events are only propagated to consumers that have expressed an interest in them [17]. Filters have typically been applied to the subject or the content of an event. In addition to these filters, MoCoA also supports proximity-based filtering which is used to define geographical areas, ***proximities***, within which events are valid. Such proximity-based filtering confines propagation of events to a geographical area surrounding the producer since locality considerations suggest that the closer event consumers are located to a producer the more likely they are to be interested in the events that it produces.

Proximities not only serve as a unit of scoping large-scale systems into areas of localised interaction, but also provide the basis for time-bounded event delivery, even in wireless networks. ***Event channels*** are defined to specify constraints on propagation and delivery of asynchronous events within a proximity. An event channel is specific to an event type and allows a producer to define the real-time guarantees that have to be maintained within a given geographical area.

In the robot scenario, only the local geographical environment is required to be sensed, all other information outside the vicinity of the individual robots is

not relevant and can be filtered out. Similarly, traffic lights disseminate their status to a nearby proximity over an event channel providing real-time delivery.

To support event-based communication with event channels and proximities, the MoCoA framework uses different instantiations of the STEAM event service [18]: STEAM for use in wireless ad hoc networks, and RT-STEAM for soft and hard real-time in ad hoc environments. In STEAM, proximities can be circular or hull-shaped. The set of filters available depends on the type of event parameters. For example, if the parameter is a location, the following filters can be used: `distanceIncreases`, `distanceDecreases`, `withinRange`, and `beyondRange`. Parameter types include location, time, integer, double and string.

Remote and Local Event Channels. Given that sentient objects may communicate with collocated sensors and actuators, wireless communication is not always necessary. For instance, the ultrasound sensors mounted on the robots are a priori known and their location is fixed. In this case, when communication with identified sensors and actuators that are not discovered dynamically is required, STEAM uses local event channels. In contrast, communication via the network is required to support spontaneous interactions with anonymous producers of events that are discovered dynamically, e.g., remote event channels are used between robots and traffic lights encountered en route.

Event Channels with Timeliness Constraints. To interact in the real-world, sentient objects may require temporal guarantees. To provide them, even where changes in the quality of communication are frequent, the proximities in which events are to be delivered are adapted to reflect changes in the underlying infrastructure under a model known as space-elastic adaptation [19].

In essence, when there is a change, like a disconnection or a significant decrease in message-delivery latency, that would prevent the temporal properties required by an event channel being achieved, the underlying infrastructure notifies the event producer by means of an adaptation event (an example of an event raised by the infrastructure) describing a revised proximity in which the required guarantees can be met. The producer is then free to dynamically adapt its behaviour in order to respect the safety and timeliness requirements of the application. Even in an ad hoc network subject to partitions, this model provides well-defined guarantees to applications allowing stringent safety constraints to be enforced. For example, even if communication between one of our sentient robots and the traffic lights in its path is not possible, it is still possible to guarantee that no robot will ever pass through a red light. In this case, the traffic light adapts its behaviour when it realises that messages are not being delivered in a large enough proximity for the robots to stop safely [20].

In MoCoA, the RT-STEAM component is used to support the definition of hard real-time event channels and space-elastic adaptation. RT-STEAM is implemented using RTAI [21] and relies on further components providing real-time resource reservation and routing in ad hoc networks, as well as time-bounded media access control [22,23].

3.3 Readings and Facts

Sensor fusion is used to manage the uncertainty of data captured from the real-world and to derive higher-level context information. In MoCoA, the fusion process relies on a set of pipelines, each pipeline being composed of a combination of generic and sharable components (c.f. Figure 3). Input events in a pipeline are processed through different stages.

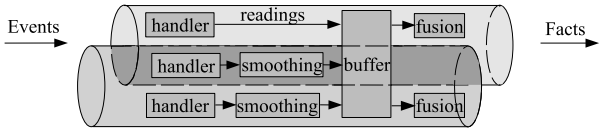


Fig. 3. Example of two pipelines

First, a pipeline extracts ***readings*** (i.e., raw values produced by a sensor, a sentient object, or the local infrastructure) from events. Then, a sequence of transformations is applied to the readings by the components present in the pipeline. The final result is a piece of higher-level information, in the form of one or more ***observed facts***.

Table 1. Types of pipeline components

Handler	Extracts readings from events and propagates them in the pipeline.
Smoothing	Preprocesses readings before fusion.
Buffer	Stores readings delivered at different times, allowing synchronised fusion or smoothing.
Fusion	Derives high-level observed facts from readings.

A pipeline may be composed of different components: handler, smoothing, buffer, and fusion (see Table 1). MoCoA already provides a set of implementations of these components, e.g., fusion can perform a sum, an average function or might rely on a Bayesian network. However, we expect developers to implement their own (targeted to real-time usage or not), extending the existing library.

3.4 Contexts

Context is defined as any information currently available in the environment that can be used to characterise the situation of an entity [24], such as its current location, the presence of other sentient objects in its vicinity or the state of its underlying infrastructure. In MoCoA, the contexts in which a sentient object can be during its lifetime are organised as a context graph, where only a subset of contexts can be transitioned to from the current context. Not only are contexts useful to structure complex applications, contexts are also the basic abstraction for Context-Based Reasoning [25] within a sentient object allowing the set of event

channels, pipelines, facts, rules and actions that are relevant at any time to be filtered. When in a context, only one pipeline is active thereby constraining the set of event channels being used. Because the number of pipelines is restricted, only some facts may be asserted in this context. Subsequently, as depicted in Figure 4, only a subset of rules needs to be evaluated and the permitted actions are limited.

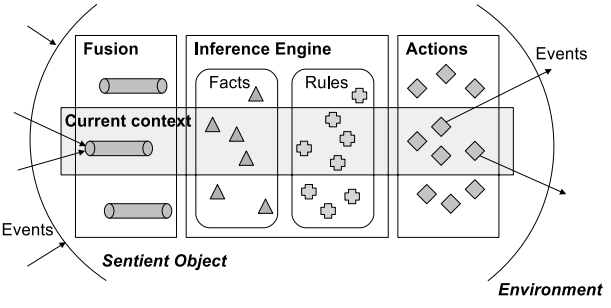


Fig. 4. The MoCoA architecture

3.5 Rules and Actions

The knowledge of a sentient object is structured into facts. To ultimately determine the appropriate behaviour of a sentient object in response to its environment, an inference engine is used to infer knowledge (i.e., to assert ***derived facts***) from previously asserted facts and to select the actions to be taken. Table 2 summarises the different types of ***action*** that an object is able to perform. The action selection decision within a sentient object is based on ***rules.*** Rules may be predefined as applicable within specific contexts, or rules defining what actions to take in specific contexts may be learned.

Table 2. Types of actions

Fact assertion	Action that asserts a fact.
Fact retraction	Action that retracts a fact.
Event production	Action that produces an event.
Execution of code	Action that carries out a portion of code.

In MoCoA, a first order logic inference engine reasons about a set of facts and predefined rules in order to derive intelligent behaviour. The rules take the form of `condition/actions`. Conditions are expressed in terms of asserted facts and can include operators, e.g., `&&`, `||`, $\exists$ and $\forall$. The actions associated with a rule are executed when the condition evaluates to true. The inference engine that we have implemented uses forward-chaining. Rules are only evaluated when a fact relevant to their condition is asserted or retracted, i.e., by relevant changes to the set of facts. For example, the transition rule `(onCourse && !atWaypoint)` in the robot scenario is only evaluated if the facts `onCourse` or `atWaypoint` are changed in the knowledge base.

Since the set of actions available is typically fixed (e.g., constrained by the available actuators) rules may be learned by attempting actions in different situations. Depending on the feedback returned from the environment, this process attempts to learn the long term reward of choosing these actions again given the current context.

Collaborative reinforcement learning (CRL), is a technique for building decentralised coordination models, that extends reinforcement learning [26] with feedback models to update an agent's (in this case, a sentient object's) policy and enables a consensus to emerge between individual policies [27]. Each sentient object uses its own policy to decide probabilistically on which action to take to attempt to achieve its goal. In dynamic distributed systems, sentient objects may also have a changing number of neighbouring sentient objects that can be used to help achieve the goal. For example, when a sentient object either cannot achieve the goal locally or when the estimated cost of achieving it locally is higher than the estimated cost of a neighbour solving it, an event is sent to its neighbour to delegate the action. Furthermore, each sentient object maintains a local view of its neighbours by exchange of advertisements.

3.6 Composition

Through an high-level API in Java, software architects can design sentient objects using the basic abstractions. To describe the services provided and required by a component, as well as the dependencies between components, we are defining an architecture description language using XML descriptors. As each component of the library exhibits its dependencies with a descriptor, MoCoA can select the appropriate components according to application requirements, customises them, and finally generates the executable code of the sentient object.

The code generation currently provides C++, but as it is decoupled from the object design process, different languages may be easily incorporated. Additionally, an XML descriptor of the sentient object, reusable in other applications, may be generated. To further assist the construction of sentient objects, we are currently extending MoCoA with a graphical interface.

4 Modeling the Application Scenarios

We have designed some scenarios representative of different classes of context-aware applications. As they expose different requirements, different components from the MoCoA library are employed. In this section, we describe how the abstractions outlined above are used to model these diverse applications and the components necessary to support each application.

4.1 The Sentient Sofa

Sentient Objects, Sensors and Actuators. To monitor movements on the sofa, four load cell sensors fitted to the legs are used. Another sensor is employed

to register new users with a keyboard. For this application, two sentient objects have been defined. As they are collocated and identified, local communication is used to support their interactions within a proximity that corresponds to the sofa. While the `sentient sofa` object produces events for the current mass on the sofa and its position, a `recogniser` sentient object uses this information to identify the user or to register him/her when necessary. Finally, to alert carers, an actuator producing an audio stream via a set of speakers is used.

Events, Event Channels and Proximities. Sensor fusion consists of summing readings extracted from the events of the four load sensors in order to determine the total weight of the person (each reading corresponding to the weight applied to one of the legs of the sofa). The location is then determined by comparing each sensor reading with the average of the four readings: a person is said to be in one quarter if the load measured by the corresponding sensor is greater or equal to the average of the loads read on all sensors. Consequently, sum and average are used for the fusion of sensor information in this scenario.

Contexts. The context graph of the `sentient sofa` is small, one context is identified as `Empty` when there is no one on the sofa and a composite context is `InUse` when there is someone on it. Fifteen distinct sub contexts of `InUse` can be transitioned to depending on the location and movements of the patient.

Rules and Actions. The behaviour of the `sentient sofa` object is rule-based and consists of the publication of events of type `massChanged` when its context changes for consumption by the `recogniser` sentient object, that in turn raises an alarm in case of unexpected behaviour, or registers a new person if the new mass does not correspond to the weight of a previous user. An example transition rule is illustrated below, where `mass` and `average` are observed facts asserted by the fusion components. This rule states that if the load cell sensors on the bottom right and bottom left are measuring a mass greater than the average mass on the sofa, whilst load cell sensors on the top are measuring a mass less than the average, then a transition to the context `bottom` is triggered.

```
(mass.id == bottomleft && mass.value > average)
&& (mass.id == bottomright && mass.value > average)
&& (mass.id == topleft && mass.value < average)
&& (mass.id == topright && mass.value < average)/
transition(bottom)
```

The evaluation of this rule followed by the context switch takes approximately 91μs. Since the sofa and its sensors/actuators occupy a fixed location, components supporting mobility are not included in the application.

The sentient sofa runs on Debian GNU/Linux and stores user profiles in a relational database. It was first developed with a CORBA interface, including in total 1789 lines of code. The new implementation of the sofa generated with MoCoA represents for the two sentient objects around 900 lines of code and requires only 287 additional lines for the database access. Using MoCoA, even

for a small sensor-augmented artefact, reduces the need for complex and error-prone programming.

4.2 Sentient Traffic Lights

Sentient Objects, Sensors and Actuators. In this large-scale and highly decentralised application, which aims to minimise the average waiting time of all vehicles in the network, we have modeled traffic light controllers as sentient objects. Each controller manage a 4-way crossroad and obtain information about its local environment from simulated location sensors attached to nearby vehicles. Simulated actuators are responsible to change the light phases.

Events, Event Channels and Proximities. A custom sensor fusion component is required to provide per-approach vehicle counts. To enable convergence towards a shared view of the congestion in the surrounding area, traffic light controllers have to share their local view of the congestion at their junction with neighbouring controllers. Therefore, traffic light controllers advertise via events their view of the level of congestion associated with a particular approach from a neighbouring junction. As there is no real-time requirement, STEAM is used between traffic light controllers.

Contexts. As a 4-way crossroad comprises 17 possible phases, sentient traffic light controllers can take 17 possible actions: stay in the current phase or switch to another phase. Each sentient object as part of its context has a representation of the congestion on each approach (i.e., high, medium or low). Sentient traffic light controllers are then composed of a $17 \times 3^4 = 1377$ contexts corresponding to different signal phases and levels of congestion on the approaches to the junction.

Rules and Actions. Instead of using predefined rules, we use the CRL inference engine component to learn the appropriate action to take within a particular context. With the incorporation of neighbour views into the knowledge base, the engine chooses a particular action to take, such as changing from one phase to another at a particular junction. This is achieved by sending an event to a simulated actuator responsible for changing the phases at this junction. Subsequently, a reward from the environment for taking this action is received, e.g., change in the velocities of vehicles on the various approaches to the junction.

Running the simulator described in [28] on Debian GNU/Linux 3.1 and using a map of Dublin with relevant junction information, we simulated a flow of vehicles in the city. Our initial results show that an individual sentient object at a junction can learn the different vehicle flow patterns across the junction and that the set of sentient objects in the system can optimise their collective behaviour.

4.3 Sentient Robots

Sentient Objects, Sensors and Actuators. In this scenario, traffic lights and cars that navigate towards some destination are both implemented as sentient objects. In order to complete their mission, the vehicles are equipped with various

sensors. Location and orientation information is received from a GPS receiver and digital compass respectively. Six ultrasonic sensors are also mounted on the vehicles to allow obstacle detection. Furthermore, traffic lights periodically produce events to inform approaching cars of their states. Once the cars have infered the actions to take in their current context, they send commands to the actuator, a software component that controls the gear and the wheels.

Events, Event Channels and Proximities. As robots are mobile, they spontaneously communicate with anonymous traffic lights that they discover in their vicinity. To ensure that the robots can react in time to a message, e.g., by braking if the light is red, the traffic light must have timely communication within a large enough coverage area. Traffic lights are guaranteed to receive adaptation events from their infrastructure in time when there is a significant change in the size of their proximity. As a consequence, they will always have time to adopt a fail-safe behaviour if communication with nearby robots is not possible, i.e., turn to green when they can not communicate with cars. This way, no car will ever pass through the traffic light when it is red [20]. Real-time wireless communication between remote and anonymous traffic lights and robots is provided by RT-STEAM. However, local event channels are used between the sensors mounted on a car and the car itself. Finally, as robots are mobile and only have interest in their local environment, proximity-based filtering filters out irrelevant events occurring in the environment.

Readings and Facts. The autonomous vehicle uses four pipelines - `Command`, `ObstacleDetection`, `AcquireWayPoint` and `ObeyTrafficLight` - to produce observed facts. In the following, we only look in detail at the `ObstacleDetection` pipeline. The `ObstacleDetection` pipeline uses readings from the ultrasonic sensors to assert facts concerning obstacles near the vehicle. To fuse information and handle uncertainty, the Bayesian network defined in the MoCoA's library is used and, as a result, the observed fact `obstacleLeft/Right/StraightAhead` is asserted when an obstacle is detected with some associated probability. On average, a fact is asserted 100μs after a sensor reading is received by the Bayesian network. Training the network beforehand is necessary to obtain the probabilities of these hypotheses given available evidence from sensors.

Contexts. The context hierarchy for the autonomous vehicle is too complex to examine in detail here (see Figure 5). Instead, we will look at the `AvoidObstacle` major context and some of the rules associated with that context. Notice that several contexts in the context graph can be active at the same time: the robots may be trying to avoid an obstacle while still obeying a relevant traffic light.

Rules and Actions. To drive the behaviour of the autonomous robots, rules have been defined using the rule-based inference engine. For instance, the `Avoid-Obstacle` context is transitioned to when an obstacle is detected, i.e., each context which can transition to this context has a rule of the form:

```
obstacleRight||obstacleLeft||obstacleStraightAhead/
transition(AvoidObstacle).
```

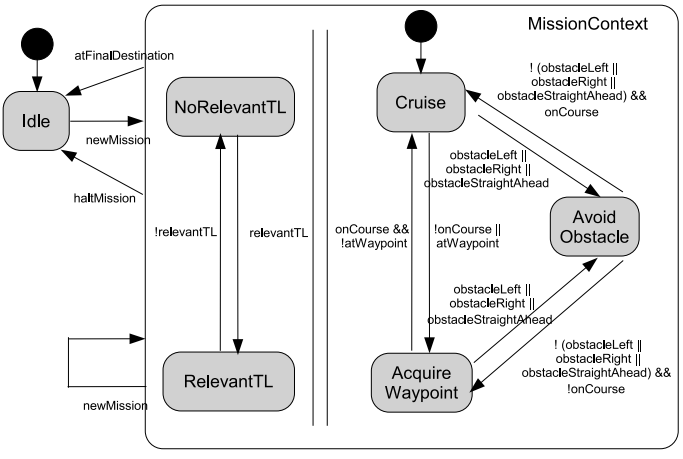


Fig. 5. Context graph of the sentient robots

In turn, the new context changes the behaviour of robots by changing the state of their actuator, e.g., `raise(leftCommand)`.

The rule below is an example of a behavioural rule defined in the `RelevantTL` context. Cars might have to stop if a traffic light is on their way and will be red by the time they arrive at it. The decision to start braking is made when the car reaches the braking distance from the traffic light. This rule expresses a condition on derived (e.g., `brakingDistance`) and observed (e.g., `trafficLight`) facts. `distanceTo` is a function that compares the current position of a robot with the position of a traffic light. If the condition is true, a stopCommand event is sent to the actuator that will apply the brakes.

```
distanceTo(trafficLight.position) == brakingDistance
&& trafficLight.colour == red/
raise(stopCommand)
```

This scenario represents 588 lines of high-level code and the complete application has been entirely generated by the MoCoA framework in 538ms.

4.4 Discussion

As space in this paper is limited, we cannot discuss in detail the different scenarios and the implementation of the components used, instead we prefer to highlight the flexibility and generality of our framework. Our experience and the above applications show that providing high-level abstractions for sensor-oriented computing allows to develop easily complex applications ranging from augmented artefacts to autonomous mobile robots to city-wide smart space applications. The developer needs only to identify facts, contexts and possibly rules, then MoCoA automatically selects the appropriate components and generates (most of the time) the complete code for the specified application.

5 Related Work

There have been numerous efforts at providing middleware to ease the development of context-aware applications. However, as discussed below, context-aware frameworks are often limited to an application domain and fail to provide a generic solution for all types of context-aware applications.

Dey et al. [29] provide a toolkit which enables the integration of context data into applications, but do not provide mechanisms for systematically performing sensor fusion or reasoning about context in a mobile ad hoc environment, as our framework does. Other work provides mechanisms for reasoning about context [24] but still does not provide a well-defined programming model and does not address the challenges of mobility.

More recently, Solar [30] has been developed. This software infrastructure supports context collection, aggregation and dissemination, and provides a small composition language, allowing applications to construct a graph of operators in order to compute desired context from appropriate sources. Our middleware provides a more generic way of specifying the behaviour of context-aware applications using different reasoning and learning mechanisms and is not limited to infrastructure-based applications. Gaia [31] provides different components to support sensor fusion and reasoning about context. However, it fails to address the challenges of mobility and is primarily targeted at smart space applications.

Cooperative artefacts [32] are also restrained to one class of context-aware applications and additionally do not provide much support when reasoning about contexts. Context-based reasoning is not used to filter the relevant facts/rules/actions. Similarly, intelligent agents [33] select an action that is expected to bring them closer to a goal, given their built-in knowledge and the evidences that they perceive from the environment. Although they present powerful mechanisms to deal with uncertain knowledge, reason and learn, the agents do not exhibit a context-driven behaviour. As such, decision-making is less efficient and therefore seems suitable only for small-scale applications.

Another class of middleware for mobile context-aware applications based on tuple spaces exists. LIME [34] enables mobile coordination by abstracting communication into a tuple space that contains the data available on all connected devices. EgoSpaces [35], evolved from LIME, defines views that do not require distributed transactions to be maintained. The goals of these systems are in line with our objectives but their solutions do not allow decentralised optimisation through collaboration and are not suitable for applications with timeliness requirements.

6 Conclusion

The MoCoA middleware framework provides a systematic approach to the development of a wide range of context-aware applications in fixed or mobile ad hoc environments. Through its common programming abstractions, MoCoA describes a family of possible middleware platform addressing multiple combinations of non-functional requirements. The middleware framework builds on sentient objects,

which support sensor fusion, context extraction, reasoning and event-based communication over fixed or wireless ad hoc networks with the possibility to include real-time support. Multiple off-the-shelf components offering implementations of these abstractions are provided by the MoCoA library. In order to illustrate the generality of our approach, we have successfully instantiated MoCoA for three different types of sensor-based applications, which involve context-aware entities with different requirements in terms of safety and mobility. We are currently investigating the behaviour of context-aware application developers using MoCoA.

References

1. Addlesee, M., Curwen, R., Hodges, S., Newman, J., Steggles, P., Ward, A., Hopper, A.: Implementing a sentient computing system. IEEE Computer Magazine **34**(8) (2001)
2. Dearle, A., Kirby, G.N.C., Morrison, R., McCarthy, A., Mullen, K., Yang, Y., Connor, R.C.H., Welen, P., Wilson, A.: Architectural support for global smart spaces. In: 4th International Conference on Mobile Data Management. (2003)
3. Tapia, E.M., Intille, S.S., Larson, K.: Activity recognition in the home using simple and ubiquitous sensors. In: International Conference on Pervasive Computing. (2004)
4. Decker, C., Beigl, M., Krohn, A., Robinson, P., Kubach, U.: eSeal - a system for enhanced electronic assertion of authenticity and integrity. In: International Conference on Pervasive Computing. (2004)
5. De, P., Basu, K., Das, S.K.: An ubiquitous architectural framework and protocol for object tracking using rfid tags. In: International Conference on Mobile and Ubiquitous Systems: Networking and Services. (2004)
6. LaMarca, A., Koizumi, D., Lease, M., Sigurdsson, S., Borriello, G., Brunette, W., Sikorski, K., Fox, D.: Making sensor networks practical with robots. Technical Report IRS-TR-02-004, Intel Research and University of Washington (2004)
7. Sakamoto, D., Kanda, T., Ono, T., Kamashima, M., Imai, M., Ishiguro, H.: Cooperative embodied communication emerged by interactive humanoid robots. International Journal of Human-Computer Studies **62**(2) (2005)
8. for Injury Prevention, C.N.C., Control: Injury Fact Book (2001-2002)
9. Rubenstein, L.Z., Josephson, K.R., Robbins, A.S.: Falls in the nursing home. Annals of Internal Medicine **121**(442) (1994)
10. Sims, A.: The sydney coordinated adaptive traffic system. In: ASCE Engineering Foundations Conference on Research Priorities in Computer Control of Urban Traffic Systems. (1979)
11. Hunt, P., Robertson, R., Winton, R., Bretherton, R.: Scoot - a traffic responsive method of coordinating signals. Technical Report TRRL Report 1014, Road Research Laboratory (1981)
12. Cunningham, R., Dowling, J., Harrington, A., Reynolds, V., Meier, R., Cahill, V.: Self-optimisation in a next generation urban traffic control environment. ERCIM News: Special Edition on Emergent Computing (2006)
13. International Federation of Robotics. http://www.ifr.org/ (2006)
14. Hirose, S., Fukushima, E.F.: Development of mobile robots for rescue operations. Advanced Robotics **16**(6) (2002)
15. Verssimo, P., Cahill, V., Casimiro, A., Cheverst, K., Friday, A., Kaiser, J.: Cortex: Towards supporting autonomous and cooperating sentient entities. In: European Wireless. (2002)

16. Biegel, G., Cahill, V.: A framework for developing mobile, context-aware applications. In: 2nd IEEE Conference on Pervasive Computing and Communications, Percom 2004. (2004)
17. Mühl, G., Fiege, L., Buchmann, A.: Filter similarities in content-based publish/subscribe systems. In: International Conference on Architecture of Computing Systems. (2002)
18. Meier, R., Cahill, V.: Exploiting proximity in event-based middleware for collaborative mobile applications. In: 4th IFIP International Conference on Distributed Applications and Interoperable Systems, Springer-Verlag (2003)
19. Hughes, B., Meier, R., Cunningham, R., Cahill, V.: Towards real-time middleware for vehicular ad hoc networks. In: 1st International Workshop on Vehicular Ad Hoc Networks. (2004)
20. Bouroche, M., Hughes, B., Cahill, V.: Building reliable mobile applications with space-elastic adaptation. In: International Workshop on Mobile Distributed Computing. (2006)
21. : Real Time Application Interface for Linux (2006) https://www.rtai.org/.
22. Hughes, B., Cahill, V.: Achieving real-time guarantees in mobile wireless ad hoc networks. In: Real-Time Systems Symposium. (2003)
23. Cunningham, R., Cahill, V.: Time bounded medium access control for ad hoc networks. In: Workshop on Principles of Mobile Computing. (2002)
24. Schmidt, A., Aidoo, K.A., Takaluoma, A., Tuomela, U., Laerhoven, K.V., de Velde., W.V.: Advanced interaction in context. Handheld and Ubiquitous Computing, Springer-Verlag (1999)
25. Gonzalez, A.J., Ahlers, R.: Context-based representation of intelligent behavior in training simulations. Transactions of the Society for Computer Simulation International **15**(4) (1999)
26. Sutton, R., Barto, A.: Reinforcement Learning. MIT Press (1998)
27. Dowling, J., Curran, E., Cunningham, R., Cahill, V.: Using feedback in collaborative reinforcement learning to adaptively optimise manet routing. IEEE Transactions on Systems, Man and Cybernetics **35**(3) (2005)
28. Reynolds, V., Cahill, V., Senart, A.: Requirements for an ubiquitous computing simulation and emulation environment. In: First International Conference on Integrated Internet Ad hoc and Sensor Networks. (2006)
29. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction **16**(2-4) (2001)
30. Chen, G., Kotz, D.: Solar: An open platform for context-aware mobile applications. In: International Conference on Pervasive Computing. (2002)
31. Roman, M., Campbell, R.H.: A middleware-based application framework for active space applications. In: International Middleware Conference. (2003)
32. Strohbach, M., Gellersen, H., Kortuem, G., Kray, C.: Cooperative artefacts: Assessing real world situations with embedded technology. In: International Conference on Ubiquitous Computing. (2004)
33. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice Hall (2003)
34. Murphy, A.L., Roman, G.C.: Lime: A coordination middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology (To appear)
35. Julien, C., Roman, G.C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering **32**(5) (2006)

A Framework for Adaptive Mobile Objects in Heterogeneous Environments

Rüdiger Kapitza[1], Holger Schmidt[2], Guido Söldner[1], and Franz J. Hauck[2]

[1] Dept. of Comp. Sciences, Informatik 4, University of Erlangen-Nürnberg, Germany
{rrkapitz, sigusoel}@cs.fau.de
[2] Distributed Systems Laboratory, University of Ulm, Germany
{holger.schmidt, franz.hauck}@uni-ulm.de

Abstract. The majority of object migration systems do not support heterogeneous environments. Few systems solve this challenge by specifying a platform and language independent state transfer format, requiring a compatible implementation for every target language. However, fields of research like Ubiquitous and Pervasive Computing with mobile users and applications demand an even more platform-independent, flexible and adaptive approach.

This paper presents a novel approach for adaptive object and agent migration in heterogeneous environments based on our former work enabling language- and platform-independent object mobility in CORBA. By providing flexible mechanisms to reduce, expand and transform an object's state and functionality during migration, we support adaptation to the context and application-specific demands at the target system.

This is achieved by introducing a separation of state, functionality and implementation code instead of mapping particular state on particular code. Our prototype system supports object migration from Java to C++ and vice versa. In principle, our concept can be transferred to any CORBA-supported programming language.

Keywords: Object Migration, Object Adaptation, Platform Independency, CORBA, Life Cycle Service, Value Types, Dynamic Loading of Code.

1 Introduction

Nowadays, computers surround us almost everywhere. This trend is even pushed by the idea of Ubiquitous Computing where computing is integrated into the environment instead of using particular devices. For fulfilling this task, these devices are linked with one another, which should enable an autonomous reaction to particular situations. Accompanied by this evolution and the rising diversity of systems, new concepts and techniques for providing adaptive and context-aware applications are required. Often, these applications should migrate between different platforms during their lifetime. As a typical example, a "follow me" application [1] (e.g. personal information manager application) can have a different interface and state on a desktop computer, a PDA (might

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1739–1756, 2006.
© Springer-Verlag Berlin Heidelberg 2006

also be a smartphone) or a public web terminal. In the first case, the full set of features is provided, whereas in case of the PDA some data-intensive parts are left out. The PDA might even abandon communication-intensive parts, as communication is expensive and restricted in some areas, respectively. Finally, in the web terminal's case, one would only provide the essential parts of the state to fulfil the demanded task for privacy reasons. In all cases, different hardware and software systems can be expected. In other words, we assume that a mobile application has to adapt its state, the provided functionality and the implementation basis to its execution context, the target system and application-dependent restrictions.

Furthermore, there is a need for frameworks supporting complex mobile processes consisting of long-lasting activities. These activities are spread over the network altogether fulfilling the mobile process's task. E.g., Kunze et al. [2] propose a process description language and an execution model for such tasks. However they do not address context-aware and platform-dependent adaptation, which is often required in heterogeneous environments.

Recent object-based middleware and agent platforms restrict migration support to a particular programming language and environment. Only a few systems provide support for migration in heterogeneous environments such as AgentFactory [3] and ATS [4]. These rely on a custom language-independent serialisation format for state transfer and require an implementation providing the same functionality in every target language. A custom state format results in high implementation costs in terms of porting the system to a new platform and providing a fully-fledged implementation of a mobile object for each supported target language. This is especially the case if only parts of the functionality should be provided or are required on certain systems. In scenarios similar to the "follow me" example, which require the ability to dynamically adapt the provided functionality and state according to the current context, a more platform-independent, flexible and adaptive approach for object migration is required.

In this paper, we propose the concept of adaptive mobile objects. These are capable of adapting their state, functionality and underlying code basis during migration to the requirements of the target platform and the needs of the object itself. We focus on weak migration, which means that only the state of an object is migrated but no execution-dependent state, e.g., values on the stack and CPU registers. Our solution is based on CORBA value types [5], a standard CORBA mechanism for passing objects by value, and a dynamic adaptation service for mobile objects, the adaptive object migration service (AOM). Additionally, we provide support for mobile objects acting as mobile agents (an object having an own thread that executes autonomously on behalf of a user). By building on our recent platform- and ORB-independent implementation of the CORBA Life Cycle Service [6] we provide support for heterogeneous environments. In fact, our current prototype supports the migration of objects between Java and C++. Supporting other CORBA languages requires only moderate implementation effort.

The paper is structured as follows: In Section 2, we give an overview of our recent realisation of the CORBA Life Cycle Service. Based on this, in Section 3, we present our concept of adaptive object migration for mobile objects. Then, the development process of an adaptive mobile object is shown by an example application in Section 5. In Section 6, we show related work, and finally, we conclude and discuss future work in Section 7.

2 CORBA Life Cycle Service

CORBA is a standardised architecture defined by the Object Management Group (OMG). This architecture enables programmers to create and access objects deployed in a distributed system, and provides platform and language transparency. Common middleware tasks like object location, request marshalling and message transmission are performed by the Object Request Broker (ORB). A server object is specified by describing the object's interface using the Interface Definition Language (IDL). This interface is used to generate platform-specific stubs for the client and skeletons for the server side. Both entities act as surrogates dealing with heterogeneity and handle remote invocations and marshalling for the object. Objects are actually implemented by servants that are registered at an object adapter. For invoking remote methods, a client only needs a valid object reference that can be bound by the local ORB instance.

CORBA specifies a set of CORBA services. These services represent optional ORB extensions and address general needs of CORBA applications. The Life Cycle Service [7] is such a CORBA service, as it enables application-controlled object management including mobility and other life-cycle operations.

The CORBA Life Cycle Service describes the interfaces of all required components for enabling object mobility in detail. However, the specification leaves open important issues. For migrating a mobile object, the state and the code have to be transferred (this implies determining the state of the mobile object dynamically at runtime). In principle, the specification describes these processes as an object-specific task that should be handled by the object developer. This has various disadvantages as state transfer methods have to be implemented from scratch for each object class. This is error-prone and leads to serious interoperability problems in heterogeneous environments.

However, we identified the fact, that migration in heterogeneous environments enforces differentiating between implementation-dependent and -independent state. A `java.util.Hashtable` object from the Java class library contains many implementation-dependent state variables, e.g., a set of diverse constants regarding the used hash-function. In heterogeneous systems, it does not make sense to transfer state that is highly specific for a certain implementation, as other implementations are not able to interpret these values. This is especially the case if an object is migrated between different programming languages (enabled by our implementation of the CORBA Life Cycle Service). In such systems, just the implementation-independent state should be transferred; in case of a `Hashtable`-Object, this state is represented by the actual <*key,value*>-pairs. To sum it up we consider

- *Implementation-dependent state* as values that are specific for a certain implementation variant of an interface, and
- *Implementation-independent state* as values that are needed by any possible implementation to provide the functionality defined by an interface and values that would be considered as information loss if they were omitted.

Recently, we proposed a platform-independent implementation of the CORBA Life Cycle Service specification [6]. Although our initial prototype implementation is based on Java, it is easily portable to any other CORBA-supported language and can be used in different ORBs as it just relies on standard CORBA without any ORB-specific extensions. This has been verified by a second and interoperable implementation in C++.

For state transfer in heterogeneous environments we introduced value types. Value types are a well-known part of the CORBA specification [5]. In contrast to standard CORBA objects, value type objects are copied by value (call-by-value parameter semantics), leading to a complete copy of the value type at the receiving side. As value types actually represent abstract data types [8], they enable an easy description of an object's *implementation-independent* state using standard IDL syntax. Additionally, developers do not have to write special methods for state transfer, as value type objects are marshalled and demarshalled automatically by the ORB. Like standard CORBA objects, value type objects can also be activated. In this case, a value type has to support a standard CORBA interface, which enables a call-by-reference parameter semantics.

Thus, weak heterogeneous object migration can be realised using the CORBA Life Cycle Service in combination with value types. Fig. 1 shows the basic workflow of an object migration according to the Life Cycle Service specification. The specification defines a `LifeCycleObject`-interface that is implemented by our mobile object. The mobile object is actually a value type that supports the `LifeCycleObject`-interface. Beside other methods, it offers a `move()`-method, which implements object migration in conjunction with a remote factory (`GenericFactory`), that enables the creation of objects on remote servers. For this process, another entity—a `FactoryFinder`—is needed for searching for an appropriate factory on a remote server. Possible locations are represented by a particular `FactoryFinder`. Appropriate locations are selected by evaluating parameterised criteria that restrict possible target locations. For fulfilling this task, the `FactoryFinder` has to maintain a repository of possible factories within the `FactoryFinder`'s domain. Therefore, our factory enables registering and deregistering factories including descriptive metadata. After having found such a factory, the object aka value type can be passed by value as a method-parameter using standard marshalling and demarshalling mechanisms. The created value type has to be activated by the factory using the original object's object-identifier. Finally, the original object is removed.

As we also intend to support heterogeneous systems, in which the object's code might not be present at the final location, we provided a solution to dynamic code provision as well. Based on our previous work, the Dynamic Loading Service (DLS) [9], we developed a special type of `GenericFactory` that transparently

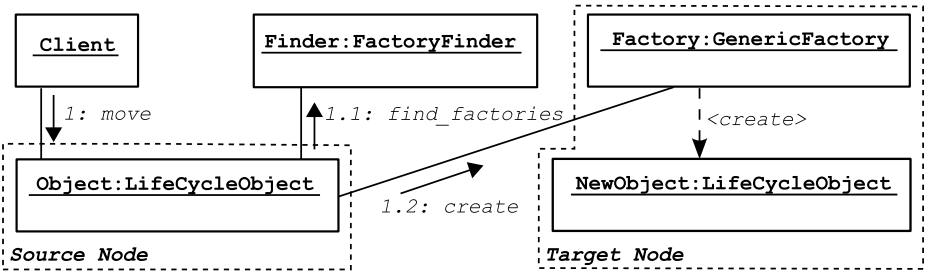


Fig. 1. Object migration using the CORBA Life Cycle Service

enables loading of appropriate platform-specific code on demand. This allows dynamic instantiation of previously unknown objects.

Additionally, our solution enables persistent object references by maintaining the object identifier for the whole lifetime of a mobile object. We achieve this by providing a special type of location service that manages the system's object references.

3 Adaptive Object Migration

In this section, we present our approach of object migration supporting adaptation of state, functionality and code. We give basic background information and outline our basic concept.

3.1 Basic Definitions

For transferring a mobile object in homogeneous environments, usually the state and the code of this object have to be transferred. Therefore, recent systems only separate the state and the code of an object. For addressing heterogeneous platforms, we introduce another abstraction: the separation of state, functionality, and the realising code. Thereby, the functionality of an object is defined by its most derived supported interface. This enables a selection of an appropriate implementation based on a supported interface as already described in Section 2. We call these triples of state, functionality and code the *facets* of the mobile object during its life cycle.

We identified the fact that in various scenarios only parts of the object state are accessed, can be available or should be accessible. If a mobile object moves from one host to another one for fulfilling various steps in a complex workflow, there is data, which is not needed on every target platform. Thus, there is state that we call passive on these platforms. Going back to the "follow me" example from the introduction, the whole state of the personal information manager (PIM) application can be considered as active state on a laptop, as all data can be accessed and modified. As this application object moves to a PDA, certain

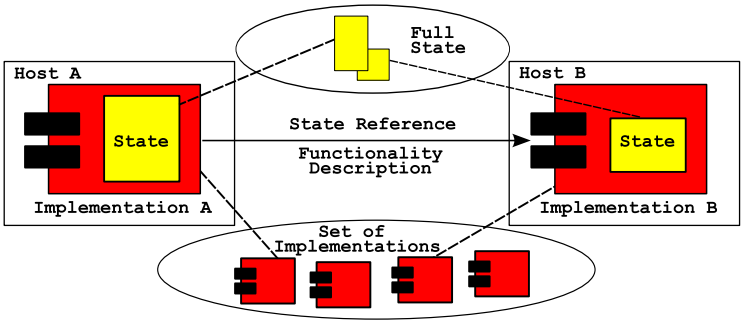


Fig. 2. Adaptation of state according to an object's functionality

data items might not be accessible due to device limitations, like private movies or pictures from the last holiday. In this case, the movies and pictures are passive state that is not available. If the PIM application is accessed from a public web terminal, huge parts of state might not be transferred due to privacy reasons. Thus, this information can also be considered as passive state as it should not be accessible. We define active and passive state as follows:

- *Active state* is represented by the state variables that are used and needed for fulfilling the object's functionality at a certain location.
- *Passive state* is the state of an object that is not needed, not available or not accessible at a certain location.

We believe that in all these three exemplary cases of the PIM application not only the state of an object changes or should be changed but also the interface and the implementation that are provided at the target platform. Fig. 2 shows the adaptation of state and interface on demand according to an object's functionalities' requirements. For this purpose, parts of the full state might become passive. However, this passive state might become active again, whenever the object migrates to another location. Thus, even the passive state has to be stored, as it might be used again later.

3.2 Basic Concept

Fig. 3 shows our concept of adaptive object migration (AOM). We enable adaptation of an object's state according to the required functionality as specified by the interface (cf. Section 3.1). In contrast to previous work, we enable attaching state to a special functionality instead of attaching an object's state to a special implementation (represented by code). For migrating a mobile object, just a reference to the state and a description of the required functionality have to be transferred to the new location (cf. Fig. 2). At the new location, our *transformer factory* is able to create an object that fulfils these requirements containing the active state, which is specified by the required functionality. Our infrastructure

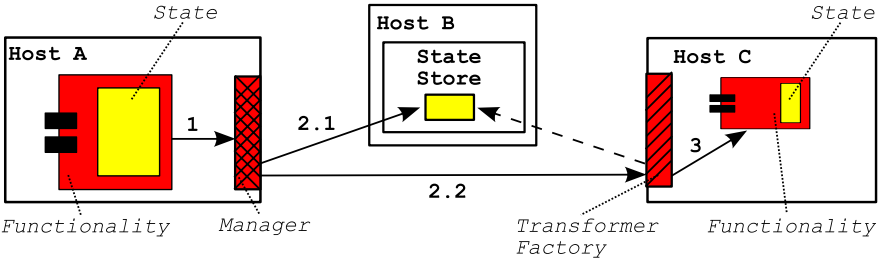


Fig. 3. Basic concept of the adaptive object migration mechanism

supports passivating parts of the state in a *state store* that might be local or remote regarding the current object. Therefore, a *manager* saves the object's current state to the state store (cf. Fig. 3, 2.1). Then, information referencing the state and required functionality is transferred to the target platform (2.2). There, a transformer factory creates the object that is adapted according to the functional and state needs (3). The target side allows receiving and storing the complete state for further local adaptation steps. This requires a local state store and makes especially sense on devices with reduced communication resources and devices that are temporarily disconnected (e.g., a laptop).

As we focus on a mobile environment with resource-limited devices, we support mobile agents. These agents behave autonomously, i.e., they migrate on their own, which reduces resource costs caused by managing the mobile object during its lifetime. Thus, beside a pure migration support, these agents need a facility to get restarted on the target side. We support this by providing a mechanism that automatically invokes a user-defined asynchronous method for agent initialisation upon migration completion. Furthermore, beside supporting migration that is triggered externally, we enable internally-triggered migration allowing agents to autonomously migrate to another node during their life time.

4 Implementation

In this section, we present our implementation for realising our introduced concept for adaptive object migration.

In a first step, the developer has to decide which kinds of functionalities should be supported. As mentioned before, these represent the different facets, which the mobile object can adopt during its life cycle (this affects the behaviour of the object but not the actual object identifier; cf. Section 1).

For any functionality, an IDL interface and the corresponding value type (supporting the interface and specifying the active and implementation-independent state) have to be specified. For supporting our adaptive object-migration service the value type has to support the `LifeCycleObject` interface as well. Additionally, the value type has to inherit from an `AOMObject`[1] value type. This ensures

[1] Implementation classes have the prefix AOM (adaptive object migration).

that the value type contains administrative state needed by our infrastructure. Among other things the object's universally unique identifier (UUID), a reference to the local manager object and a reference to the state store.

Transition of state between the diverse facets is realised by a simple name matching algorithm, i.e., if a state variable has the same name in different value types with the same object identifier, then we assume that these state variables represent identical data. Thus, defining value types containing states with identical names but different types results in an error. A value type's state might become active and passive during runtime, respectively.

In the next step, the developer has to run an IDL compiler that generates code for the interface and value type definitions. As the manager has to extract the current state and write it to the state store (cf. Fig. 3), there is a need for introspection. Therefore, methods for state introspection have to be generated for any CORBA-supported programming language that is not capable of native introspection. Thus, we provide a special IDL compiler for C++ and for Java, we use its native reflection capabilities.

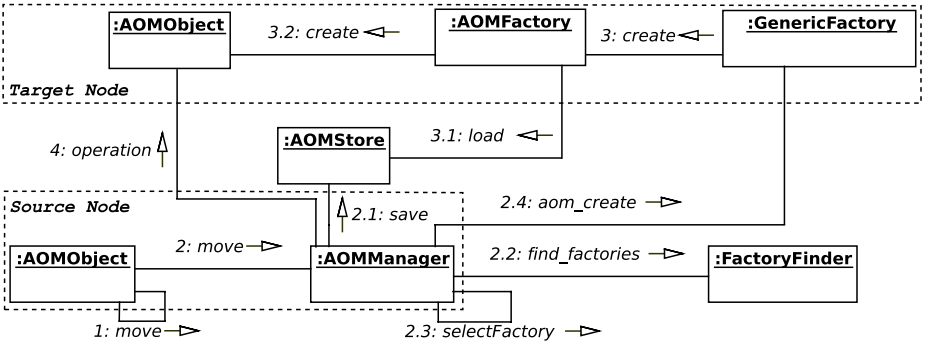


Fig. 4. Process of object migration

After having run the IDL compiler, the developer has to implement the actual value type realisations for all the programming languages and platforms that should be supported (supporting all facets of the mobile object). This also includes the implementation of the life-cycle operations. However, this effort is quite moderate because the whole migration functionality is provided by the `move()`-method of the provided `AOMManager` (cf. Figure 4). There, criteria (cf. Section 2) and information regarding the mobile agent facility have to be provided as parameters. This method just has to be called from within the `LifeCycleObject`'s `move()`-method. An implementation of the other life-cycle operations is simple as well [6]. For instantiation of the first object facet, the developer should use our factory. Therefore, the developer uses `aom_create()` of the `GenericFactory`. Then, the factory implicitly handles activation and sets the environment.

4.1 AOMManager

The `AOMManager` is responsible for triggering the adaptation process on the origin side (cf. manager, Fig. 3). First, it saves the active state of the value type to a state store (`AOMStore`). Therefore, the AOMManager uses the CORBA interface repository for obtaining the state information of a specific value type (i.e., the names of the state variables). For reading the state, native reflection is used for Java; for C++ the generated methods for introspection are used. This state is stored to the `AOMStore` referring the object identifier (cf. Fig. 4, 2.1).

Then, the `AOMManager` is responsible for searching for an appropriate target location for a pending migration. For this task, a known `FactoryFinder`, that is part of the AOM system, is queried on factories representing possible migration targets (within the scope of the `FactoryFinder`). These factories have to fulfil given `criteria`. Our provided `FactoryFinders` support `criteria` defining concrete locations (e.g., DNS-Name, IP-Address) and discrete locations specified by a required functionality (see Section 4.2).

The result of the query is a list of appropriate factories, which are able to instantiate the desired mobile object for realising the migration. Our `AOMManager` is able to select the "best" factory that is used for object migration. In our prototype implementation, we use the factory at the head of the list. As the `AOMManager` is a standard object we provide, user-defined policies can be implemented by providing a custom-build `FactorySelector` object. This object contains a custom `selectFactory()` method that can implement any user-defined selection strategy (cf. Fig. 4).

In the next step, the `AOMManager` invokes the remote method `aom_create()` at the `GenericFactory`. There, the fully-qualified name of the functionality and a reference to the `AOMStore` are transferred to the factory as parameters. In contrast to our recent Life Cycle Service implementation we do not transfer actual value types. This makes no sense as the `AOMManager` has low influence on the actual value type realization at the target. Therefore, the target has full access to the object's state by obtaining the reference to the `AOMStore`. The application developer decides if state should be stored in a local `AOMStore` by passing a boolean parameter (`attached`).

The `move()` method of the `AOMManager` has additional parameters for specifying the name and the parameters for an initialisation method that is invoked after a successful migration, which is especially useful for mobile agents acting autonomously. However, the developer should specify this method as a *one-way* method, which results in an asynchronous invocation.

If the mobile object has been created successfully at the factory, the original object is removed and the reference in the location service is updated to the new location (see [6]).

4.2 FactoryFinder

As already mentioned in Section 2, the `FactoryFinder` represents some kind of abstract location within the CORBA Life Cycle Service specification and

provides methods for querying for appropriate factories acting as targets for object migration.

For supporting our AOM service, the `FactoryFinder`'s `find_factories()` method allows to search for registered factories residing at particular locations and providing particular functionalities, respectively (a list containing every appropriate factory is returned). For this purpose, `find_factories()` receives a CORBA Naming Service `NameComponent` sequence consisting of an <*identifier,kind*>-pair of strings. We specified two possible *kind*-values for our `FactoryFinder`. For a functionality, we specified "IFC" and for a location we specified "LOC". The particular *identifier* represents the functionality and the location, respectively. As `find_factories` receives a `NameComponent`-sequence, searching for specific functionalities at specific locations within the `FactoryFinder`'s repository is possible as well.

4.3 Generic Factory and AOMFactory

The conceptual transformer factory in Fig. 3 is actually represented by two entities in our implementation (`GenericFactory`, `AOMFactory`, cf. Fig. 4). The `GenericFactory` is responsible to create the actual `AOMFactory` for creating adapted objects. Therefore, the `GenericFactory` is able to use our Dynamic Loading Service (DLS, cf. Section 2) to load locally unavailable code on demand. This even enables instantiating previously unknown objects.

At the `AOMFactory`, the value type is created according to the transferred interface specification. If needed, the factory is even able to adapt the implementing value type according to the actual platform and criteria requirements. The active state is then loaded from the `AOMStore` into the adapted value type implementation. Therefore, the CORBA interface repository is queried for the state information of the actual value type, i.e., the names of the actual state attributes. This information is loaded from the state store and set in the value type by using native Java reflection and generated C++ introspection methods, respectively. After this process, the value type is activated, which enables accessing the object from remote clients.

If appropriate, i.e., if the variable `attached` is true, a local `AOMStore` is used for storing the complete state referring to a specific object identifier. The `AOMStore` has to be accessible as a CORBA initial reference (if there is no `AOMStore` as an initial reference, an error is reported to the application). Then, the complete state regarding a particular object identifier is loaded, stored locally and the state store variable in the actual value type is set to the local store. Later, instances can reference this state store remotely.

4.4 AOMStore

The `AOMStore` is an entity that is able to store the current state of a mobile object (cf. state store, Fig. 3). This entity is a plain CORBA object that can be located locally or remotely to the mobile object. The `AOMStore` contains the current state of an object related to an object identifier and offers two methods

for loading and saving the state, respectively (cf. Fig. 5). The `load()` method takes the object identifier and the names of needed active state variables of the desired value type as a parameter and, thus, is able to return the appropriate part of the state.

```
1  interface AOMStore{
2      void save (in OctetSeq oid, in AnySeq state);
3      AnySeq load (in OctetSeq oid, in StringSeq values);
4      AnySeq load_full (in OctetSeq oid);
5  };
```

Fig. 5. IDL interface of the `AOMStore`

The actual implementation of the `AOMStore` is left open to the developer. In our prototype, we realised a naive implementation that is based on Java HashMaps. However, an implementation that is realised by serialising the mobile object's state to external storage is possible as well (reducing memory load).

4.5 Accessing the AOM Object

In heterogeneous environments containing mobile objects, persistent object references are needed for clients to access the mobile object anytime. We realised these object references using a special location service [6]. Thus, object references point to the current location for the object's whole lifetime. However, adapting the functionality (interface) of a mobile object will probably result in remote call exceptions on the client side, as particular methods might not be specified within the new interface any more.

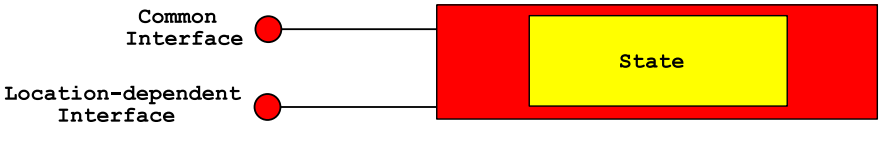


Fig. 6. Common and location-dependent interface of a mobile object

This problem can be solved by assuming that the mobile object has a common interface that does not change and a location-dependent interface that might change (cf. Fig. 6). This can be realised by a value type that has to support the common interface while optionally supporting other location-dependent interfaces. Thus, the location-dependent interface is just used for platform-internal purposes. Nevertheless, this interface is remotely accessible and can be used by clients (these have to handle exceptions if a special location-dependent interface is not existent).

As already mentioned in Section 1, our system is able to realise an agent that is able to adapt to the current location's needs. An adaptive mobile agent will have management methods for intervening into the agent's workflow (common interface). Besides, the agent will also offer methods that can be used on the current local node, e.g., for collecting data (location-dependent interface).

Beside pure migration support, these agents need a facility to restart their thread on the target side. These methods for initialising the agent are supported by the `AOMManager`. However, these methods should be declared *one-way* as this enables an asynchronous invocation. The `AOMManager` receives the method name and parameters and, then, is able to automatically invoke the method after a successful migration. Therefore, the CORBA dynamic invocation interface (DII) is used, as there is no way for the `AOMManager` to obtain the actual interface of the object after migration (the `AOMFactory` is able to adapt the object to the current context's needs).

5 Example Application

For outlining the development process and the necessary steps to implement an application using our adaptive mobile object concept, we present a simple distributed raytracing application.

Starting with the definition of a basic workflow, a developer has to identify the necessary interfaces and states of an adaptive mobile object. In context of our example application, first, the user defines the layout of a rendering scene, then submits the scene for processing and, in a final step, accesses the resulting frame. As indicated by the described workflow, we identified three different steps that can be modelled by an object supporting three facets providing different interfaces and state as displayed in Fig. 7.

We implemented a Java-based client application providing object facets for step one and three, represented by the `PerpareJob` and `JobResult` interface together with their associated value types `PrepareContainer` and `ResultContainer`. As these steps cover only get- and set-operations they can be performed on an arbitrary node running the application. The second step, however, performing the actual rendering process specified by the interface `RayTraceJob` and the associated value type `RaytraceContainer` requires slightly more resources and might be time consuming. Therefore, the rendering process should be done on a lightly loaded node having a powerful CPU for a highly optimised rendering implementation. The first criteria can be satisfied by a custom factory finder being able to locate such a node. The second one can be fulfilled by implementing the object facet in C++ using an appropriate library.

After specifying the scene via the set and add methods of the `PrepareJob` interface, a user calls `renderJob()` which turns the object into an agent, initiating the migration to the `RaytraceJob` interface that supports the rendering process (cf. Fig. 8, lines 2-6). Migration to an interface instead to a concrete location enables load balancing as our custom `FactoryFinder` selects an idle node. Along the lines to the interface change, the state is changed. While it is

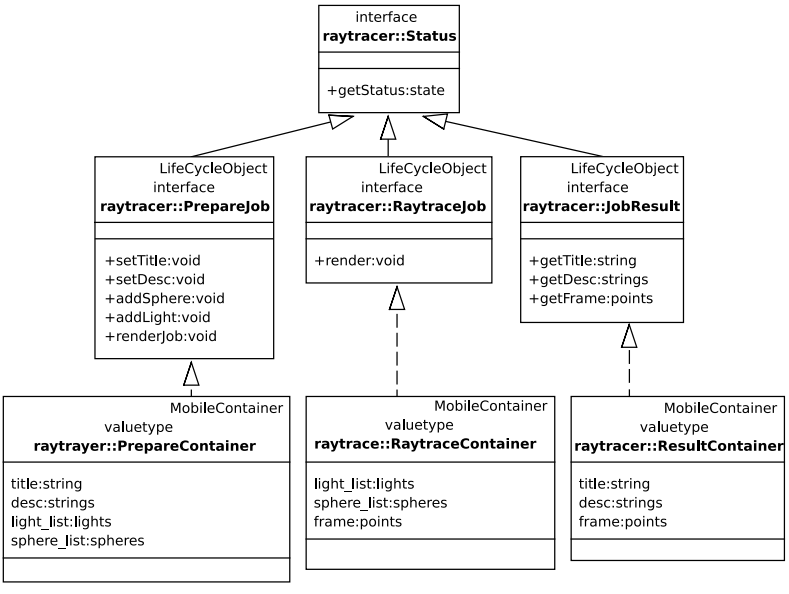


Fig. 7. Object facets of a basic distributed raytracing application

important for a user to have a title and description of a rendering scene, these information is negligible for the rendering process so these meta data is left out in the `RaytraceContainer` value type and, therefore, not migrated. Instead, it contains an additional field for the resulting frame. The migration process is finished by invoking the `render()` method as displayed in Fig. 8 (line 9). After successfull completion (line 10), the object at the source-location is implicitly removed by our service implementation. After processing the scene, the agent returns to its original location and changes the interface to provide the results. All three supported interfaces extend a simple `Status` interface providing information about the current workflow state. Thus, regardless of the current location, an application can monitor the execution process and, based on this information, narrow to one of the facets, e.g., to finally access the frame (`JobResult`).

For evaluation of the implementation, we compared the implementation of the presented example application using our former plain Life Cycle Service (LCS) with the implementation using the AOM infrastructure. Therefore, we implemented all supported facets and the LCS-based variant in Java and C++. The LCS object implementation supports all the interfaces by a union value type comprising the complete state of all three facets. For ruling out network irregularities, the measurements were performed on a single AMD Opteron 2.2 GHz Linux server machine using JacORB 2.1 for Java and the Orbacus 4.30 for C++. We measured the roundtrip between the client application and a process acting as a rendering node without actually rendering the frame, as this would distort the measurements. Table 1 displays the results for the LCS and the AOM,

```
1  public void renderJob() {
2      NVP [] criteria  = new NVP[1];
3      Any ifcAny = orb.create_any();
4      ifcAny. insert_string ("IDL:RayTraceJob:1.0");
5       criteria [0]  = new NVP("interface",ifcAny );
6      Object obj = manager.move(criteria, this);
7
8      RayTraceJob job = RayTraceHelper.narrow(obj);
9      job.render();
10 }
```

Fig. 8. Realisation of interface-directed migration (Java)

Table 1. Comparison of migration of a raytracing application using the standard CORBA Life Cycle Service and AOM

Migration	Source	Target	Duration (ms)
LCS	Java	Java	9.62 (± 0.48)
	Java	C++	5.08 (± 0.25)
	C++	C++	4.30 (± 0.21)
AOM	Java	Java	130.46 (± 6.52)
	Java	C++	82.70 (± 4.14)
	C++	C++	19.28 (± 0.96)

respectively. The AOM implementation using only facets in C++ is almost 7 times faster than the Java implementation. One of the reasons is the usage of the interface repository for state transformation in Java (a query to the interface repository takes about 20ms). In C++ this is done by custom generated methods included in the value type code. As the comparison to the plain migration support provided by the LCS shows, flexibility provided by our adaptive mobile object infrastructure comes at some cost. However, we believe that benefits of adaptive object migration outweigh this performance penalty. Furthermore, our prototype has not been optimised for performance.

6 Related Work

In the past, many mobile agent systems have been developed. However, most of them only provide mechanisms for migration in homogeneous environments (e.g., MOA [10], Mole [11] or Aglets [12]). In general, these systems are Java-based and rely on the Java serialization mechanism. In this case, the state of an agent is tightly attached to the actual implementation. Thus, migrating a specific version to a new place with another implementation version will lead to serialization errors. Moreover, no adaptation of the mobile agent's state, functionality or code is supported at all.

Nevertheless, mobile agent systems for heterogeneous environments have been developed as well. In [3] a platform-neutral approach of agent migration is presented, based on transferring a blueprint instead of the code. This approach is realised by assuming that an agent consists of a set of different components. Then, a special *AgentFactory* is able to create an executable agent consisting of the right components from a received blueprint and related state. However, there is no support for the adaptation of state or functionality.

Takashio et al. present a mobile agent framework for supporting follow-me applications [1]. This framework allows mobile agents to adapt their code to the current context, i.e., applications are able to benefit from using high-performance implementations. However, this solution relies on Java and thus does not support heterogeneous programming-language environments. Additionally, there is no concept for adaptation of state and functionality, respectively.

Choy et al. describe a CORBA environment supporting mobile agents based on mobile CORBA objects [13]. Based on the Life Cycle Service a concept was developed, but apparently not implemented. Furthermore, the concept does not support adaptation of the mobile agent's state or code.

An Agent Transport Service (ATS) was specified in [14]. There, the CORBA Life Cycle Service was considered, but finally sorted out in order to support lightweight agents. As in our solution, all migration methods are offered entirely by the platform. In contrast to our approach, ATS does not provide support for adaptation of the agent's state or functionality.

Bellavista et al. propose the SOMA programming framework for mobile agents [4]. This work offers compliance to CORBA, and to the mobile agent standards MASIF and FIPA Thus, it achieves a good interoperability with many other mobile agent systems. However, SOMA neither supports adaptation of the mobile agent's interface nor of the agent's state.

Brandt et al. suggest reassembling agents from smaller subcomponents [15]. This allows exchanging environment-dependent implementations at runtime by selecting an appropriate implementation for a specific environment at runtime. Our approach is even better compared to this solution as our Dynamic Loading Service (DLS) [9] also allows selecting appropriate implementations for specific environments, and in contrast to Brandt et al. we also support changing the object's interface during runtime.

Almeida et al. propose a dynamic reconfiguration service for CORBA [16]. This service is able to upgrade objects without taking them offline by entailing operations for migration, replacement, addition and removal of objects. In contrast to our approach, developers have to implement methods for inspecting and modifying the state on their own and the dynamic reconfiguration service does not allow switching the object's interface or state adaptation.

The work of Garbinato et al. introduces Frugal Objects (FROBs) for mobile computing [17]. FROBs are objects within an event-based computing model, which provide adaptability according to their interface (which is represented by acceptable events) and their code. These objects support migration, but a drawback of this solution is the requirement of using a special non-intuitive

FROB programming model that does not allow loops, forks or synchronization primitives.

The programming language Self [18] allows dynamic adaptation of local objects during runtime. Self is an object-oriented programming language based on the prototype concept, which allows manipulation of an object's methods and state at runtime. Instead of using inheritance for specialisation, an object developer copies an existent object and modifies the behaviour to the current needs. There is also a distributed Self variant called dSelf [19]. However, there is no support for heterogeneous object migration supporting other programming languages.

Table 2. Overview of capabilities of related migration approaches

System	Environment	Code	Functionality	State
Self	Local	Static	Dynamic	Dynamic
dSelf	Homogeneous	Static	Dynamic	Dynamic
Aglets	Homogeneous	Dynamic	Static	Static
Takashio et al.	Homogeneous	Dynamic	Static	Static
FROBs	Homogeneous	Dynamic	Dynamic	Dynamic
Agent Factory etc.	Heterogeneous	Static	Static	Static
ATS	Heterogeneous	Static	Static	Static
SOMA	Heterogeneous	Static	Static	Static
Brandt et al.	Heterogeneous	Dynamic	Static	Dynamic
AOM	**Heterogeneous**	**Dynamic**	**Dynamic**	**Dynamic**

Table 2 shows an overview of the capabilities of the presented related work. Our approach is the only one supporting dynamic migration of code, functionality and state in heterogeneous environments.

7 Conclusion and Future Work

We presented a novel approach of a dynamic adaptive object migration service for CORBA. Our service is build on top of the CORBA Life Cycle Service specification and, thus, provides compatibility to standard Life Cycle Service implementations. In contrast to previous implementations, we propose a separation of state, functionality and code. This enables providing an adaptive service that is capable of supporting heterogeneous platforms and programming languages.

CORBA is not per se a platform for Ubiquitous Computing as it is a fully-fledged middleware originally targeting at arbitrary distributed applications running on standard computers. This situation changes with the upcoming CORBA/e standard for embedded systems [20]. The new standard specifies a *compact profile* that supports CORBA value types. Thus, this enables a general transfer of our concept to embedded devices. However, as the CORBA/e standard does not support many dynamic features of standard CORBA we have to

adapt our concept to these new requirements (e.g., CORBA/e does not support the interface repository and the dynamic invocation interface). However, using our approach in context of Ubiquitous Computing applications is already possible. There are CORBA implementations like TAO [21] that address small and also embedded devices even under real-time conditions.

Despite this fact, we will investigate to transfer our approach to other technologies, like, e.g., XML RPC and Web Services, for providing an infrastructure that needs fewer resources.

Our concept for adaptive object migration does not consider security so far. However, we enable applications handling security themselves by transferring a reference to actual state only. An integration of security mechanisms into our infrastructure is possible as well. Thus, for realising security, the remote state store will have to be trusted, allowing to secure the access to parts of the object's state. We will investigate the CORBA Security Service [22] for this purpose.

As the measurements showed, there is still the opportunity for performance optimisations. An obvious improvement would be the usage of custom code generation instead of native reflection for Java. However, we will also investigate to optimise the state transfer. For applications, which are able to statically define all facets before runtime, this can be realised by transferring a union value type representing the complete state of all facets instead of lists containing *Any* values. A static value type reduces serialisation costs and speeds up access to the state store.

Finally, we would like to offer extended support for developing applications basing on adaptive mobile objects. This includes a graphical UML-based modelling tool offering annotation support for the diverse facets of adaptive mobile objects and specialised code generators for multiple languages. Such a support should ease application development and help the developer to keep a consistent view across different facets, all supported platforms and multiple versions of adaptive mobile objects.

References

1. K. Takashio, G. Soeda, and H. Tokuda. A mobile agent framework for follow-me applications in ubiquitous computing environment. *ICDCSW*, 2001.
2. C. P. Kunze, S. Zaplata, and W. Lamersdorf. Mobile Process Description and Execution. In *6th Int. Conf. on Distr. App. and Interop. Sys.—DAIS*, 2006.
3. F.M.T. Brazier et al. Agent factory: generative migration of mobile agents in heterogeneous environments. In *ACM Symp. on Applied Comp.*, pages 101–106, Madrid, Spain, 2002.
4. P. Bellavista, A. Corradi, and C. Stefanelli. CORBA Solutions for Interoperability in Mobile Agent Environments. In *2nd Int. Symp. on Distr. Obj. and Appl.—DOA'00*, pages 283–292. IEEE, Sep 2000.
5. Object Management Group (OMG). Common object request broker architecture. OMG Document formal/2004-03-12, 2004.
6. R. Kapitza, H. Schmidt, and F. J. Hauck. Platform-Independent Object Migration in CORBA. In *OTM'05*, LNCS 3760, pages 900–917. Springer Verlag, Oct 2005.

7. Object Management Group (OMG). Life Cycle Service Specification. OMG Document formal/2002-09-01, 2002.
8. B. Liskov and S. Zilles. Programming with Abstract Data Types. In *ACM SIGPLAN Symp. on Very High Level Lang.*, pages 50–59. ACM Press, 1974.
9. R. Kapitza and F.J. Hauck. DLS: a CORBA service for dynamic loading of code. In *OTM Confed. Int. Conf.*, Sicily, Italy, 2003.
10. D.S. Milojicic, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *4th USENIX Conf. on OO Tech. and Sys.*, pages 179–194, 1998.
11. M. Strasser, J. Baumann, and F. Hohl. Mole: A Java based mobile agent system. *2nd ECOOP Works. on Mob. Obj. Sys.*, 1997.
12. D.B. Lange and M. Oshima. Programming and Deploying Java Mobile Agents with Aglets, 1998.
13. S. Choy, M. Breust, and T. Magedanz. A CORBA Environment Supporting Mobile Objects. Technical Report White Paper Draft Version 1, IKV++ GmbH, 1999.
14. C. A. Mendez and M. Mendes. Agent Migration Issues in CORBA Platforms. In *5th Int. Symp. on Autonom. Descentral. Sys.—ISADS'99*, Mar 1999.
15. R. Brandt, C. Hörtnagel, and H. Reiser. Dynamically Adaptable Mobile Agents for Scaleable Software and Service Management. *Journal of Comm. and Netw.*, 3(4):307–316, Dec 2001.
16. J. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *3rd Int. Symp. on Distr. Obj. and Appl.—DOA'01*, pages 197–207. IEEE, Sept 2001.
17. B. Garbinato, R. Guerraoui, J. Hulaas, O. L. Madsen, M. Monod, and J. H. Spring. Mobile Computing with Frugal Objects . Technical report, EPFL I&C, 2005.
18. R. B. Smith and D. Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conf.*, Aug 1995.
19. R. Tolksdorf and K. Knubben. Programming Distributed Systems with the Delegation-based Object-Oriented Language dSelf. In *ACM Symp. on Appl. Comp.—SAC'02*, 2002.
20. Object Management Group (OMG). Corba/e draft adopted specification. OMG Document ptc/06-05-01, 2006.
21. D. C. Schmidt. Real-time CORBA with TAO (The ACE ORB), May 2004.
22. Object Management Group (OMG). Security Service Specification. OMG Document formal/02-03-11, 2002.

A Novel Object Pool Service for Distributed Systems

Samira Sadaoui and Nima Sharifimehr

Department of Computer Science,
University of Regina, Regina, SK Canada S4S 0A2
{sadaouis, sharifin}@cs.uregina.ca

Abstract. Efficient resource management is a great concern for many enterprise developers. One of the most important resources that should be carefully managed is the amount of memory consumed by applications. This paper proposes a novel distributed memory management framework, namely Distributed Object Pool Service (DOPS) which efficiently manages the lifecycle of entity objects in distributed systems. The generality and scalability of DOPS makes it reusable for any commercially distributed middleware, such as CORBA and COM+. In addition, since there is no absolute proper configuration to be used efficiently for object pools of any software system, we offer a workload simulator which simulates target systems in order to find the appropriate configuration for them. Using our workload simulator, we conduct several experimental tests using CORBA. These experimentations demonstrate the effectiveness of DOPS including the positive affect of dynamic sizing of object pools and the productivity of preparing fine-granularity configuration facilities.

Keywords: Memory Management, Object Pooling, CORBA, Workload Simulator.

1 Introduction

Object pooling is an important aspect in the design and development of distributed object-oriented systems. Two different categories of distributed objects exist: persistent and transient. Persistent objects are stateful objects whose lifecycles are longer than of their container (i.e. process or thread) that creates them. In contrast, the lifecycle of transient objects is limited by the scope of their container. So, the internal state of persistent objects should be stored in persistent object storages (i.e. ODBMS, File System). However, retrieving objects from persistent storages is a time consuming process that dramatically reduces system performance. Thus, it is better to keep a copy of all persistent objects in memory. But in enterprise systems which have billions of persistent objects, this scenario will not work because of the memory limitation. Hence, the primary goal of object pooling is to improve the performance by keeping a limited number of objects with more access probabilities in memory.

Swapping useless objects from the memory is not a new idea. For instance, paging systems have been widely used at the memory manager unit of operating systems [7]. But, there are too many factors that differentiate the paging systems from object pooling in distributed object-based applications. For example, in paging systems, there is a unified behavior to all replaceable pages, while object pooling systems have access to meta-knowledge about their objects. Also, paging systems consider the pages as dumb

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1757 – 1771, 2006.
© Springer-Verlag Berlin Heidelberg 2006

objects that can not influence their lifecycle, but on higher level of abstraction, in object pooling systems, objects have the processing power. The differences originate from the more abstract concepts involved in object technology. So, we use the object pooling approach, which considers object-orientation abstract concepts, to sweep objects from memory. There are many research projects and commercial products such as Oracle JOC [11], JBossCache [12] and Apache JCS [13] which provide object pool facilities for available middlewares. However, all of them are middleware-dependent and also they do not provide sufficient flexibility and reusability to be used by different software systems. The main issue that these products do not address is fine-granularity configuration possibility. Beside these products, there are some design patterns such as Object Pool [3], Evictor [5], Lazy Acquisition [1] and Leasing [2] which can be used together to build an object pooling service. However, there is no integrated framework pattern to prepare a detailed architecture for object pooling service. Also they do not address any adaptive facility to change the behavior of object pooling when dealing with the dynamic nature of software systems. On the other hand, none of the existing solutions provides a facility to help figuring out an appropriate configuration for a specific software system.

Distributed reference counting approach [15] is another solution to sweep distributed objects from memory. However, this solution comparing to object pooling approach suffers from the following flaws: (1) no limitation is applied on the number of available objects in the memory; (2) probable frequent accesses to an object after releasing all of its references are neglected; (3) broadcasting reference management messages forces additional network traffic; (4) erroneously written client-side code result in the failure of solution applicability. According to these flaws, using distributed reference counting is not recommended in practice. On contrary, object pooling not only does not involve the flaws mentioned above but also reduces the costs of development and maintenance through its client transparency.

Considering the problems above, in this paper we introduce a novel framework for object pooling named Distributed Object Pooling Service (*DOPS*). *DOPS* offers an applicable and efficient solution to prepare a middleware-independent object pooling service. Also, it offers an adaptive approach to automatically figure out the optimum size of object pools. Although some configurations should be done manually, we provide a workload simulator that allows us acquiring the most appropriate configuration for any specific system. We have fully implemented *DOPS* with *Microsoft Visual C++ 6.0* (the whole software contains 64 classes and 10,000 lines of code) and we have conducted several experimentations to demonstrate the effectiveness of *DOPS* in practice.

The remainder of this paper is organized as follows. Section 2 describes the overall architecture and design criteria of *DOPS*. Section 3 explores the facilities of our proposed framework considering our workload simulator. Section 4 illustrates the results of workload simulations that evaluate the performance of *DOPS*. Finally, Section 5 concludes the paper with some future directions.

2 DOPS Design

2.1 DOPS Architecture

As shown in Fig. 1, *DOPS* comprises seven major components: (1) a *DOPS* Connector which offers a connector interface to distributed middlewares; (2) a Statistics Manager

which gathers statistical data about incoming requests and pools activities; (3) a Performance Monitor which uses collected data by the Statistic Manager to show the throughput of *DOPS*; (4) a Pool Container which manages the containing object pools and dispatches the incoming requests to them. Also, it fetches objects to related object pools whenever it can not find the required objects in its object pools; (5) an Intelligent Pool Container Manager (*IPCM*) which is responsible to figure out the appropriate size of each object pool existing in the pool container; (6) a Reap Manager which is responsible to determine victim objects that should be thrown away from each object pool; (7) an Eviction Signal Generator which sends activation commands to the Reap Manager.

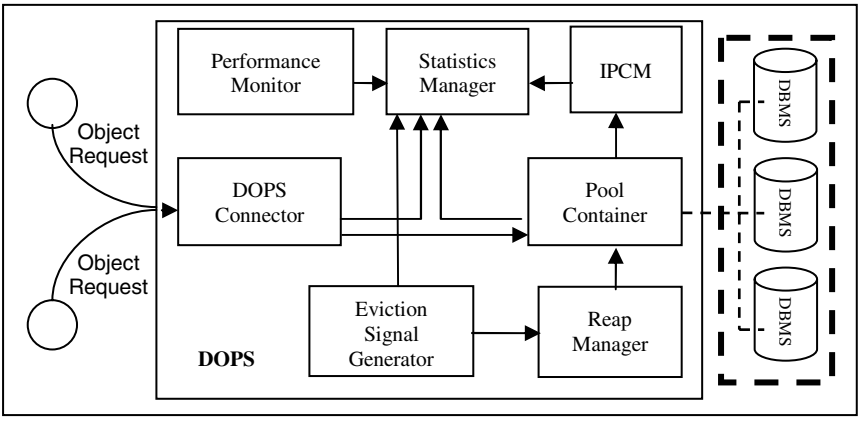


Fig. 1. DOPS Architecture

According to the architecture above, whenever a middleware directs an object request to *DOPS*, at first it goes through the *DOPS* Connector. This latter assures that all the incoming requests conform to the expected request format of *DOPS*. Then, *DOPS* Connector sends the object request to the Pool Container and also updates the statistical data about incoming requests. When the Pool Container receives the object request, at first it determines the target object pool. Later, it searches the target object pool to find the required object to perform the request. If it does not find that object inside the target object pool, it will fetch the object from associated persistence storage (i.e. DBMS, File system) to the related object pool. Then, the Pool Container returns a reference to that object. Consequently, the middleware calls the request on the object and releases the object reference. Since, the capacity of the Pool Container is fixed and limited, the Pool Container restricts the size of each object pool. The Pool Container uses *IPCM* to figure out the appropriate size of each object pool. On the other hand, whenever the Eviction Signal Generator sends activation signals to the Reap Manager, this latter process the signals and starts evacuation of the associated object pools to meet their allowed size by the Pool Container.

2.2 DOPS Design Criteria

DOPS Connector. Integration of distributed middlewares into DOPS is an important issue that we address using a connector interface that includes two methods: (1)

GetPooledObject method which returns a reference to the requested object and it is used to access an object registered in DOPS; (2) ReleasePooledObject method which has to be called by the middleware to release the acquired reference when processing of the object request is complete. This interface allows any distributed middleware which provide message profiling or general proxies to be integrated into DOPS. For example, here we explain the integration of CORBA [15] into DOPS. There is a component in CORBA namely POA which is responsible for managing the mapping between CORBA objects (or remote object references located on the client-side) and servants (or concrete objects located on the server-side). POA can be configured in such a way to perform this mapping through custom Servant Managers. POA supports two different kinds of Servant Managers: Servant Activator and Servant Locator. However, the integration into DOPS is only possible through servant locators, because a Servant Locator is responsible for supplying a servant for a CORBA object each time that a client makes a request on that object. Hence, this kind of Servant Manager is appropriate for connecting CORBA to DOPS. The Servant Locator interface has a pair of methods as follows: (1) preinvoke method which is called when the POA receives a request for an object. This method is called to acquire an appropriate servant for the requested object; (2) postinvoke method which is called when the processing of the object request is complete. Calling this method allows the implementation of Servant Locator to dispose of the servant when processing of the object request is complete.

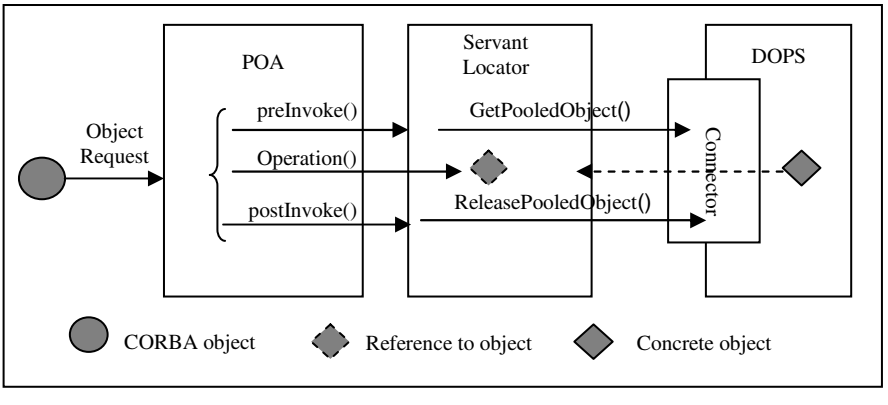


Fig. 2. *CORBA* integration into *DOPS*

Hence, there is a straight approach to integrate a servant locator to *DOPS* connector. Within *preInvoke* method of the servant locator, we should return the object acquired through calling *GetPooledObject* method of *DOPS* connector. Also within *postInvoke* method of the Servant Locator, we should call the *ReleasePooledObject* method of *DOPS* connector to release the acquired object reference (cf. Fig. 2).

Pool Container. Pool Container manages the containing object pools and dispatches the incoming requests to them. Also, it fetches objects to related object pools whenever it

can not find the required objects in its object pools. We should force objects that want to register in pools to implement certain interfaces with the following capabilities:

- Diverse data storages and retrieval mechanisms can be used by objects. So, Pool Container is independent from persistence services or data storages that different objects use.
- Integrated reference counting is prepared. So, Pool Container uses local reference counting mechanism that manages the lifecycle of concrete objects independent from any middleware.
- The objects of pools can interfere in reaping activities and help to make correct decisions about their eviction as victims. So, Pool Container considers the consultation with candidate objects as victims to make correct decisions during the eviction process.

Since Pool Container contains a number of object pools, it has to share the available memory capacity efficiently. Hence, we have to consider the variable size of pools during the design of this container. Also, we connect Pool Container to a component namely *IPCM* to figure out the appropriate size of each pool during the life cycle of the container.

IPCM. In order to restrict the size of each pool and select it optimally, we design IPCM. IPCM assumes that all pools are inserted in a common container with a specified size. So, the total size of pools will be restricted to an upper bound which is the size of their container. The size of the pool container must be configured manually with respect to the availability of memory resources. However, the size of each pool is determined through IPCM. This approach monitors two parameters for each pool: activity ratio and hit ratio. Hence, IPCM is a feedback-based approach that uses activity ratio of each pool to make better decisions. Consequently, using the values of these parameters, an optimal size for each pool can be figured out. Considering both activity ratio and hit ratio, an acceptable total hit ratio for the pool container is gained but also the occurrence of starvation for each pool can be prevented. We define the activity ratio of a pool as follows[1]:

$$ARPool_i(t) = \frac{PTIRPool_i(t)}{CTIR(t)}.$$

Where:

- $ARPool_i(t)$ is the activity ratio of i'th pool.
- $PTIRPool_i(t)$ is the total incoming requests on i'th pool.
- $CTIR(t)$ is the container's total incoming requests.

And the *hit ratio* of a pool is defined as follows:

$$HRPool_i(t) = \frac{PTHNPool_i(t)}{PTIRPool_i(t)}.$$

[1] All equations explaining *IPCM* are based on *t* which shows time. So, the dynamic behavior of *IPCM* comes from this parameter.

Where:

- $HRPool_i(t)$ is the hit ratio of i-th pool.
- $PTHNPool_i(t)$ is the total hit number of incoming requests on i-th pool.
- $PTIRPool_i(t)$ is the total incoming requests on i-th pool.

To determine the size of each pool according to above parameters, *IPCM* considers the following points: (1) Decrease the size of pools with lower activity ratios; (2) Increase the size of pools with lower hit ratios. So regarding these points, we create the following mathematical equation to figure out the size of each pool:

$$basesizePool_i(t) = ARPool_i(t) * C - HRPool_i(t) * ARPool_i(t) * C.$$

Where C is the total size of the pool container**.** However, if we calculate the size of each pool with the equation above, the summation of sizes will be less than the total size of the pool container. Hence, to prevent wasting the available space of the pool container, we divide the remaining capacity equally between all pools as follows:

$$sizePool_i(t) = basesizePool_i(t) + \left(\sum_{j=1}^{N} HRPool_j(t) * ARPool_j(t) * C \right) \Big/ N$$

Where N is the number of pools existing in the pool container.

Reap Manager. Reap Manager contains different reap strategies that apply object replacement policies on associated object pools. In fact, there is a one to one mapping between reap strategies and object pools. This one to one mapping structure prepares precise configuration capability for each object pool. Since, the sequence of callings on each object pool is different, no single strategy works best. Therefore, we provide such a design that allows us to associate each object pool with an appropriate reap strategy. Whenever a reap strategy receives an activation signal from Eviction Signal Generator, it explores its associated object pool to find victim objects. Then, victim objects are thrown away to prepare free space in the object pool for loading new objects. Each strategy must satisfy the following properties:

- Use simple and fast explorative techniques to find victim objects.
- Select optimum objects as victims in such a way to increase the throughput.

Some of the classic approaches which are used as reaping strategies are: *Least Recently Used* (*LRU*) [7], *Most Recently Used* (*MRU*) [7], *oldest object* [4] and *object lease* [2]. However, sizes of objects in each object pool are different and also the costs of loading those objects to object pools are not the same. Consequently, we better use cost-aware approaches such as greedy-dual-size [14] algorithm. But, using cost-aware approaches is not necessary for those kinds of object pools which contain objects with the same size and load cost. Thus, we must consider the following points to choose an appropriate reaping strategy for each object pool:

- The size of objects which are loaded to the given object pool.
- The cost of loading different objects to the given object pool.
- The sequence of callings on different objects of the given object pool.

Choosing victims from working sets of a pool causes frequent load/unload actions and hence decreases the performance of the pool. To select optimum victims, the reaping strategies at first determine the working sets. Afterwards the selection is performed among objects which do not exist in those sets. To determine the working sets, intelligent approaches are used. One of the most applicable approaches is *Markov Model* [6] which is also used in speech recognition and data compression techniques. Using pool's *Markov Model*, eviction policies can find out which objects or group of them are probable to be needed. Using this information, eviction policies can determine the working sets. And as a result, selecting victims from those determined working sets can be avoided. On the other hand, prefetching objects to pools is possible through anticipation.

Since there is not any absolute proper reaping strategy to be used efficiently in the pools of any software system, we emphasize the simulation of target systems through the workload simulator that we define and implement.

Eviction Signal Generator. Eviction Signal Generator schedules the Reap Manager. It sends activation signals to the reaping strategies registered in the Reap Manager (cf. Fig. 3). Hence, Eviction Signal Generator associates different eviction triggers to existing reaping strategies. Important design criteria about this component are as follows:

- Each eviction trigger can send eviction signals to one or more reap strategies.
- Each eviction trigger can be a logical combination of other eviction triggers.

The most famous strategies [4] to implement eviction triggers are as follows:

- Explicit call: clients use explicit calls to send activation signal.
- Connection closure: when a client's connections to the server get closed, the server side connection managers send activation signals to the reap strategies.
- Periodic eviction: the activation signals are sent in regular intervals to the reap strategies.
- Object threshold: the activation signals are sent when the number of objects, available in pools, goes beyond a determined threshold.
- Object activation: when incoming requests cause the pool container to load objects to an object pool, the activation signals are sent.

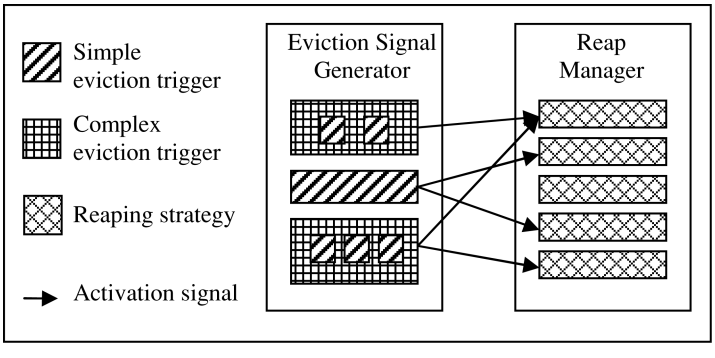


Fig. 3. Eviction Signal Generator Structure

3 Workload Simulator Development

As we have illustrated in the previous section, the appropriate configuration of DOPS involves many factors that can not easily be figured out mathematically. So, we need an instrument to figure out the most appropriate configuration. On the other hand, *DOPS* introduces an appropriate level of abstraction that makes it independent from the remote calling mechanisms and distributed middlewares. Consequently, *DOPS* offers a workload simulator that simulates remote callings and generates requests on objects according to different frequency functions and locality functions (cf. Fig. 4).

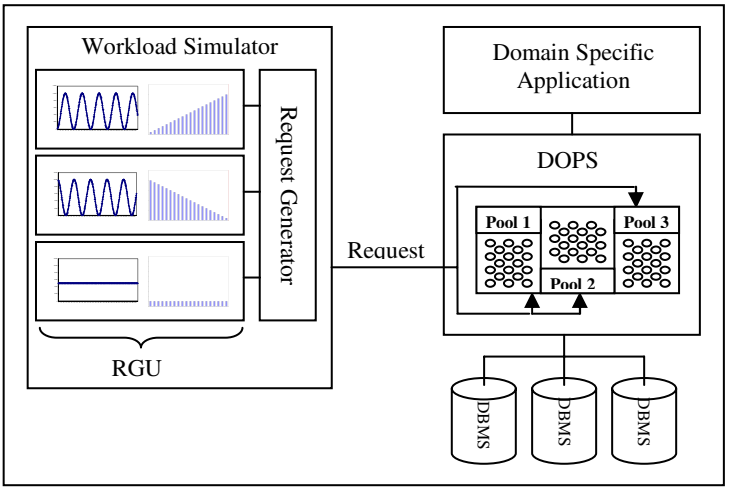


Fig. 4. The Workload Simulator

We define Request Generation Units (*RGU*) (cf. Fig. 4) in the workload simulator to generate requests on each object pool. Each *RGU* includes the following identifiers: (1) a Request Frequency Function (*RFF*); (2) a Request Locality Function (*RLF*). The calling sequences of the workload simulator depend on the determined *RFF* for each *RGU*. In fact, each *RFF* determines the total incoming requests on the associated object pool according to a given time spot. Hence, the workload simulator uses *RFF*s to figure out the upper bound number of requests that should be sent to *DOPS* on a specific time spot. In order to simulate request localities, we define *RLF*s for each *RGU*. In fact, a *RLF* is a Probability Density Function (*PDF*). Hence, we generate random requests according to associated *RLF*s to object pools. We perform the process of random number generation from *PDF*s using the method described by Kronmal and Peterson in [10]. The workload simulator during the simulation time repeats the following steps on each registered *RGU*: (1) calculating the number of requests according to current time using *RFF* unit of *RGU*; (2) generating the calculated number of random requests using *RLF* unit of *RGU*; (3) sending generated requests to the object pool associated with *RGU*.

According to the architecture shown in Fig. 4, we can change the configuration of the workload simulator and determine *RFF*s and *RLF*s. Afterwards the performance measurement of the system is prepared using *DOPS* monitors registered with each eviction policies. So, when the acquired performance measurements are not desirable, we can change the configuration of *DOPS* to achieve an acceptable level of performance. We should note that the workload simulator is used beside the real system and we do not remove the real system itself. It means that we replace the real middleware by a workload simulator. Therefore, after the determination of *DOPS* configuration through workload simulator, we can use the acquired configuration for the real system.

4 Performance Evaluations and Analysis

Now we use our workload simulator to do some experimentation and evaluate the performance of *DOPS*. The goal of these experiments is to evaluate the effectiveness of the proposed techniques for sizing multiple object pools dynamically and using separate replacement strategies for each object pool. We evaluate cases with different number of object pools. However, for more readability, we just show the test cases with three object pools here.

4.1 Experimental Setup and Evaluation Criterion

We use a machine equipped with one 1.7 GHz PIV processor, 1GB of RAM. Also we use MSSQL Server 2000 as the backend DBMS running on MS windows 2000. We develop an application with *Microsoft Visual C++ 6.0* containing 20 entity classes which each of them only maps to one table in the database. Also, all of the objects instantiated from the same class have the same size. Consequently, through the interfaces offered by *DOPS*, we connect our application, which uses IONA/Orbacus 4.0.5 (a commercial implementation of CORBA) as its middleware, to *DOPS*.

4.2 Effectiveness of Using Multiple Replacement Strategies

Here we demonstrate that why using one global replacement strategy for all object pools is not a satisfactory solution. In other words, we prove that using a separate replacement strategy for each object pool can result in better throughputs. As a test case, we simulate a system containing two object pools, *Poo1* and *Pool2*, with the following *RLFs*:

- $localityPool1(\mathrm{x}) = \dfrac{(2 \times x - 1)}{N^2}$, $x \in [1, N]$

  This *RLF* is an Increasing Density Function (*IDF*). It means that the simulator makes requests on objects which are used longer time ago with higher probability.
- $localityPool2(x) = localityPool1(N - x + 1)$, $x \in [1, N]$

  This *RLF* is a Decreasing Density Function (*DDF*). It means that the simulator makes requests on objects which are used recently with higher probability.

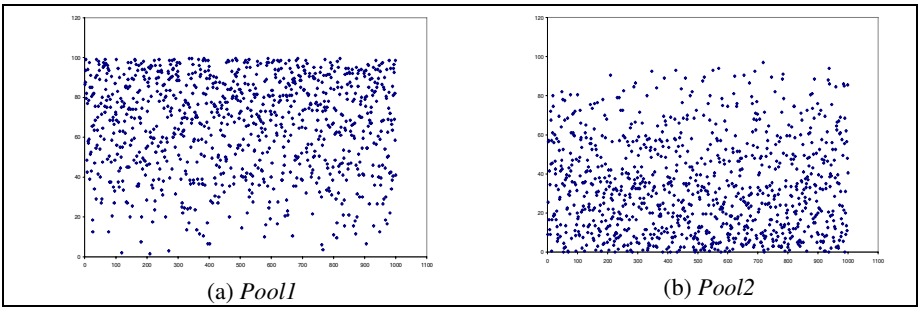


Fig. 5. Scatter presentation of random requests generated according to associated request locality functions with object pools. Diagram (a) shows the requests made on *Pool1* and Diagram (b) requests made on *Pool2*.

We suppose that the number of associated objects with each pool, which is denoted by N in equations above, is equal. Then, we configure the workload simulator to make requests on each object pool according to given request locality functions. The requests made by the simulator on both of object pools when $N = 100$ is shown in Fig. 5. In Fig. 5(a), the requests made on *Pool1* have a high density on objects which are used long time ago. Also, we can observe in Fig. 5(b) that the requests made on *Pool2* have a high density on objects used recently. Afterwards, we measure the throughput of each object pool while we associate them with two different replacement strategies: *LRU* and *MRU*. As illuminated in Fig. 6(a), the throughput of *Pool1* when it uses *LRU* is better. Also Fig. 6(b) shows that *Pool2* gains better throughput when it uses *MRU*. Hence, this test case shows that if we use separate replacement strategies, through facilities provided by *DOPS*, we acquire better throughput comparing the case when we use only one global replacement strategy.

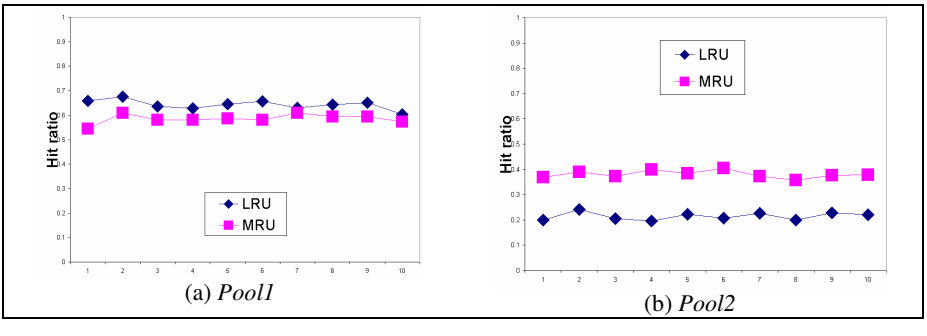


Fig. 6. Throughput of each object pool for two replacement strategies: *LRU* and *MRU*. Diagram (a) shows the hit ratios of *Pool1* and Diagram (b) the hit ratios of *Pool2*.

4.3 Evaluation of IPCM Performance

To experiment the performance of *IPCM* with respect to existing approaches, we use the workload simulator. Our test includes two experimental variables: the frequency

of callings on each pool, and the locality of incoming requests for each pool. So, we simulate a specific system with the following properties as shown in Fig. 7:

- We create three different object pools in the pool container [2].
- We determine frequency of incoming requests for each pool through a function based on time spots:
  - PTIRPool1(t) = A* (sin(t) + 1).
  - PTIRPool2(t) = B * (sin(t + pi / 2) + 1).
  - PTIRPool3(t) = C.

  In fact, we choose two types of frequency functions: sinus form (which exhibits variable frequency of incoming requests on different time spots) and constant form (which shows constant frequency of incoming requests in all time spots). There are many other types of frequency functions that can be used in this experiment. However, they are mainly combinations between sinus and constant forms. *A*, *B* and *C* are the peak of request loads assigned to the pools. From now on, we call these variables calling frequency variables. So, through the assignment of different values to calling frequency variables, we simulate different workloads on each pool.
- We generate incoming requests on each pool according to a *PDF*. Each *PDF* exhibits the probability of making a call on a specific object of a given pool after a specific time. So, we provide each pool with a separate *PDF*, which is based on *x* that shows the place of objects in a sorted list according to their last access time. This *PDF* is used to simulate different request localities for each pool:
  - localityPool1(x) = D.
  - localityPool2(x) = E.
  - localityPool3(x) = F.

  D, *E* and *F* are *PDF* type variables that can accept one of the following values: *IDF*, *DDF*, and Uniform Density Function (*UDF*). From now on, we call these variables request locality variables. So, we assign different *PDF*s to these request locality variables to simulate different request localities on object pools.

So, we break down our two experimental variables to 6 new ones (*A, B, C, D, E, and F*) introduced in the properties of the simulated system above. To compare *IPCM* to other approaches, we consider different values of these variables. Then we start the simulation to figure out the throughput of the system using each approach. Here, we compare *IPCM* to the following approaches which can be used for determination of the size of each pool:

- Static approach: in this approach, the total capacity of pool container is divided equally without caring about *activity ratio* and *hit ratio*. So, the size of each pool is calculated through the following equation:

$$sizePool_i(t) = C/N.$$

[2] We evaluated cases with different number of object pools and achieved similar results, so we just show the case for three object pools here.

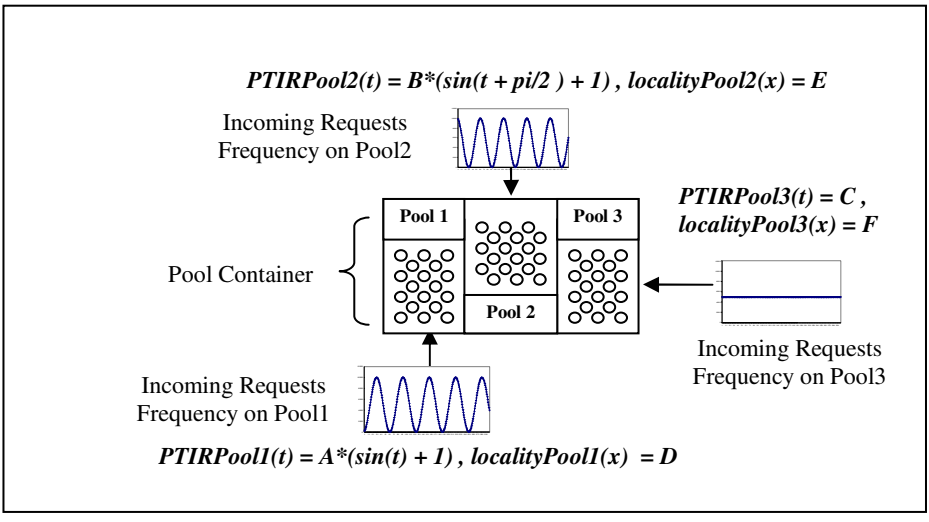


Fig. 7. The conceptual diagram of the simulated system

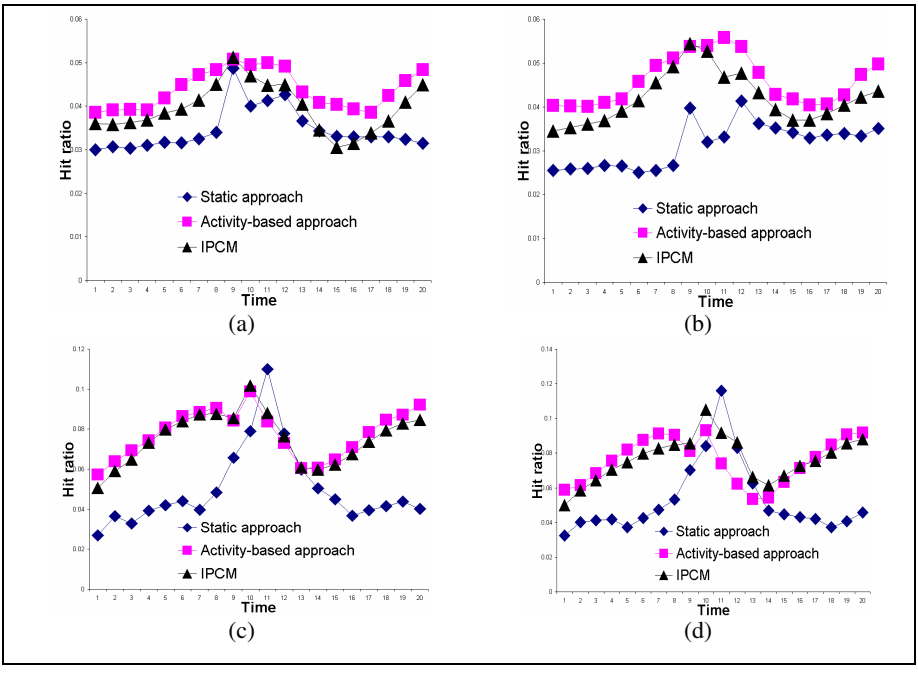


Fig. 8. Results of the experiments (independent of request locality variables). Diagrams (a) and (b) show the simulation results for larger sample variances of calling frequency variables, and diagrams (c) and (d) show the results when the sample variance is smaller.

- Activity-based approach: through this approach, the total capacity of pool container is divided according to *activity ratio* of each pool:

$$sizePool_i(t) = ARPool_i(t)/N.$$

At first, we assign *UDF* to all request locality variables. In this way, we figure out the result of comparison independently from request locality variables. The result of this phase of simulation for different values of variables *A*, *B*, and *C* is shown in Fig. 8, we show from the simulation outcomes that the larger the sample variance the better performance of *IPCM* comparing to the other approaches. As shown in Fig. 8 for all values of calling frequency variables with different sample variances, the static approach is the worst solution. On the other hand, *IPCM*'s performance is lower than activity-based approach (cf. Fig. 8(a) and Fig. 8(b)) when the sample variance of calling frequency variables is small. However, when their sample variance is larger, the performance of *IPCM* is close to activity-based approach (cf. Fig. 8(c) and Fig. 8(d)).

However, in practice we can not ignore the request locality variables. So, in the next phase of experiment we must figure out the affect of assigning different values to the request locality variables. The result of this phase of simulation is shown in Fig. 9 activity-based approach is the worst solution and the *IPCM* is the best one. However, sometimes all approaches result in close hit ratios (cf. Fig. 10(a)). In fact, *IPCM* considers the satisfaction of all object pools at the same time. Static-based approach wastes the available capacity of container because it divides the container's capacity equally. Also, activity-based approach just improves the total hit ratio while it starves object pools with low activity ratios. Hence, we offer a detailed test case to show the throughput of each approach for each object pool separately. In Fig. 10(a) the total

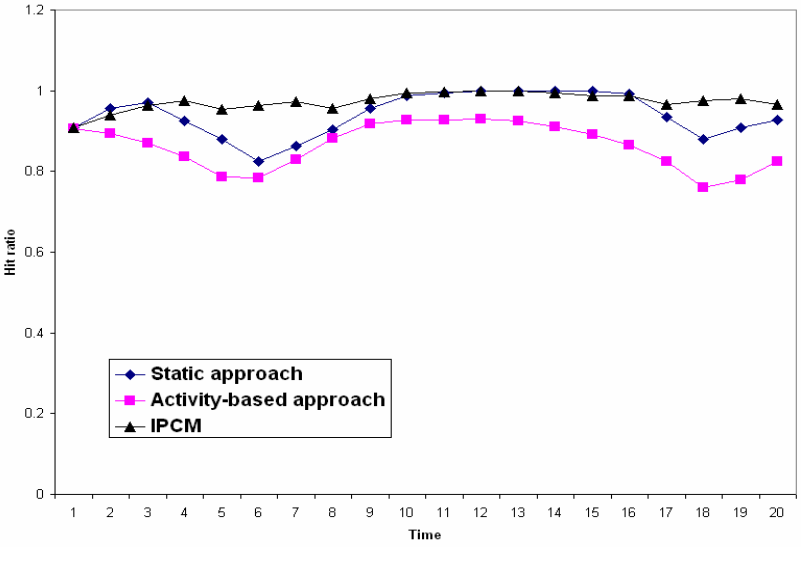


Fig. 9. When we consider the locality parameters, the activity-based approach is the worst solution and the IPCM is the best one

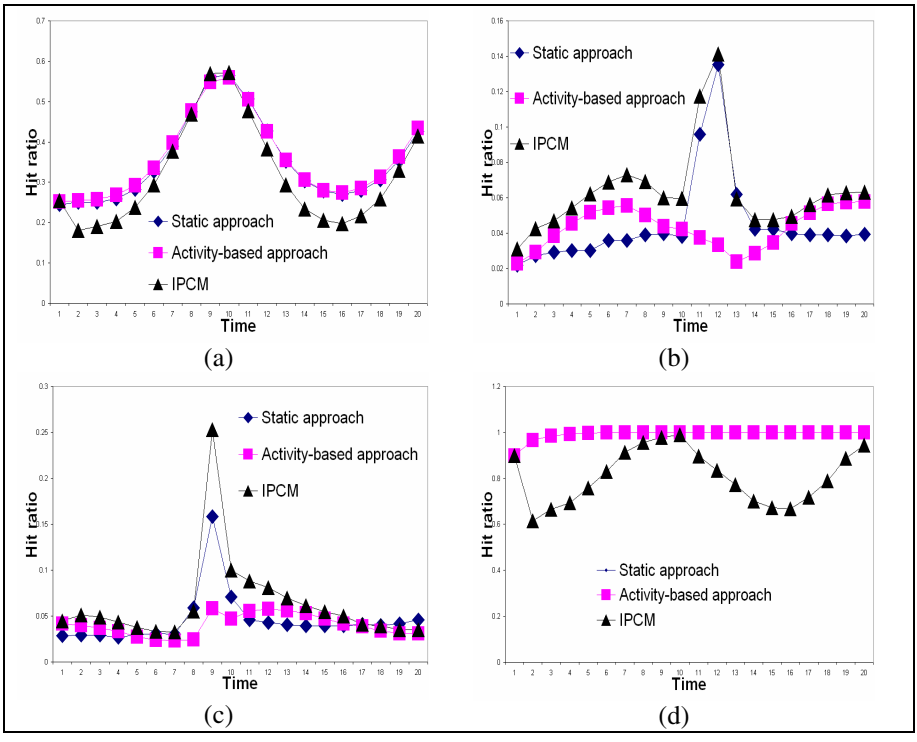


Fig. 10. Analysis of a test case with close total throughputs. Diagrams (a) shows the totals hit ratios for each approach, and diagram (b), (c) and (d) show the throughput of each pool separately.

throughput of all approaches are close together. However, *IPCM* acquires better throughputs for two object pools (cf. Fig. 10(b) and Fig. 10(c)) comparing to other approaches. While the other approaches only result in better throughput for one pool (cf. Fig. 10(d)). Generally, simulations for containers with more object pools show that *IPCM* results in better throughputs for more object pools when the total hit ratio of all approaches are close together.

According to above experiments, using *IPCM* in systems with different calling frequency functions and high probability of the locality of incoming requests is profitable. Hence, enterprise complicated software systems are appropriate cases for applying *DOPS* architecture.

5 Conclusion and Prospects

This paper has presented DOPS for object pooling in distributed object-oriented systems. It has showed that DOPS provides a flexible and reusable solution to triumph over memory management difficulties. DOPS is suitable for distributed middlewares which provide message profiling mechanisms and also use general proxies to access concrete objects existing in the server side.

On focus of our future work is to making an integrated approach for object pool configuration. Using a central configuration repository, we can isolate the client code from the configuration mechanisms. Also the capability of using intelligent methods will be provided. For example, machine learning techniques can be used to find optimum configurations transparently and automatically. Other research directions include adding fault tolerance to DOPS through replication mechanisms. Concerning the different degrees of backup pools readiness [4], an applicable pattern will be provided.

Another interesting issue to be addressed is applying concurrency controls on DOPS. DOPS builds object pools in the memory and separates the objects from the underlying transactional storages. This causes the concurrent access to shared objects without any concurrency management. So, ACID (Atomicity, Consistency, Isolation, Durability) [8] properties of transactions, especially consistency and integrity, will be missed. Consequently, providing locking mechanisms for DOPS is mandatory.

Also DOPS can be used by agent-oriented middlewares [9] to manage the lifecycle of agents. However, yet there is no operational agent-oriented middlewares having proper message profiling facilities.

References

1. M. Kircher, Lazy Acquisition Pattern, Proceedings of the European Pattern Language of Programs Conference, Kloster Irsee, Germany, July 4-8, 2001
2. P. Jain and M. Kircher, Leasing Pattern, Proceedings of the Pattern Language of Programs Conference, Allerton Park, Illinois, USA, August 13-16, 2000
3. M. Grand, Patterns in Java - Object Pool, John Wiley & Sons,1998
4. D. Slama, J. Garbis and P. Russell, Enterprise CORBA, Prentice Hall, 1999
5. P. Jain, Evictor Pattern, Proceedings of the Pattern Language of Programs Conference, Allerton Park, Illinois, USA, 2002
6. D. Salomon, Data Compression: The Complete Reference, Springer, 2004
7. A. Tanenbaum, Modern Operating Systems, Prentice Hall, 2001
8. J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993
9. A. Omicini, G. Rimassa, Towards Seamless Agent Middleware, Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004
10. Kronmal RA, Peterson AV Jr (1979a), On the Alias Method for Generating Random Variables from a Discrete Distribution, American Statistician, Vol. 33, No. 4, pp. 214-218
11. Oracle JOC 10.1.2 Tutorial, Oracle Technology Network, 2004
12. B. Ban and B. Wang, JBossCache Reference Manual V. 1.2, JBoss Inc., 2005
13. JCS v.1.2.7.3 Project Documentation, Apache Software Foundation, 2006
14. L. Cherkasova, Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy, In HP Technical Report, Palo Alto, November 1998.
15. D I Bevan, Distributed garbage collection using reference counting, Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe, p.176-187, March 1987

A Java Framework for Building and Integrating Runtime Module Systems

Olivier Gruber[1] and Richard S. Hall[2]

[1] IBM Watson Research Center, Hawthorne, NY, USA
ogruber@us.ibm.com
[2] Laboratoire LSR, Grenoble University, Grenoble, France
richard.hall@imag.fr

Abstract. We present the design of core mechanisms for building interoperable runtime module systems. Runtime modules are the cornerstone of component platforms, such as Eclipse Rich Client Platform or Apache Geronimo, since they provide components with their namespace for Java classes and resources. Unfortunately, much diversity exists today regarding module systems, thereby hindering the creation of a global Java ecosystem for components. Our approach proposes a first step towards a solution since our design can model and integrate different module systems such as traditional Java class loaders, Geronimo Beans, or the OSGi framework. We propose low-level core mechanisms, both modular and flexible, that enables building such module systems from reusable and adaptable mechanisms. Furthermore, once module systems are built with our core mechanisms, our approach enables their integration within a single runtime environment. This is especially important at a time where different open-source component platforms, as well as the products built upon them, are facing the integration challenges between heterogeneous module systems. Our design is the basis of the current implementation of Felix, within the Apache community; it illustrates an integration path towards a global Java ecosystem for components.

1 Introduction

In recent years, there has been growing interest in runtime module systems within the Java community [10,11]. Java has a simplistic built-in module system through class loaders. To support other runtime components and services, different Java communities have proposed, above class loaders, more advanced runtime module systems such as GBeans [5] or OSGi technology [1].

It appears there is an overall agreement that a runtime module is a namespace for Java types and resources. Furthermore, there is usually a delegation model where modules may delegate to other modules for loading types or resources. However, there is almost no agreement on the details of runtime modules, such as the granularity of delegation (e.g., Java types, packages, or entire modules), encapsulation capabilities or the flexibility of dependencies. We also find very

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1772–1789, 2006.
© Springer-Verlag Berlin Heidelberg 2006

different physical packaging of modules, including download formats and local storage formats.

While being fundamentally similar in concepts, these module systems differ and thereby fracture the overall Java community, hindering the creation of a global ecosystem for Java components. This translates into two major challenges. The first challenge is the integration of these various module systems in a single runtime environment. Indeed, an ecosystem delivers its promises when one can freely assemble components, as one sees fit, regardless of their origin. This suggests that different module systems can be installed in a single runtime environment and offer an efficient integration.

The second challenge is promoting sharing, reuse, and overall synergy across module systems. Indeed, building a complete module system from scratch is a difficult task; equally difficult is the evolution of such a module system through its releases as its functionality matures and naturally evolves. This suggests that we apply modularity and its corresponding separation of concerns to the architecture and design of the module systems themselves, building them from low-level well-defined concepts that cleanly separate mechanisms from policies.

Our work addresses these two challenges through a comprehensive Java framework for building and integrating runtime module systems, which work continues previous work in this area [8,7]. Our work equally applies to existing and new module systems. Existing module systems may be integrated through our approach within a single Java runtime environment. Our work defines the interoperability interfaces that each module system must implement. This can be done with minimal impacts on their current implementations and designs. Moreover, our work helps designers and architects build new module systems. It provides a blueprint for the architecture and design of module systems that can be more easily built and understood. First, module systems are designed in terms of the same core concepts, thereby promoting reuse. Second, our clean separation between mechanisms and policies promotes the creation of common and reusable policies. We claim that this results in specific module systems that can be developed and evolved by simply reusing or customizing existing policies or mechanisms.

This framework is the result of years of experience within various communities (OSGi, Eclipse, Felix and Geronimo) and their corresponding approaches to module systems. This work is driving the design and architecture of Felix, within the Apache community. We have experienced the flexibility and adaptability of the approach through the various releases of the OSGi Technology. We also realized that these mechanisms could also be used to implement a large number of existing module semantics, including GBeans or XBean from Geronimo. We further realized that with minor modifications to the current Felix design, we could enable interoperability between heterogeneous module systems.

This paper is structured as follows. In Section 2, we introduce the overall architecture of our approach for runtime modules. In Section 3, we discuss our framework from the perspective of integrating different module systems in a

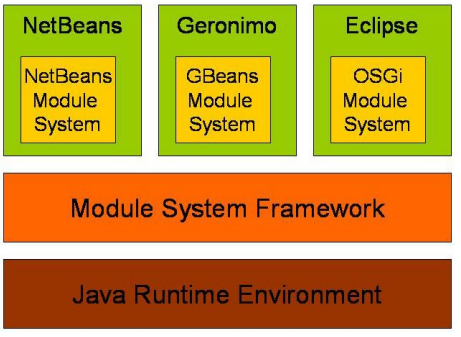


Fig. 1. Integrated Module Systems

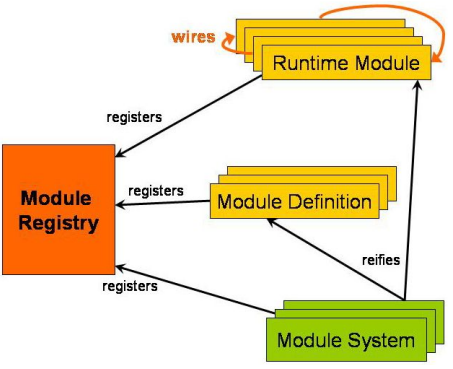


Fig. 2. Our Framework Design

single Java Runtime Environment. In Section 4, we focus in contrast on the building of a module system using our framework. In Section 5, we illustrate our approach with concrete examples of known module systems. In Section 6, we conclude.

2 Overall Architecture

Our focus is on the design of a low-level framework to be integrated in a Java Runtime Environment. The framework is aimed at runtime platform architects and designers, helping them build and integrate various module systems, as depicted in Figure 1. The rationale for such an integration is a global ecosystem for Java components, across different module systems. In particular, we feel important that one can pick and choose between components from the successful Java platforms such as NetBeans [9], Geronimo [2] or Eclipse [6].

We approach this integration challenge from a foundation perspective. We didn't want a veneer-like integration with its performance drawbacks. We wanted to provide a low-level framework that would provide suited core mechanisms. As a primary goal, we wanted existing module systems to be able to integrate with minimal impacts on their current implementations. As a secondary goal, we wanted to provide mechanisms for helping with the internals of module systems, offering well-defined concepts that would promote sharing and reuse of both mechanisms and policies.

The integration approach we chose requires each module system that wants to integrate to follow our overall architecture and design, as depicted in Figure 2. Our design introduces three core concepts: module systems, module definitions and runtime modules; all three defined as Java interfaces. Each integrated module system has to implement these interfaces, in its own specific way. For example, this means that the implementations of Geronimo and Eclipse have to be modified; they have to provide implementations for our interfaces but this is

quite straightforward to put in place as we will illustrate in Section 5. Furthermore, the impacts are solely about the internals of module systems; there are no visible impacts on the component models that module systems propose to their developer communities.

At the heart of the impacts is the integration with the module registry. The module registry (*IModuleRegistry* interface) is a shared registry, one per Java Runtime Environment (JRE). Each module system (*IModuleSystem* interface) must register with the registry. Furthermore, each module system registers module definitions with the registry. A module definition (*IModuleDefinition* interface) is an abstract description of a version of a module, identified by a module name and version. Module definitions may be registered or unregistered to the registry at any time.

Each module system acts as a factory for the module definitions it registered. In other words, a module system reifies module definitions into actual runtime modules. A runtime module (*IModule* interface) is a namespace for accessing classes and resources. The module system that creates a runtime module is called its *owner*. We do not impose any restriction on this reification process. In most environments, a module definition will be instantiated only once, bringing in memory only one version of a module. Other environments will need to be able to load different versions of certain modules. In rare situations, one may want to instantiate one definition multiple times, the main reason for doing so being to avoid sharing of static fields in Java classes. Our framework allows it, but we discourage it.

As a central actor, the component registry coordinates the integration of the various module systems. First, the registry gives a place to find existing runtime modules, which promotes sharing these modules amongst module systems. Here follows a typical step-by-step process for resolving a dependency on a module. A module system will look in the registry for a suitable module. If it finds one, it reuses that module, effectively sharing it. If it does not find a suitable one, it will search for a suitable module definition in the registry. If it finds one, it asks the registry to create a module from that definition. The registry will relay that instantiation request to the module system that registered that particular definition. If no suitable definition is found, one can fail or create one or download one from the network.

Second, the registry is the guardian of the overall lifecycle of modules. A runtime module goes through two states: *unresolved* and *resolved*. A module is resolved when it has all its dependencies satisfied. Dependencies among modules are reified by wires (*IWire*), which materializes delegation among modules. The creation of wires, called the resolution process, is a two-level recursive process. One level is local to each module system and it involves a specific resolving algorithm in an homogeneous world, among modules instantiated by this module system. Each module system will have its own approach for this first level. The second level however is the coordination among module systems and is the responsibility of the module registry.

3 Integrating Module Systems

This section focuses on the integration of module systems through our module registry. We discuss the details of module definitions, modules and wires. We also discuss the details of the module resolution process.

The module registry, see Listing 1.1, is a mechanism for integration that we can characterize in different dimensions. The first dimension is a registry of module systems, where installed module systems are registered with a unique key. It is important for the registry to track installed module systems to coordinate them during the resolution process for created runtime modules. The second dimension is a registry of known module definitions, registered by module systems. Notice that when a module definition is added to the registry, it is associated with the module system registering it, called its *owner*. For each module definition, the registry will use its owner as a factory to create runtime modules from that module definition. The third dimension is the coordinator of the overall resolution process across module systems where it is decided if modules can be resolved and corresponding wires created.

Listing 1.1. Module Registry

```
public interface IModuleRegistry {

  boolean addModule(IModuleDefinition md,IModuleSystem ms);
  boolean removeModule(IModuleDefinition md);
  IModuleSystem getOwner(IModuleDefinition md);

  IModuleDefinition[] find(String name);
  IModuleDefinition[] find(String name, String version);

  IModule makeInstance(IModuleDefinition md, boolean isResolving);
  void destroyInstance(IModule module);

  void resolve(IModule module) throws ResolveException;
  boolean isResolving(IModule module);

  IModule[] getModules();
  IModule[] getModules(String name);
  IModule[] getModules(String name, Version version);
  IModule getModule(String instanceId);

  IModuleSystem getModuleSystem(String key);
  void addModuleSystem(String key, IModuleSystem framework);
  void removeModuleSystem(String key);

  void addListener(IRegistryListener listener);
  void removeListener(IRegistryListener listener);
}
```

A module definition (*IModuleDefinition*, see Listing 1.2) describes one version of a given module; it is therefore identified by the name and version of that module. It is an abstract description of that module version, each module system would define more concrete representations, providing access to the full details of the module description.

The module description reified as module definitions allows different module systems to locate desired module definitions. This process is most often based on the module name and version. It may, however, also use the imports and exports, a common concept to all module descriptions we know: imports describe what a

Listing 1.2. Module Definition

```
public interface IModuleDefinition {

  String   getName();
  Version  getVersion();
  Map      getAttributes();

  Object[] getImports();
  Object[] getExports();

  IModuleDefinition getSubmodule(String name);
}
```

module will need to be resolved; exports describe what a module will offer once resolved.

Module definitions, however, only provide abstract access to imports and exports. Using imports and exports to locate a specific module definition requires more concrete understanding of these imports or exports. We are not specifying how this negotiation happens. It can be a negotiation with the owner of the module definition, obtained from *IModuleRegistry.getOwner*. Knowing the owner and therefore its unique key may be enough to know the actual format of imports and exports, which enables downcasting *IModuleDefinition* into a more concrete interface with more information. This is not about downcasting to implementation specific interfaces but rather suggests that families of module systems will specify more concrete models about imports and exports. We would expect one such specification for OSGi and may be a few others for simpler models such as GBeans or XBean.

For example, assume we are looking at fine-grain dependencies on Java packages. In other words, one is not looking for a specific module, but he is rather looking for a particular Java package exported by any module offering it. One would therefore need to look at the exports of all module definitions it can understand. This often returns several possible candidate definitions, a choice must be made. Looking at imports may be useful in this case as it might help selecting among the candidate definitions. For instance, one would prefer a module definition with resolvable imports over one that is not resolvable in the current configuration.

3.1 Modules and Wires

A module instance (*IModule*) is an actual runtime concept, providing a namespace for classes and resources. For Java applications, it provides a way to find classes or access resources (see methods *IModule.getClass* and *IModule.getResource*). In its simplest form, a runtime module loads classes and resources solely from local content, which is the role of the content loader (*IContentLoader*). Content loaders reify classes and resources; they are discussed in details later in Section 4. More complex modules depend on each other, with various granularity for the delegation of class or resource loading among modules. Such delegation is described through imports and exports; it is reified through wires. For module and wire interfaces, see Listing 1.3.

Our wire mechanism supports delegation at different granularity. At the very least, we promote module-level delegation where modules can create wires for exporting themselves. Such a wire is created through the *IModule.createWire* method, passing only an importer module. For finer-grain wires, one would use *IModule.createWire* method, passing both an importer module and an export object. This results in a wire for that particular finer-grain export. Any created wire may be dropped by its importer later on through the *IModule.droppedWire* method.

Note that the wire is created by the exporting module and used by the importing module, not the other way around. The rationale is that an export has its semantics defined by the exporting module, not the importing module. In simple terms, the importer needs what the exporter offers, the exporter decides what that is.

Listing 1.3. Module Wire

```
public interface IModule {

  String getInstanceId();
  IModuleRegistry getRegistry();

  IModuleDefinition getDefinition();
  IContentLoader    getContentLoader();

  IWire[] getUsedBy();
  IWire[] getUses();

  IWire createWire(IModule importer);
  IWire createWire(IModule importer, Object export);
  void  droppedWire(IWire wire);

  boolean isResolved();

  Class getClass(String name) throws ClassNotFoundException;
  URL getResource(String name) throws ResourceNotFoundException;
}

public interface IWire {

  IModule getImporter();
  IModule getExporter();

  Object getExport();

  Class getClass(String name) throws ClassNotFoundException;
  URL getResource(String name) throws ResourceNotFoundException;
}
```

Further note that finer-grain wires do require from importing module systems a concrete understanding of exports. Given an importer module, its owner has to be able to match imports to exports; exports potentially proposed from runtime modules from other module systems. As illustrated before, this matching goes beyond the simpler matching on module names and versions. It requires downcasting exports and potentially imports to more concrete types, since the methods *getExports* and *getImports* on *IModuleDefinition* are returning untyped objects.

For example, assume we want to integrate a module system with the OSGi module system that provides both module-level and package-level exports. The OSGi specification defines the exact syntax of exports, so this suggests that exports would be corresponding strings at the module definition level. Any module system understanding that specified syntax could look at the exports and decide

which one is necessary to satisfy a given import, asking it to create a wire to that particular export.

This OSGi example also illustrate why the wires are created by exporters and not importers. In OSGi, an exported Java package may be associated with a class-level filter, allowing it to control which classes are exported and which are not. This filtering role is achieved by the wire. Note that this does not apply solely for finer-grain exports, but could also be used to control, in a similar spirit, what is exported at the module level. Indeed, a module may want to filter what packages or classes it exports as a whole.

3.2 Resolving Modules

The resolution of modules is the process of finding matching exports for imports. This resolution process is a shared responsibility between the module registry and the installed module systems. For a module to be resolved, it needs all its imports wired to exports of resolved modules. This is therefore a recursive process since module definitions, along with their imports and exports, describe a directed graph of dependencies, with potential cycles.

The resolve process starts with attempting to resolve one module, calling the *IModuleRegistry.resolve* method for that given module. This starts the resolve process as a two-phase protocol between the registry and installed module systems. The first phase of the resolution process is called the *prepare phase*. When the registry is asked to resolve a module, it forwards the request to the owner of that module through calling *IModuleSystem.prepareResolve*, see Listing 1.4.

Listing 1.4. Module System

```
public interface IModuleSystem {

  IModule makeInstance(IModuleDefinition md);

  boolean prepareResolve(IModule module);
  void completeResolve(boolean result);
}
```

Resolving that module usually depends on other modules being resolved as well, because of imports. The owner needs to match these imports to acceptable exports. A given import may match many exports from different modules, potentially from different module systems. Each module system will implement its own policy for matching imports to available exports. This matching process may yield several candidate exports per import; one candidate must be selected. Such selected exports must have their module resolved so that a wire can be created. If any such modules are not resolved, the process recurses through calling *IModuleRegistry.resolve*.

Note that we do not impose any selection policy, each module system is entirely free to adopt its own approach. It can use a constraint solver, often called a resolver, like in most OSGi implementation such as Eclipse or Felix. It can also use some simpler wiring logic, either expressed as scripts or metadata. Usually, policies will favor selecting exports from already resolved modules. If no resolved

module is available, policies will often favor trying to resolve existing modules rather than creating new ones from module definitions. But again, those are only best practices and module systems are free to do as they see fit.

During the prepare phase of the resolution process, we potentially face cycles among modules. Cycles are detected through keeping track of whether module instances are resolving or not. This happens in the registry. Whenever the registry is asked to resolve a module instance, it sets its state to resolving. The state can be queried through *IModuleRegistry.isResolving*. To break cycles, the registry optimistically answers that a module instance is resolved if it is already resolving. At some later point, the cycle unwinds and a real decision is made. The decision is returned by the original call to the *IModuleSystem.prepareResolve* method and it is remembered by the registry.

Once the prepare phase ends, a global decision is made for the second phase, called the *completion phase*. The completion phase is linear; the registry sequentially invokes, through *IModuleSystem.completeResolve*, all module systems that were involved in the prepare phase. This call carries the global decision: *commit* or *abort*. The global decision is an abort if any module needing to be resolved could not be resolved. The final decision is a commit if all modules needing to be resolved could be resolved. This means that all module instances being resolved must be set to a state of resolved.

In other words, the entire resolution process is atomic, it entirely fails or entirely succeeds. Moreover, it is isolated; the registry ensures that there is only one such process happening at any given time. When the process commits, this also means that wires may have to be created for the newly resolved module instances, materializing the delegation on selected exports during the prepare phase. If wires were eagerly created by some module systems during that first phase, such module systems will need to drop the created wires if the global decision is an abort.

3.3 Registry Events

We wanted to support both static and dynamic environments, this means that both module systems and modules can be added and removed dynamically. This suggests that the registry offers an event framework for dispatching what is happening within the registry. Dispatching events on listeners, see Listing 1.5 is important because it avoids regularly polling the registry.

Listing 1.5. Registry Listener

```
public interface IRegistryListener {

  void notifyAdd(IModuleDefinition md);
  void notifyRemove(IModuleDefinition md);

  void notifyAdd(IModuleSystem ms);
  void notifyRemove(IModuleSystem ms);

  void notifyAdd(IModule module);
  void notifyRemove(IModule module);

  void notifyResolve(IModule module);
  void notifyUnresolve(IModule module);
}
```

This is especially important for dynamic module systems that want a dynamic approach to resolving modules. In a dynamic environment, modules may become resolved at any point in time and newly resolved modules could allow other modules to resolve. Also, this allows module systems to know about newly created modules, which could also be used to resolve pending unresolved modules.

4 Building Module Systems

After focusing on the integration challenge of different module systems, we will focus on the building challenge of such module systems. We are proposing low-level mechanisms that cleanly separate the different concerns at play when building module systems. These mechanisms are optional for module system designers, although these mechanisms are designed for faster development, higher reuse, and easier adaptation.

These mechanisms are about the internals of the previously introduced concept: the module. There are four identified facets to a module implementation: interface to access classes and resources; content loading and management; delegation among modules; URL creation for resources.

One can see these four facets in the traditional framework for Java class loaders. For most Java code, a class loader is nothing more than an interface for accessing classes and resources. Internally, class loaders reify classes from external representations such as class files. One essential aspect of class loaders is their ability to understand an actual content management such as a classpath based on directories, JAR files, or URLs. One aspect of this is the ability to find class files, the other is the ability to create URLs for identifying resources. Finally, class loaders offer a simple parent-based delegation for class and resource loading.

Unfortunately, class loaders mix these concerns into a rather simplistic and monolithic module system. Our design aims to separate these different concerns. The *IModule* interface provides access to classes and resources through *IModule.getClass* and *IModule.getResource*. The content loading and management is addressed through content loaders (*IContentLoader*). The delegation among modules is addressed through search policies (*ISearchPolicy*). The URL creation for resources is addressed through URL policies (*IURLPolicy*). Each of these mechanisms will be discussed in turn in the following subsections.

4.1 Content Loaders

The content loader is the entity that knows how to reify Java types and access resources from the actual module content. For classes, the content loader is responsible for accessing an external representation of classes and internally calling some JVM-specific interface to create Class objects. For resources, the content loader is responsible for finding available resources and returning URL objects identifying them.

The content loader interface and its policies are introduced in Listing 1.6. We will discuss the search and URL policies in following subsections. We focus here

on the role of the content loader as a bridge between an external representation of module content and the runtime access to Java types and resources.

Regarding Java types, a content loader knows how to find external representations of Java types and reify them using some JVM-specific interface. There is some heterogeneity among JVMs. Traditional Java offers only one mechanism to reify a class: the *ClassLoader.defineClass* method. This means that a content loader would have to use a class loader internally. The IBM J9 virtual machine offers specific support for the JXE format. The JXE format is the result of ahead-of-time loading and compilation of class files into a memory segment that can be mapped by the JVM in one operation.

Listing 1.6. Content Loader and Policies

```
public interface IContentLoader {

  void          setSearchPolicy(ISearchPolicy policy);
  ISearchPolicy getSearchPolicy();

  void       setURLPolicy(IURLPolicy policy);
  IURLPolicy getURLPolicy();

  Class       getClass(String name);
  URL         getResource(String name);
  InputStream getEntryAsStream(String name);
}

public interface ISearchPolicy {

  IModule getModule();
  Class findClass(String name) throws ClassNotFoundException;
  URL findResource(String name) throws ResourceNotFoundException;
  String findLibrary(String name);
}

public interface IURLPolicy {
  URL createURL(String path);
}
```

This separation of concerns is powerful. It is now easy for a module system to adapt to different JVMs. For instance, on J9 virtual machines, a module system can easily leverage the high performance of the JXE format. It just needs a different content loader, but the rest of the module system remains unaffected. But even more traditional formats for module contents give rise to various challenges that content loaders help address.

With modules as JAR files, native libraries and nested JAR files are an issue. Native libraries require special treatment since most operating systems cannot load libraries from JAR files. Nested JAR files also require special treatment because standard class loaders are not able to load classes from nested JARs[1].

The role of the content loader is to insulate the rest of the module system from these variations in physical layouts. This is important because many module systems simply differ through their physical layout. This helps also with evolution, like for example with the Eclipse case moving from simple JAR files to nested JAR files. Originally, Eclipse always stored plug-ins as exploded directories, but class files were provided as JAR files. In later version, Eclipse has been keeping plug-ins as JAR files in the file system, with nested JAR files for class files.

[1] E.g., nested JAR files as used by the Eclipse platform.

Moreover, content loaders simplify adaptability. For instance, different content loaders may provide different caching policies. For instance, one content loader may be extracting the embedded JARs into a local cache. Another one may be extracting single classes at a time. The decision may be different if the JAR files are local or are accessed through remote URLs. The caching policy might also be different if the JAR files are in a standard file system or in flash or any low-bandwidth device.

4.2 Search Policy

The search policy is the policy behind where to look for a Java type or a resource. Most modules have local content and they have wires to other modules. The search policy decides the order between wires and the local content for looking for a Java type or a resource. The rationale is that search policies are a dimension of much diversity among module systems. It is therefore interesting to have this policy isolated into its own pluggable object. Furthermore, one can hope that reusable but parameterized search policies will increasingly be used.

We do not impose a search policy, which is an important part of a dependency model. For instance, we have implemented different dependency models, such as OSGi R3 (package-level dependencies) and R4 (more expressive dependencies and module-level dependencies), yielding different search policies.

To understand how a search policy works, we need to look at the two paths leading to the loading of a class in Java. One path is a direct call to the method *IModule.getClass()*. This is typically the case when some Java code wants to directly load a class. The module directly forwards the request to its search policy through *ISearchPolicy.findClass*. The search policy will determine where the class should come from. If the class should come from the local content, the request is passed to the content loader (*IContentLoader.getClass*). If the class should come from another module, through a wire, the request is forwarded to that module. It may be the case that the request may need to be forwarded to different modules through different wires.

The other path is when the Java virtual machine needs a class. For clarity, we are assuming standard Java and the internal use of class loaders for implementing content loaders. So each content loader has its own class loader, recipient of the JVM upcalls for needed classes through (*ClassLoader.loadClass()*). For each upcall, the content loader checks if it has already loaded the requested class. If it has, it returns it; if not, it delegates to the search policy of its module through *ISearchPolicy.findClass()*. Then, we are back to the first case discussed above.

4.3 URL Policy

The notion of URL policy comes down to the creation of URL objects for resources in module content. The content loader does not have enough semantics or knowledge for creating such URL objects. Indeed, it is often the case that the syntax of these URLs are specific to module systems, such as using a specific protocol. Furthermore, they may include module-level information such as

the module name or version. In order to preserve the ability to define reusable generic content loaders, we introduced the URL policy.

The URL policy is usually implemented in the module system and most often as part of the implementation of runtime modules. The content loader uses the URL policy to upcall with the path of the resource in the local content. Using that path as a black box and the content loader, the module instance may create a meaningful URL that will allow later access the resource as an input stream (see *IContentLoader.getResouceAsStream*).

5 Examples

This section discusses several different existing model systems, namely GBeans, XBean and OSGi. We will discuss how one could build and integrate these module systems using our approach. Taken together, they cover a wide range of semantics, which illustrates well the flexibility of our approach as core mechanisms for building and integrating heterogeneous module systems.

For each case, we will first discuss how integration is achieved, within our framework, which allows the module systems to globally integrate in a single Java Runtime Environment. This requires minimal changes to each existing module systems. We will then discuss how our concepts can help restructure their implementations. On that last point, our intent is not as much to tell the corresponding communities how to build their module systems than it is to offer adequate mechanisms to build new module systems and offer a synergy path for existing ones, if the communities are interested.

5.1 Discussing GBeans

GBeans is developed in the Geronimo project, the open-source J2EE-certified Web Application Server from the Apache community. It uses a module layer based on a tree of modules, faithful to the traditional Java class loading architecture. The deployment unit is a configuration, which is described by an XML file. From a module perspective, a configuration is one module. It has a classpath expressed as a list of JAR files containing classes and resources. It also contains the name of a parent configuration, expressing parent-child dependencies at a module-level granularity.

It would be easy to add to the current implementation of GBeans the concepts of *IModuleSystem* with *IModule* and *IModuleDefinition*. These are straightforward interfaces to provide on the current implementation of GBeans. For instance, a module definition can easily be provided from a GBeans configuration. It is trivial for the names and versions. For imports and exports, the approach solely supports module-level dependencies, shaped as a tree of modules. So a module definition would export itself and would import its parent definition. Furthermore, the module concept can also easily be provided on top of the GBeans class loaders.

A next step would be the integration of the overall module system with our registry. Registering the GBeans framework as a module system is also straight-

forward as is the registration of modules. The global resolution process for modules would also be straightforward given the tree-based dependencies. In GBeans, a module is resolved if its parent module is resolved.

This is all that is necessary to integrate the GBeans framework. One will notice that the overall design and implementation of the GBeans framework does not need to change. The local management of configurations, their downloads, or even the current service layer with dependency injection, above the module layer, do not need to change at all. Optionally, the GBeans framework could benefit from adopting some of the internal concepts we are proposing such as content loaders and search policies. Again, the current design and implementation is quite close to our concepts, but no sharing and reuse can be achieved because of a monolithic and proprietary implementation.

Regarding the search policy, this is straightforward since the model is a tree of modules. The search policy delegates to the parent module before it attempts to find a class or a resource locally. This adheres to the traditional semantics of Java class loaders with a parent delegation. For the content loader, one only needs to encapsulate in a content loader the current GBeans class loading scheme based on a *URLClassLoader*. The design goal would be to make both the policy and content loader reusable across different module systems.

5.2 Discussing XBean

The new release of GBeans, called XBean [4], represents an evolution in several directions, based on the Geronimo community feedback. We will discuss here only the evolution regarding the module system underlying XBean[2]. The module system seems to evolve towards a Directed Acyclic Graph (DAG) for module-level dependencies. Moreover, the module system also seems to be moving towards using the Maven repository [3] for managing JAR files upon which classpaths of modules are expressed.

Not much needs to be changed from the above discussion about GBeans in order to accommodate this evolution. In fact, the current kernel of XBean clearly separates the class loader level from the actual management of services. We entirely support this clean separation of concerns between the module and service layers. Regarding integration, the approach is as straightforward as it was for GBeans, maybe even simpler because of this clean separation between module and service layers.

The only point to consider is the resolution process. Moving from a tree to a DAG is inherently a more complex resolution process; this has nothing to do with our approach or integration with our framework. Fortunately, because dependencies are still on modules, through explicit names and versions, the resolution process remains a simple recursive one. Two small points deserve mention. Module definitions will have to reflect this with multiple imports and exports that remain at a module level. Although it is unclear if XBean will allow multiple versions of the same module to be loaded, once integrated in a general context,

[2] Personal communications with Dain Sundstrom.

XBean will have to deal with other module systems potentially allowing it. Nevertheless, the selection process remains simple, selecting among different versions of modules.

Beyond integration, the evolution from GBeans to XBean is a perfect example of the power of separating concerns and using clearly defined core concepts. From an implementation of GBeans using our concepts of search policy and content loaders, moving to XBean would be straightforward with much reuse. In particular, the content loaders can be reused since XBean still uses JAR files. Using Maven simply impacts how JAR files are obtained. We would simply have different content loaders, as reusable objects, for loading classes and resources from different layouts in the file system.

The most impact is in regards to the search policy, which would be encapsulated by the search policy interface. Since we are moving from a simple tree to a directed acyclic graph, modules may have more than one wire to the different modules they depend on. Hence, the search policy will delegate first to the imported modules, using wires in any order. If nothing is found through the wires, the search policy will ask the content loader. This may be considered a performance problem because it yields a depth-first search across the DAG for each class loading.

To solve this problem, one could adopt the same optimization as used in Eclipse. The optimization consists of building a package-level index at resolution time. At resolution time, once a module has its wires, it is easy to do one depth-first search in order to collect which Java packages are exported by which modules. The optimization comes from doing it once when a module is resolved as opposed to every time a new class is needed. This results in each module having an index allowing class loading to only search a class in the transitively wired modules that export the package of that class. In our framework, this can easily be done through extending the *IWire* interface to support this index-building approach.

5.3 Discussing OSGi

OSGi Technology, from the OSGi Alliance, defines an advanced service platform with one of the more advanced module systems we know of. Eclipse used to have a similar module system to what XBean is considering. Since its 3.0 release, Eclipse has adopted OSGi Technology which has considerably helped maturing it to its current definition.

For OSGi Technology, a deployment unit is a bundle, which is a JAR file with a manifest containing module metadata. At runtime, a bundle is reified as a module. The OSGi R4 specification defines two levels of granularity for dependencies among modules: Java packages or modules. Furthermore, OSGi Technology provides the full spectrum from completely flexible dependencies to as much control as necessary. This is achieved through metadata describing exports with dictionaries of name-value pairs and approaching imports as quite generic filters on these dictionaries.

This results in a resolution process that is fundamentally a back-tracking constraint resolver. The Equinox project, from Eclipse, has implemented one

in the open source world. That resolver is independent from Eclipse and can be reused elsewhere. The resolver models modules as resources exporting or importing capabilities. The resolver matches imports and exports and determines if modules are resolved or not. Such an approach can be adapted to our context, where certain modules belong to other module systems.

There would be no major difficulty either for the Eclipse implementation to adopt the concepts of modules and module definitions, interacting with our registry. Regarding the internals of Eclipse modules, we feel that our concepts of search policy and content loaders would have tremendously helped the Eclipse evolutions through the last years, from its release 2.x to its current releases 3.x. This belief has many dimensions and is rooted in our dual involvement in the Equinox project in Eclipse and our participation in expert groups in the OSGi Alliance.

The use of a search policy would have helped evolve the model for module dependencies. As we already mentioned, Eclipse 2.x had a proprietary module systems based on a directed acyclic graph, like the one we discussed for the potential future of XBean. When adopting OSGi as its foundation, Eclipse had to integrate package-level dependencies as well. The search policy would have encapsulated such changes.

Our approach with a more explicit resolution process would also have helped. The move to OSGi strongly suggested moving to a constraint solver, dealing with both granularities of dependencies. Within our framework, the presence of such a resolution process is well identified as a recursive process through our registry. Again, it would have provided an architecture to guide the changes.

On another dimension, one early tension about adopting OSGi Technology in Eclipse was around the physical format of module contents. Most OSGi implementations were promoting the use of JAR files for module contents, keeping JAR files locally on the file system. Eclipse was promoting exploded JAR files, as a local management of contents. Furthermore, Eclipse was offering URL-based access to the contents of modules. This tension, seemingly easy to resolve, was at the origin of much discussions and much re-engineering because the differences were felt as profound because they were deeply rooted in rather monolithic implementations.

With our approach, the concepts of content loaders and contents would have showed simple points of flexibility and adaptability. It is fairly easy to have both content loaders on JAR files and exploded JAR files. This is exactly the purpose of content loaders, to encapsulate variations on physical management of content, including different approaches to class paths in these different content formats. The content, with the URL policy, is the optional mechanism for providing resource access.

It is interesting to see that Eclipse and the OSGi specification have both come full circle around. The current OSGi specification now promotes both module-level and package-level dependencies. Eclipse now supports both exploded JAR files as well as non-exploded ones. It is also interesting to see that Eclipse supports installing bundles in different ways, such as by-value or by-reference. In

by-value semantics, the install is done through actual content, such as an input stream, from which Eclipse makes a copy. In by-reference semantics, the install is done through a reference to actual content and Eclipse only remembers the reference, using the content in place. These are other variations on the local management of content that our approach allows and encapsulates.

6 Conclusion

In this paper, we presented a framework for building and integrating diverse module systems. As a primary focus, we felt it was important to provide a path towards the integration of existing approaches. As a secondary focus, we felt that it was also important to provide core mechanisms that promote sharing and reuse of core building blocks across different implementations of module systems. We have two main rationales for this work.

The first rationale is promoting a global ecosystem for Java components. Despite the fact that current approaches for module systems are fairly similar, they are built as monolithic platforms that actually fragments the world of components. This fragmentation actually hinders the creation of a global ecosystem where one could freely and productively assemble components across approaches. We believe that module systems are a first cornerstone to address since the ability to share types is a mandatory first step for assembling services.

Our framework enables integration, while being minimally invasive. With all the current platforms we know of, such as GBeans, XBean, Eclipse and Felix, the impacts of integration with our approach are really minimal. It is straightforward for these platforms to implement the concept of modules and interact with our registry. The global resolution process does not introduce any new complexity. In fact, it allows rather simple approaches to integrate with rather complex ones with a pay-as-you-know approach. For instance, an XBean platform can integrate with module-level dependencies on OSGi modules with no undue complexity. Furthermore, the approach would allow XBean to evolve towards finer grain dependencies with limited added complexity.

The second rationale is that variations across approaches are often minimal regarding the implementation impacts, although most communities have their own proprietary implementations and designs. This hinders innovation and synergy because a component platform is a complex system to build and test. Again, module systems are an essential layer of these systems, in which we saw much variations as well as much potential synergy.

Our core mechanisms help to understand and structure the internals of module systems. They help for the first implementation, providing a blueprint architecture. They also help for the evolution of the module system as it naturally changes, as we discussed with the different Eclipse releases and its adoption of OSGi Technology. Our core mechanisms also help synergy among module systems, focusing on similarities rather than differences. As we discussed across our examples, with minimal changes in the implementation of just a few core mechanisms, one can cover the entire spectrum of module systems.

Our approach promotes the separation between mechanisms and policies. It also promotes the ability to provide custom implementations for these core mechanisms. Two key aspects are that it separates the dependency model from actual class loading as well as separates the reification process of classes from issues of module content management. Overall, this approach provides unprecedented flexibility and reuse in building module systems.

Our approach is the only proposal towards the integration of diverse module systems. Our approach raises the consideration of the desired degree of interoperability among module systems. At a minimum, we promote module-level wiring but our concepts support finer granularity interoperability, such as package-level wiring. One key aspect is that we recognize and provide support for a global resolution process for module dependencies. While we enable integration, we do not impose policies or semantics; each individual module system resolves its modules as it sees fit.

The proposed approach and corresponding interfaces are the results of several years of designing and implementing module layers, with different dependency models and different class loader delegation. We believe that the proposed architecture and design is sound for core mechanisms that could be integrated in the Java Runtime Environment for supporting modules, without imposing an actual model for modules or their dependencies. Overtime, this suggests the eventual deprecation of the class loader framework and the adoption of our flexible and adaptive mechanisms for building and integrating module systems.

References

1. OSGi Alliance. Osgi service platform core specification release 4. `http://www.osgi.org`, 2005.
2. Apache Software Foundation. Apache geronimo. `http://geronimo.apache.org/`, 2005.
3. Apache Software Foundation. Apache maven project. `http://maven.apache.org/`, 2005.
4. Apache Software Foundation. Xbean framework. `http://www.xbean.org`, 2005.
5. Apache Software Foundation. Geronimo beans wiki. `http://wiki.apache.org/geronimo/GBeans`, 2006.
6. Eclipse Foundation. Eclipse platform. `http://www.eclipse.org`, 2004.
7. Olivier Gruber and Richard S. Hall. A java framework for runtime modules. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2006.
8. Richard S. Hall. A policy-driven class loader to support deployment in extensible frameworks. In *Proceedings of the 2nd International Working Conference on Component Deployment*, 2004.
9. NetBeans. Netbeans module (plug-in) and rich client application development. `http://platform.netbeans.org`, 2006.
10. Java Community Process. Jsr 277: Java module system. `http://www.jcp.org/en/jsr/detail?id=277`, 2005.
11. Java Community Process. Jsr 291: Dynamic component support for java se. `http://www.jcp.org/en/jsr/detail?id=291`, 2006.

Transparent and Dynamic Code Offloading for Java Applications

Nicolas Geoffray, Gaël Thomas, and Bertil Folliot

Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott, 75015 Paris France
firstname.lastname@lip6.fr

Abstract. Code offloading is a promising effort for embedded systems and load-balancing. Embedded systems will be able to offload computation to nearby computers and large-scale applications will be able to load-balance computation during high load. This paper presents a runtime infrastructure that transparently distributes computation between interconnected workstations. Application source code is not modified: instead, dynamic aspect weaving within an extended virtual machine allows to monitor and distribute entities dynamically. Runtime policies for distribution can be dynamically adapted depending on the environment. A first evaluation of the system shows that our technique increases the transaction rate of a Web server during high load by 73%.

1 Introduction

Some systems evolve in unpredictable environments: choices that were appropriate when launching an application might become inappropriate at a given time during execution. Pervasive computing and web servers belong to this kind of systems. Pervasive computing aims at bringing computer technologies into all kinds of surrounding entities: cars, highways, desktops, refrigerators, and so on. Embedded devices will evolve in different environments, from having plenty of external resources with good network bandwidth like in airports or hotels, to no external resources at all. If an application would like to take advantage of all these resources, it would have to take into account distribution issues and dynamicity of the environment. In the domain of Web servers, a hot-spot-of-the week effect [24] changes a regular situation with few connections to an exceptional situation where the server has to respond to a large amount of requests. One solution consists in dedicating a given amount of machines to balance the load, but this solution is far from optimal for a Web Server, because resources are underused during normal execution.

To face these unpredictable situations, we propose to inject on the fly an adaptable offloading system directly into the runtime when the local resources are overloaded and when new resources become available. Offloading ensures better performances and better responsiveness of the application. The offloading system is removed when unneeded. Our experience is based on an extended Java Virtual Machine (JVM), but it can be applied to other kinds of virtual machines. The main advantages of our solution are:

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1790–1806, 2006.
© Springer-Verlag Berlin Heidelberg 2006

- The system loads the offloading code dynamically: it does not have to reboot the application in order to take into account distribution.
- The offloading code is independent from the application. An application which separates platform-dependent and platform-independent computations can benefit from a network of resources, even if it was not designed for.
- The application can dynamically adapt the distribution policies such as what to offload, or under which constraints (memory, computation). The offloading system is based on a flexible runtime that enables dynamic adaptation.
- The external resources are not dedicated to one application: they can be shared between many applications.
- External resources do not have to install a dedicated virtual machine. The system sends the offloaded code to standard environments.

Our solution relies on JnJVM [29] an adaptable Java Virtual Machine (JVM). JnJVM is enriched with a Java dynamic aspect weaver [12] that modifies applications during execution. We use JnJVM because (i) it is built on the same adaptable runtime as the offloading system and (ii) its aspect weaver is efficient and fully functional [19].

Section 2 describes the underlying architecture of the system that allows dynamic modification of the Java application, and dynamic adaptation of the distribution policies. Section 3 describes what is possible to offload in a Java program. Section 4 presents our offloading system. Section 5 evaluates the infrastructure. Section 6 shows related work and Section 7 concludes the paper.

2 The Execution Environment

Our offloading system relies on three different dynamic mechanisms: code loading, adapting the running system's code and adapting the application's code. The micro virtual machine (MVM) is an execution environment which offers the two first mechanisms. The JnJVM, which is constructed on top of the MVM performs the last. We describe both of these systems in this section.

2.1 The Micro Virtual Machine

The MVM is a dynamic compiler and a dynamic code weaver. It compiles components, whose architecture allows adaptation during execution. A component contains fields and methods. Two mechanisms are provided by the component architecture: replacing a component by an other, or modifying the implementation of a component's method. The dynamic code weaver is responsible for inserting the new code. The compiler of the MVM is described in [21]. The MVM is an enhacement of the system described in [22]. The MVM is also the execution environment of many projects, such as an adaptable Web cache [18], or a flexible micro object request broker [17].

In this project we use the MVM in order to dynamically load the offloading code, and adapt its policies during execution.

2.2 JnJVM

JnJVM is a full JVM that follows the J2SE specification 1.2 to 1.4 [9].[1] It is built on top of the MVM: it extends the MVM with Java-specific components such as a bytecode compiler (JIT) or a Java class loader. JnJVM offers the same performance as open source JVMs such as Kaffe 1.1.6 and is three times slower than industrial JVMs such as IBM. We use the JavaGrande Forum Benchmark Section 2 to evaluate JnJVM on PowerPC and Pentium. We expect that an engineering effort would yield better results, because our compiler and garbage collector are not optimized. The memory footprint of JnJVM is around 20 MB.

Dynamic aspect weaving in JnJVM: An aspect is an association between existing code of the application and new code. Aspect weaving is the act of modifying the existing code to invoke the new code. In this paper, we consider dynamic aspect weaving: the aspects are woven during execution.

We have designed a Java aspect weaver for JnJVM. This aspect weaver is separated from JnJVM and is reused for this work. The Java aspect weaver was presented in [19]. In its implementation, weaving an aspect on a method affects the next calls to the method. The aspect weaver does not perform on stack replacement.

3 Offloading in a Java Application: Discussion

Java manipulates different kinds of entities: classes, objects, methods and threads. Offloading an object means sending the object to a surrogate, offloading a class means offloading all the objects of this class, and offloading a method means executing it (and all the methods it calls) on a surrogate. These entities can all be candidates in our offloading system, but each one has its advantages and drawbacks described in the remainder of this section. Class and object migration is mostly for memory saving, whereas method migration is for computation. Finally, thread offloading means thread migration. Thread migration requires to have at each side (surrogate and server) a dedicated virtual machine, because standard JVM forbids capturing or reconstructing an execution stack. Having a dedicated virtual machine for thread migration is tedious to implement, and requires dedicated virtual machines.

3.1 Object, Class and Method Offloading

In this section, we discuss advantages and drawbacks of the different strategies. Table 1 compares the three approaches on (i) their purpose (processing or memory), (ii) the difficulty to analyze them statically (on the source code), and (iii) the difficulty to analyze them dynamically (during execution).

Object offloading: Choosing an object to offload can be done by analyzing the bytecode or the source code of the application or during the execution of the application. In the first case, algorithms exist [26] to determine the relations between

[1] JnJVM is based on GNU Classpath that follows the specification 1.2 to 1.4.

Table 1. Comparison between class, object and method offloading

Strategy	Purpose	Difficulty of static analysis	Difficulty of dynamic analysis
Method	Processing	*	*
Class	Memory	*	**
Object	Memory	*	***

objects of the program and therefore to know which objects are intensively used. The code of the application is then automatically changed to track all the chosen objects. In contrast, the execution analysis monitors the objects of the application and collects statistics on objects use. With these informations (collected statically or/and dynamically), a graph can be computed. The arcs between objects represent their interactions, and the weight of an object represents the other informations, such as the object's size. We can then apply graph partitioning in order to choose the objects that are the most interesting to offload. Static analysis has a major drawback compared to execution analysis: its informations are not as precise, and the objects interactions discovered can be misleading, for example when hot objects are used intensively at startup but rarely used after.

Class offloading: Offloading a class requires to trace all objects of the class, and to send all of them to surrogates. Choosing a class to offload can also be done by static analysis or during execution time. We can apply a graph partitioning heuristic to choose the classes to offload, depending on their interactions. This strategy suffers from adding to the virtual machine instructions to locate for one class all of its objects. It also suffers from having to send all the objects to surrogates, even if some of these objects are not much used in the program. The advantages of class offloading compared to object offloading is that it can be easier in the virtual machine to monitor classes than objects: the virtual machine does not have to track creation and destruction of the monitored objects.

Method offloading: Offloading a method is an easier way to offload an entity than the two previous ones. Static analysis computes call graphs. Execution analysis collects statistics on methods usage like number of times executed, execution time or data exchanged (size of the arguments and return). Compared to class offloading, the disadvantages of method offloading is that there are more methods than classes in a program. The analyzers have to do more work in method offloading, but the offloading might be more effective for performance: offloading a method concerns especially execution, not memory usage. The memory constrained devices generally use an interpreter, thus they do not lose memory for compiled methods. Offloading objects and classes target memory usage and execution. Finally, method offloading does not suffer from heavy communications between surrogates and the server: because all methods called by the method are sent to the surrogate, execution does not have to go back and forth between the surrogate and the server. With objects and classes, the method calls inside a class's method are sent back to the server. When offloading a method, the objects it uses are attached and the server waits for the result.

3.2 Static Analaysis of Method Migration

In our first prototype we implement method offloading. We are working on integrating static analysis for objects [26], or classes [10] in order to ease the execution environment on decisions of which objects or classes to offload.

Not all Java methods can be offloaded. In Java, in order for an object to be exchanged between Java virtual machines, its class has to implement the `java.io.Serializable` interface. An object that is not serializable can not be exchanged. Our solution targets any kind of applications, but the applications might have been conceived without distribution issues, and therefore, all of their objects are not serializable. There are also some classes in the core classes of Java that are not serializable, like the `java.net.Socket` class which represents a native socket of the operating system.

Our system sends objects as method parameters to the surrogates. Therefore these objects have to be serializable. We chose to modify the bytecode of an application in order to add the `Serializable` interface to classes that do not manipulate platform dependent objects, i.e. have real non-serializable fields. We expect to have an analyzer that will find these classes automatically.

Because we chose method offloading, we have to find which methods can be sent to surrogates. Candidate methods must not manipulate platform-dependent objects, and must not call methods that manipulate platform-dependent objects. For the moment, this analysis is done by hand. The informations given for a candidate method are: (i) the classes it needs for execution, i.e. the methods classes that it calls, and the classes of the arguments, (ii) the static objects accessed by the method and submethods, and (iii) which parameters and static variables might be modified (see Section 4.4). Because hand analysis is unconceivable for large applications, we are working on a tool that will make this analysis automatically.

4 Offloading System

The runtime environment must be enhanced with a system that offloads execution or memory to surrogates. In this section we present how the runtime is modified to be able to analyze execution of the offloading entities, and distribute them. Figure 1 gives an overview of the infrastructure.

The offloading system extends the MVM. Its main activities are:

- Dynamically monitor the application execution. In order to offload computation, the system has to know what is interesting to distribute (Monitoring aspect).
- Distribute Java entities. The platform must perform transparent distribution (Distribution aspect).
- Manage the resources. External resources (surrogates) can be removed or added dynamically. The platform has to take into account all modifications of the environment (Surrogate administration).

– Adapt the distribution by an agent. The agent analyzes the results of the application's monitoring, and decides what to offload and on which resources (Offloading agent).

Our platform does not need to modify the source code of an application, nor does it need to have surrogates executing a dedicated virtual machine. The execution is distributed by sending jobs that any JVM can execute, in order to have a larger amount of external resources possibilities. The following steps are done to realize offloading:

1. Static analysis of the application's bytecode gives a set of Java entities that should and/or can be offloaded,
2. Surrogates register and remove themselves to the platform,
3. An agent is launched inside the platform: it decides when offloading should operate,
4. The agent weaves a monitoring aspect on the entities given at stage 1, when the system is not overloaded,
5. Based on this monitoring and on the current situation of the application (presence of many surrogates, heavy load, etc), the agent weaves the distribution aspect that will send jobs to surrogates,
6. When offloading is not anymore necessary, it unweaves the distribution aspect. Execution continues with no overhead.

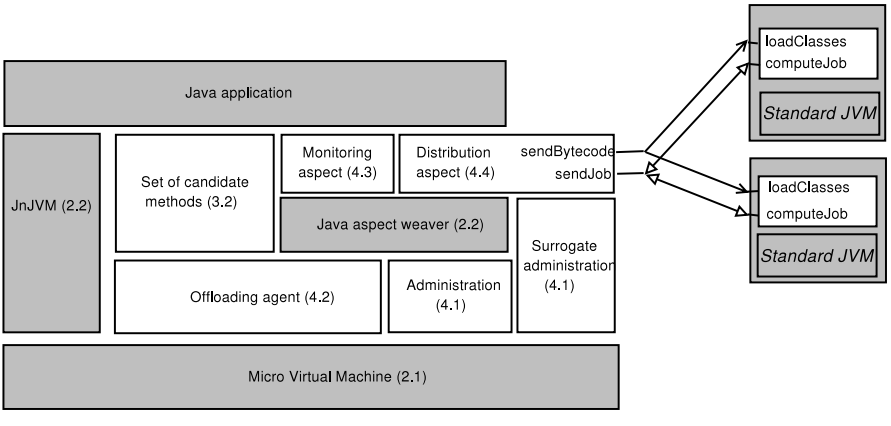


Fig. 1. Overview of the offloading architecture

4.1 Human Administration

There are two entries in the offloading system, each dedicated for two kinds of systems. The first entry targets limited resource devices. The second is Web servers. These entries serve two purposes: (i) administration for the offloading decisions, and (ii) surrogates register themselves dynamically. We are investigating on using a service discovery standard such as Jini [2]. Surrogates can be all kind of computers: grids, servers or simple personal computers.

Resource-limited devices administration: For limited devices, the system is enhanced with a communication socket. An administrator sends through the communication socket instructions for the offloading system: for instance it sends the set of methods that can be offloaded, or it forces a method to be offloaded.

Web server administration: We use servlets to administrate Web servers extended with our load balancing system. A servlet is a Java object that receives requests and sends responses. It is part of the J2EE specification. When used in a Web server, it dynamically creates a web page based on the contents of the request. A servlet is executed inside a servlet container. Our evaluation uses Tomcat (see Section 5), a servlet container. Therefore, instead of a GUI administration approach, we chose a servlet approach. The servlet is responsible for (i) communicating with the offloading system, by sending the instructions given by an administrator through a web page, (ii) printing on the web page the state (local or remote) and (iii) storing statistics usage of all methods. Instructions are sent to the offloading system through the servlet by using HTTP requests.

4.2 Autonomous Agent

The servlet and the communication socket provide entries for an administrator in order to operate on the offloading system. The administrator adapts the system depending on the current environment: heavy load, new peers, memory usage, etc. It decides when to offload a method. This task is not acceptable in the long term due to the permanent human presence it requires. For this reason, an autonomous agent inside the system should manage these tasks: the offloading agent. It is composed of probes and states, and executed as a MVM thread. We have implemented an agent for load balancing the application on top of a Linux system (the application is still in user mode): it probes the operating system's load average (eg, reads the `/proc/loadavg` file), and switches between two states, `charged` and `uncharged`. The `charged` state offloads methods which execution would benefit from being remote. The benefit is calculated by comparing the number of bytes to send versus the network's quality, and comparing the method's execution time versus the time to execute the Java distribution aspect. The `uncharged` state changes execution of all offloaded methods to local, i.e. the agent unweaves the distribution aspect.

Because it is implemented in the MVM, the agent can be dynamically modified by adding new probes or new states. Therefore, the selection of methods to offload can be adapted dynamically. For example, if memory is a constrained resource, probing the `/proc/meminfo` file could be added, as well as a new state `memory`. The agent would be in the `memory` state when there is not much memory left. In such case, it will offload methods with the highest compiled size and free memory taken by the method. Or it may switch to objects and class offloading. We will take into account lemory issues when the system will be enhanced with objects or classes offloading.

4.3 The Monitoring Aspect

We use dynamic Java aspect weaving to inject monitoring in the set of methods that can be offloaded. The monitoring aspect collects three statistics. First, the average execution time in order to know which methods cost the most at execution. Second, the number of times executed in a given elapsed time. This information gives the hot methods of the application. Third, the average size of the objects used (arguments, return, static variables). This last statistic gives an estimation of the distribution cost.

The system uses these runtime statistics in order to choose the most interesting methods to offload. An administrator or the agent can modify the monitoring aspect and add new statistics collecting for methods.

Our prototype collects statistics on Java entities execution, and records the characteristics of each surrogate. Network quality consideration is an improvement we are currently working on.

4.4 The Distribution Aspect

Offloading a method's execution to surrogates results in a distributed execution, where the surrogates (the clients) offer their processing capabilities to the server, and the server sends jobs to surrogates.

Our solution aims at sending jobs to standard environments: the owners of the surrogates do not have to install a dedicated execution environment. This facilitates usage of all kind of resources: it is easier to have a standard environment executing on different architecture platforms than with a specialized environment. The drawback is that it must use a standard distribution framework, not optimized for our kind of system. We chose RMI as a first framework for simplicity reasons. However, the system is not constrained with a unique framework, and we are currently investigating on using different ones.

Each surrogate executes an RMI object that exposes two methods: the first method, `receiveFiles`, receives the Java bytecode of the method and submethods to execute, and the second, `remoteMethodInvocation`, triggers the execution of the method.

Algorithm: The server executing the application holds references of all the peers available. A reference is a stub on which it can call the two RMI methods. When the system choses to offload a method, it weaves the distribution aspect on the method to:

1. Get the static objects used by the methods,
2. Put all arguments in an array,
3. Get a peer (i.e. get its stub) given by the agent,
4. Send the bytecode (call to `receiveFiles`) if the peer does not have it,
5. Invoke the method by calling `remoteMethodInvocation`,
6. Update the modified objects (see Section 4.4),
7. Return the result.

The call is synchronous: once a thread calls the `RemoteMethodInvocation` method, it must wait for the response. When the system decides that offloading the method is not anymore necessary, the distribution aspect is unwoven.

Distribution issues: Because we transparently distribute an application, we must face issues inherent to distribution. We now discuss how our system deals with some of these issues: remote call semantics, memory coherence and fault tolerance.

Remote call semantics: Some distribution frameworks like RMI have a call by copy semantic. This means copies of objects are sent and remote modifications done on these copies are lost when the remote method returns. This is not appropriate for our system, because we must have a transparent distribution, where a remote call becomes semantically like a local call. We used a call by copy-restore semantic to achieve such transparency.

The copy-restore algorithm is expensive: for an object Obj the remote call has to send back the new, modified 0bj and all reachable objects from the old, original Obj. The application that called the remote method receives these objects, and has to trace all references of the old Obj in order to compare them with the new Obj. It is better to analyze the application in order to know which objects must have a call by copy-restore semantic. The informations of a methods that can migrate contain therefore the parameters and the static variables modified in the method.

A non-standard implementation of call by copy-restore already exists for Java, NRMI [31]. We did not use this implementation because it changes the standard Java core classes, and JnJVM uses the GNU Classpath classes. Furthermore, the application code has to be modified in order to specify that an object has to be restored locally when sent to a remote call. Because we do not directly call a migrated method, but instead call a method of the RMI object with the information of what method to call, our solution can bypass modifications to the RMI implementation. If needed (there can be distribution frameworks that already have the call by copy-restore semantic) the call by copy-restore algorithm is inserted in the Java distribution aspect of the server. The implementation of the RMI object returns to the server an array of all objects that could have been modified by the method's execution.

Memory coherence: When a surrogate executes a method, the objects passed as arguments are duplicated on the surrogate. Therefore there can be two copies of one object. If the Java program is mono-threaded, execution is stopped locally, waiting for the result of the remote method invocation: no modifications on the objects can be done locally, and only the remote method can modify them. If the program is multi-threaded, the objects sent to the surrogate are manipulated locally and on the surrogate. This issue concerns the programmer, not the offloading system: if objects are not protected in the program (i.e. synchronized calls), whether the method is executed locally or remotely does not change the application semantic.

Synchronized methods can be migrated, but synchronizations inside the method need particular care: because the method is executed on a different virtual machine than the one that executes the application, a synchronization on a surrogate must be transmitted to the server. We are investigating on modifying

the method by adding remote calls that will inform the server of synchronizations. The modified method will then be sent to surrogates instead of the original method.

Fault tolerance: Method offloading allows our system to be partially fault-tolerant : if a peer executing a method crashes (detected by timeout), the method can be executed elsewhere, because all modifications done to objects on the dead peer are not recorded in the server. However, this solution is not possible if the method interacts with external entities, like databases. For this kind of problem, we are currently investigating on using rollbacks on databases when a surrogate crashes. When the system will be enhanced with object or class offloading, we will consider replications of distributed objects, in order to guaranty fault-tolerance.

5 Evaluation

Our evaluation compares execution of JnJVM without offloading and execution of JnJVM with offloading within a Web application. Comparisons with other JVMs without offloading are for the moment irrelevant: JnJVM and GNU Classpath are in their early stage, and they need engineering efforts to compete with commercial JVMs (see Section 2.2).

The application is a free implementation of TPC-W.[2] It is an interactive bookstore implemented with servlets, that interacts with a MySQL database. The servlet container we used is Tomcat 5.0.28. The server on which the application and MySQL are executed is a 1 GHz PowerBook G4 with 1 GB memory.

The application is composed of 29 classes, 15 of which are servlets, 49 methods and 5409 lines of code. A manual analysis of the code can detect 21 methods which execution can be migrated (most part of the other 28 methods manipulate specific servlet objects, which manipulate platform-dependent objects and are therefore not serializable). The majority of the methods that can be migrated follow this algorithm: (i) contact a local socket pool manager in order to aquire a connection to the database, (ii) make a query to the database and (iii) compute on the query's results. The Java class of the socket pool manager is serializable.

Tomcat is launched on the JnJVM without the offloading code. After startup, we load and install the offloading system in JnJVM. We give to the offloading system the set of methods chosen by our static analysis. The offloading agent starts, and begins to analyze execution of these methods. The surrogates add themselves to the offloading system: in the first experiment, a bi-processor G5 2 GHz and a 3 GHz pentium register to the offloading system. In the second experiment, a cluster of 20 bi-processors Xeon 2 GHz dynamically registers to the offloading system. The JVM executed on these machines are Sun JVM 1.4 for pentium and IBM JVM 1.4 for PowerPC. They all execute the RMI object presented in Section 4.4.

[2] http://jmob.objectweb.org/tpcw.html

Our experiment uses the `siege` utility.[3] `Siege` performs benchmarks for Web servers by creating a user defined number of simulated users that make requests to the server. The statistics reported by `siege` are: elapsed timed, data transferred, average response time, transaction rate, average throughput,average number of simultaneous connections, and availability (the number of requests that were satisfied by the servers).

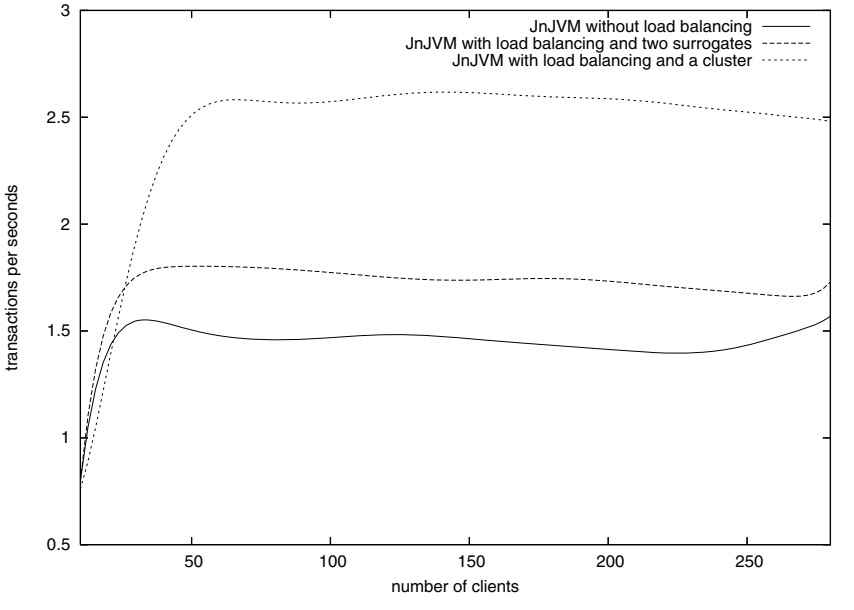


Fig. 2. Transactions per seconds

In the experiments, `siege` was launched for 10 to 400 simulated clients. The clients send requests to the TCP-W application. The first experiment uses JnJVM without load balancing, the second experiment uses JnJVM with load balancing and two surrogates, and the third experiment uses JnJVM with load balancing and the cluster. Figure 5 compares transactions per seconds. The results show that the number of transactions per seconds is increased by 17% with two surrogates, and 73% with the cluster. These numbers do not depend on the number of connections. Figure 5 compares average response time of the server. When 300 clients send requests, the average response time is decreased by 10% with two surrogates and 42% with a cluster. Figure 5 compares availability of the server between the three experiments. The availability of the server starts to decrease when 290 clients are connected, in contrast to 300 for two surrogates, and 360 with a cluster. These results show that our offloading system performs well, however note that increasing the number of surrogates does not lead to higher performance. The small improvement between two surrogates and a cluster of surrogates is mostly due to the fact that the server still has to dispatch the

[3] http://www.joedog.org/siege/

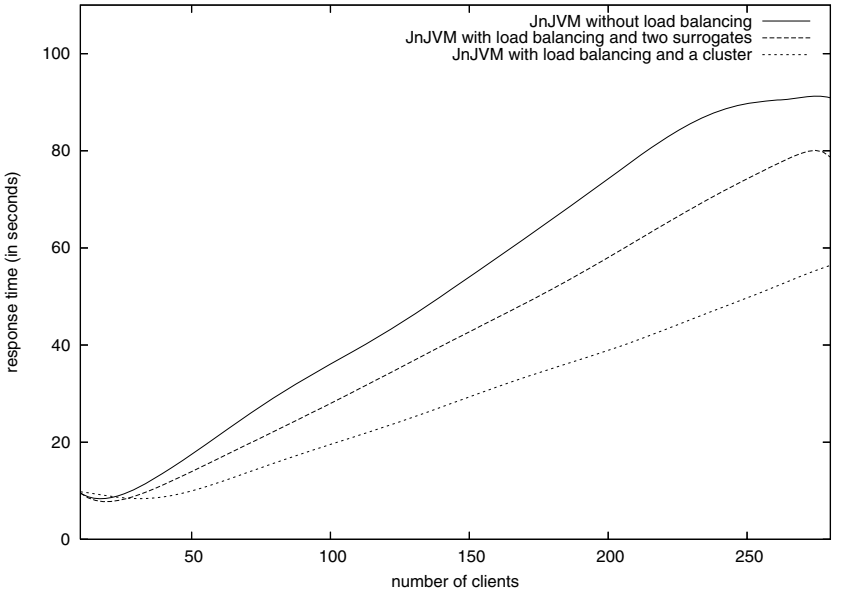


Fig. 3. Average response time

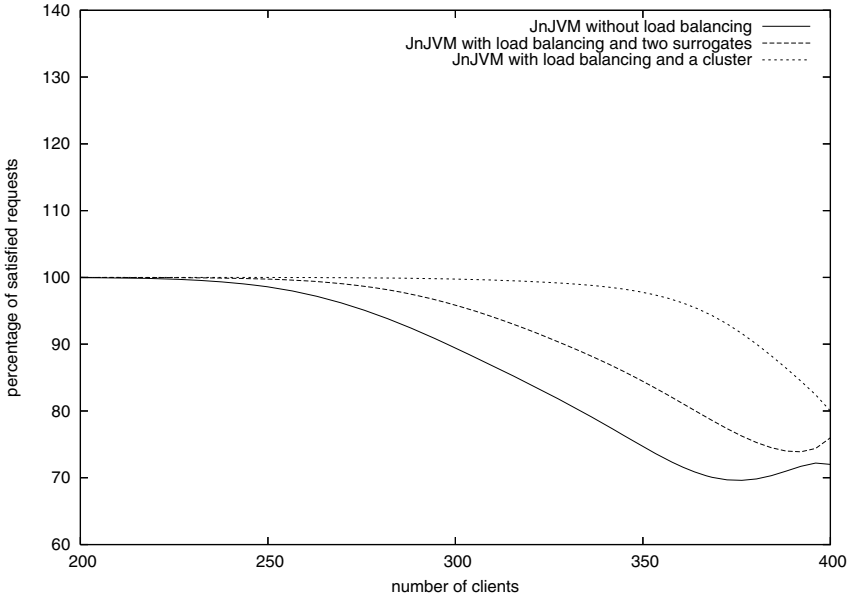


Fig. 4. Availability

requests. The performance of the server are the bottleneck. A solution would be to have a limited pool of servers, shared between many applications, dedicated only to receive requests and call offloaded methods.

6 Related Work

Our work aims at distributing on different resources a centralized Java application. There are mainly three related approaches to achieve this distribution: execution of the program on a dedicated cluster, static automatic distribution of the program, or transparent offloading on possible resources of Java entities during execution of the program.

6.1 Distributed Execution on Clusters

Distributed execution of Java programs on clusters can be achieved with thread migration, a distributed shared memory or a distributed virtual machine.

There are four approaches to thread migration: modifying the source of the application [8], modifying the bytecode [32,23], developing an extended virtual machine like MOBA [25], or using non standard functionalities of a Sun JVM [4]. Bytecode or source modifications adds computations inside the application. This enables the migration of a thread at any state. This additional computation consists in either try/catch clauses or tests and thus reduces performances. An extended virtual machine for thread migration requires to execute it at each node. Our solution aims at executing jobs on standard JVMs. Furthermore, thread migration is tedious and it is easier to implement method migration.

Distributed shared memory can be used to allocate and manipulate objects, like in Java/DSM [34], MultiJav [5], Javanaise [11] or Jessica [14]. The system executes over a cluster of machines, where each node participates in the shared memory, and therefore has an extended virtual machine. Removing or adding a node dynamically is not considered in these systems. This is not satisfactory for our solution, because nodes are here dedicated to the application, and can not remove themselves while the application is executing.

Finally, a distributed virtual machine like cJVM [3] or JESSICA2 [35] allows to execute an application over a cluster where each node participates in the extended JVM execution. These JVMs offer preemptive thread migration for dynamic load balancing. As for distributed shared memory, removing or adding nodes is not considered: nodes have to be present anytime during the application execution.

6.2 Automatic Distribution

JavaParty [20] extends the Java language with a new keyword `remote`, that developers add to classes which instances should be remote. A preprocessor is then applied to write the distributed code. Execution of the application is done in a standard virtual machine with a registry that enables locating objects. Our solution does not modify the Java language. Moreover, the distribution is decided statically in JavaParty and dynamically in our system. As JavaParty, we can offer distribution control with an instruction file that specifies which entities should be executed remotely. Remote Objects in Java [16], like in JavaParty, introduces a new keyword `remotenew` to create objects on another virtual machine. The

new keyword is translated in a new opcode, which can only be understood by an extended virtual machine, which limits the use of the bytecode.

Doorastha [6] allows a programmer to annotate a standard Java program with new keywords in order to specify which objects of the application should be distributed, and how each distributed object should be transmitted to methods. The Doorastha framework compiles the program and chooses where to place objects. Placement can be decided statically or dynamically. Our solution aims at executing existing applications and therefore does not modify the Java language.

Pangaea [27] uses an object graph algorithm [26] which operates on Java source code to find the objects that interact the most in the program. Based on the graph, a distributed code is generated to execute the application between different computers. The distribution code can either be in RMI, CORBA or Doorastha. Objects placement and number of participating processors are decided statically. J-Orchestra [30] and Addistant [28] modify the bytecode of a Java application to generate a distributed code, based on informations given by a programmer or user of the application. These informations specify which classes should have their instances distributed. In J-Orchestra, these classes are transformed to RMI classes. Addistant uses a master/proxy system, where proxies of an object communicate with the master object. Pangaea, J-Orchestra and Addistant modify an application before it is executed. Thus, dynamic adaptation is not considered, and objects that were not selected during bytecode migration can not be changed to distributed objects during execution. Participating resources are given statically, and can not be modified dynamically. Furthermore, these systems do not analyze the application during execution, and can therefore not adapt their distribution strategies at runtime.

6.3 Transparent Offloading

Lattanzi *et al.* [13] improve the performances of a FPGA with a Java interpreter by offloading CPU intensive methods to a co-processor. The Java virtual machine analyzes each method to find which method should be offloaded. The chosen method is compiled and migrated to the co-processor. This system uses shared memory and therefore does not deal with network distribution issues.

Azul Systems [1] performs dynamic load balancing for J2EE applications. The applications will be able to offload their execution to a dedicated grid, composed of dedicated hardware and a dedicated virtual machine. The technology is relatively new and specific to Web applications. Our solution aims at adding distribution to any kind of Java application.

Diaconescu *et al.* [7] propose a compiler and runtime infrastructure for transparently distributing a Java application. It performs a graph analysis based on [26] that performs on bytecode (Pangaea performs on source), and with this graph generates a distributed program. It uses the Joeq virtual machine [33] to compile the unmodified application into a distributed version. Communications between the server and surrogates are in MPI. During execution, the system is monitored to estimate the quality of repartition, but the user can not adapt the

distribution. Our prototype executes on a flexible environment, the MVM, that eases the adaptation of policies.

AIDE [15] extends the ChaiVM virtual machine, a JVM dedicated to embedded devices and analyzes interactions between classes during execution of an application. A graph partitioning algorithm uses the analysis to find the best partitioning in terms of communications between classes. Partitions can be offloaded to surrogates, dynamically and transparently for the application. Surrogates execute also an extended virtual machine. We aim at offloading on standard Java virtual machines, in order to have a large amount of external resources.

7 Conclusion

We have presented a runtime infrastructure that is able to offload computation to surrogates. It performs monitoring and analyzes the results collected in order to distribute execution. Not all applications can benefit from our offloading system: high relations between objects, or high dependencies with non-serializable objects can prevent offloading. However, we believe that a program which separates business code and platform dependent code has a good part of its entities enable to be distributed.

Future works to improve our prototype are:

- To develop the method analyzer. For the moment, the analysis is done by hand. This is not conceivable for large applications: this tool will ease usage of the offloading system.
- To offload objects and classes. Objects and classes offloading serve for more memory saving.
- To integrate different kinds of static analysis. Object graph analysis [26], and class interactions graph [10] will give additional informations on what is best to offload.
- To target different standard distribution frameworks. The current framework is RMI, but frameworks dedicated for to grids could give better performances.
- To execute different kinds of applications with different needs. We have evaluated the advantages of our system for dynamic load balancing for a Web server. Our next target is offloading under memory constraints like in embedded devices.

We have applied dynamic aspect weaving to inject transparent monitoring and distribution in applications. The result is an extensible framework for load-balancing and/or offloading. Evaluation of the system shows that a Web server is improved in responsivness and availability.

Acknowledgements

We would like to thank the anonymous referees for their suggestions to improve the content and presentation of this paper.

References

1. Azul systems. http://www.azulsystems.com.
2. Jini. http://www.jini.org.
3. Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–11, Fukushima, Japan, September 1999.
4. S. Bouchenak and D. Hagimont. Zero Overhead Java Thread Migration. Technical Report 0261, INRIA, May 2002.
5. X. Chen and V. H. Allan. MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 91–98, Las Vegas, USA, July 1998.
6. M. Dahm. The Doorastha System. Technical Report B 00-01, Freie Universität Berlin, May 2000.
7. R. E. Diaconescu, L. Wang, Z. Mouri, and M. Chu. A Compiler and Runtime Infrastructure for Automatic Program Distribution. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 52–67, Washington, USA, 2005.
8. S. Funfrocken. Transparent Migration of Java-Based Mobile Agents. In *Proceedings of the Mobile Agents Conference*, pages 26–37, 1998.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. 2000.
10. X. Gu. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *In Proceedings of the IEEE Pervasive Computing and Communication*, 2003.
11. D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *Proceedings of the Middleware Conference*, The Lake District, England, 1998.
12. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, June 1997.
13. E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo. Improving Java Performance by Dynamic Method Migration on FPGAs. In *Proceedings of the IEEE Reconfigurable Architecture Workshop*, April 2004.
14. M. Ma, C. Wang, and F. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.
15. A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, T. Giuli, and X. Gu. Towards a Distributed Platform for Resource-Constrained Devices. Technical report, Hewlett-Packard, 2002.
16. N. Nagaratnam and A. Srinivasan. Remote Objects in Java. In *Proceedings of the IASTED International Conference on Networks*, 1996.
17. F. Ogel, B. Folliot, and G. Thomas. A Step Toward Ubiquitous Computing: An Efficient Flexible Micro-ORB. In *Proceedings of the ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, sept. 2004.
18. F. Ogel, S. Patarin, I. Piumarta, and B. Folliot. C/SPAN: A Self-Adapting Web Proxy Cache. In *Proceedings of the Autonomic Computing Workshop of the International Workshop on Active Middleware Services*, page 178, June 2003.

19. F. Ogel, G. Thomas, and B. Folliot. Support Efficient Dynamic Aspects Through Reflection and Dynamic Compilation. In *Proceedings of the Symposium on Applied Computing*, Santa Fe, USA, March 2005.
20. M. Philippsen and M. Zenger. JavaParty: Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
21. I. Piumarta. The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation. In *Proceedings of the Virtual Machine Research and Technology Symposium*, pages 97–110, San Jose, USA, 2004.
22. I. Piumarta, F. Ogel, and B. Folliot. YNVM: Dynamic Compilation in Support of Software Evolution. In *Proceedings of the OOPSLA Engineering Complex Object Oriented System for Evolution Workshop*, Tampa Bay, USA, October 2001.
23. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of the Agent Systems and Applications/Mobile Agents Symposium*, pages 16–28, 2000.
24. M. Seltzer. The World Wide Web: Issues and Challenges. Presented at IBM Almaden, 7 1996.
25. K. Shudo and Y. Muraoka. Asynchronous Migration of Execution Context in Java Virtual Machines. *Future Generation Computer Systems*, 18(2):225–233, Oct. 2001.
26. A. Spiegel. Object Graph Analysis. Technical Report B-99-11, Freie Universität Berlin, July 1999.
27. A. Spiegel. PANGAEA: An Automatic Distribution Front-End for Java. In *Proceedings of the Heterogeneous Computing Workshop*, pages 93–99, 1999.
28. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. *Lecture Notes in Computer Science*, 2072, 2001.
29. G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a Flexible Java Runtime upon a Flexible Compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.
30. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.
31. E. Tilevich and Y. Smaragdakis. NRMI: Natural and Efficient Middleware. In *Proceedings of the International Conference on Distributed Computing Systems*, Providence, USA, May 2003.
32. E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the Agent Systems and Application/Mobile Agents Symposium*, pages 29–43, Zürich, Switzerland, September 2000.
33. J. Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *Proceedings of the Interpreters, Virtual Machines and Emulators Workshop*, pages 58–67, San Diego, USA, June 2003.
34. W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.
35. W. Zhu, C. Wang, and F. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proceedings of the International Conference on Cluster Computing*, Chicago, USA, September 2002.

Self-organizing and Self-stabilizing Role Assignment in Sensor/Actuator Networks

Torben Weis[1,⋆], Helge Parzyjegla[2,⋆], Michael A. Jaeger[2,⋆⋆], and Gero Mühl[2]

[1] Application of Parallel and Distributed Systems, University Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
[2] Communication and Operating Systems Group, Berlin University of Technology, Einsteinufer 17, 10587 Berlin, Germany
torben.weis@ipvs.uni-stuttgart.de,
{parzyjegla, michael.jaeger, g_muehl}@acm.org

Abstract. Developing distributed applications for sensor/actuator networks is challenging, particularly, with regard to unreliable nodes and communication links. Splitting applications into roles eases the development significantly but presumes a reliable role management that autonomously assigns roles to devices depending on their capabilities. In this paper, we present a self-organizing and self-stabilizing role assignment mechanism as an integral part of a light-weight, flexible middleware. The deployed algorithms tolerate the addition and removal of devices at runtime and are also able to recover from any transient fault. Considering that resources are restricted on many devices, we analyze the proposed algorithms with respect to communication overhead, memory usage, and stabilization time.

1 Introduction

The development of distributed applications is difficult, especially if the devices that run the code are resource constraint and error-prone. A role-based programming model can leverage a model-driven approach for application development in sensor/actuator networks (*SA-nets*) and provides advantages regarding fault tolerance and management. A typical scenario where SA-nets are used is the e-home. Programming applications for such an environment where no administrator is present and complex network structures can evolve requires the developer to provide a solution that does not rely on manual intervention and that is able to adapt to changes in its environment. These changes include, for example, adding and replacing a device. It is also essential that the application is able to get back to a correct working state (if possible) even if wireless links are disturbed or devices that run (parts of) the application crash unexpectedly. Thus, applications for SA-nets should be self-organizing as well as self-stabilizing.

We developed an integrated stack of algorithms to provide a self-organizing and self-stabilizing substrate for the development of role-based applications for

⋆ Funded by Deutsche Forschungsgemeinschaft.
⋆⋆ Funded by Deutsche Telekom Stiftung.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1807–1824, 2006.
© Springer-Verlag Berlin Heidelberg 2006

SA-nets. It consists of three algorithms that feature a spanning tree, a publish/subscribe infrastructure, and a role assignment mechanism. The algorithms are tightly integrated to reduce the overhead. For example, the publish/subscribe infrastructure uses the heartbeats of the spanning tree and the role activation relies on the publish/subscribe algorithm to discover device capabilities.

The paper is structured as follows: in the next section we discuss in more detail the role concept and what self-organization and self-stabilization means. In Sect. 3 we present the algorithm stack and analyze it in Sect. 4. In Sect. 5 we discuss related work. Section 6 features conclusions and outlook.

2 Roles

Roles provide a powerful abstraction concept by specifying a certain behavior that can be adopted by different entities (e.g., nodes, components, or objects) and abandoned again. Furthermore, a particular entity can fulfill more than one role at a given time, while a particular role can also be performed by many entities. The concept of roles was first introduced by Bachmann and Daya [2] in data modeling and, thereafter, frequently adapted. Here, we use it to ease the development of flexible applications for SA-nets. Therefore, an application is split into several roles that collaborate to provide the desired service. A role consists of the necessary code and data structures—this might also comprise several components or objects—to fulfill a certain task needed by an application. Each role presumes a set of *capabilities* that describe the minimum requirements for a node to be able to serve this role (e.g., a set of sensors or actuators). Thereby, roles decouple the distributed application from the actual nodes it runs on since a role only addresses other roles and not a concrete node itself. A robust *role assignment* mechanism is an important precondition for this role-based development. It is responsible of first assigning all roles that are needed to nodes that are capable of serving them. Afterwards it has to monitor the assigne roles and reassign them if necessary in order to ensure that all applications can fulfill their function. Due to the dynamic and error-prone nature of SA-nets, it is important that the mechanism works as autonomous as possible, i.e., it automatically adapts to dynamic changes and recovers from transient faults without external intervention. The ability to dynamically reconfigure an application here is a main motivation for the introduction of the role-based programming abstraction.

Achieving automatic adaptation in SA-net applications is hard since there is generally no fixed central computer which has global knowledge about available devices, software components, and network connectivity. Many devices in such a system are too weak to maintain global knowledge and the availability of a more powerful machine cannot be assumed. Therefore, every single device must make decisions based solely on its local knowledge such that the global result yields the expected system functionality. Organizing a system of independent entities based on local decisions is a key-property of *self-organizing* systems [10].

A traditional way of dealing with faults is trying to mask them, but obviously not every fault can be masked. If such a fault occurs, the system can get in an invalid state from which it may not recover without human intervention. In

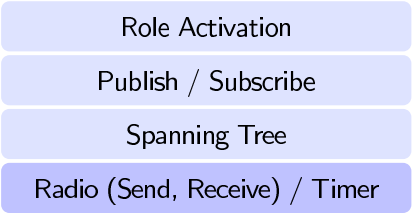


Fig. 1. The role assignment algorithm stack

our application domain, this is not acceptable since the user is hardly able to administer this complex system: a crash can occur on any kind of appliance ranging from the coffee machine to the TV set. *Self-stabilization* can be applied to tackle this problem: a self-stabilizing system is guaranteed to recover from any transient fault after a bounded time [5]. However, this comes at a cost because the system can (in the general case) not detect whether it is currently in a legal state or not. It can only guarantee that an illegal state will be left after a fixed time. This is done by constantly pushing the system towards a legal state.

3 Role Assignment Algorithm Stack

In this section, we present a self-organizing and self-stabilizing role assignment mechanism that consists of three layers made up by an own algorithm each. Figure 1 gives an overview of our algorithm stack which presumes only a simple radio interface able to *receive* and *broadcast* messages. The purpose of the lowest layer is to structure the network. Therefore, it applies a spanning tree algorithm designed for wireless networks. Please note that it is possible to support networks with a fixed infrastructure, too. In this case, a different algorithm must be used in the stack to construct the spanning tree. The second layer comprises a lightweight publish/subscribe algorithm that provides flexible communication by decoupling publishers and subscribers, i.e., senders and receivers of messages. After nodes have subscribed for the roles they perform, messages can be published related to some role instead of being addressed directly to the node they run on. The publish/subscribe algorithm takes care that a message is forwarded to the correct receiver(s). Furthermore, this indirection also eases the migration of roles, since publishers are not required to know their subscribers anymore. Finally, the third algorithm layer manages the role assignment within the network. It monitors the routing tables of the underlying publish/subscribe layer to determine whether for each role there is a node properly subscribed. If this is not the case, a message is sent to a capable node to activate a missing role.

The presented algorithm stack is tightly integrated by exploiting several cross-layer optimizations. This results in higher performance and less communication overhead. For example, the role activation directly inspects the publish/subscribe routing tables to discover devices, their capabilities and currently assigned roles.

The drawback is that one algorithm of the stack cannot easily be replaced by another one even if it exhibits the same functionality. Especially the role-activation algorithm relies on the underlying publish/subscribe algorithm. Furthermore, aspects of self-organization and self-stabilization must be considered on each layer anew and reconciled with the measures taken on other layers. In the following, we discuss each layer of the algorithm stack in more detail.

3.1 Spanning Tree Algorithm

The spanning tree algorithm of the first layer serves several functions. In wireless ad-hoc networks, it structures the network providing the basis for the hierarchical publish/subscribe algorithm which comprises the second layer. Therefore, it constructs the hierarchy which is used to disseminate messages within the network. The algorithm determines the node with the highest ID and chooses it as the tree's root node. This fact is exploited by the role activation afterwards.

The self-stabilizing spanning tree algorithm is an adaption of an algorithm published by Afek, Kutten, and Yung [1]. A pseudo code implementation is shown in Listing 1. The algorithm uses only two functions, which must be implemented by the firmware. Send() broadcasts a message and Receive() waits for a message or a timeout. Furthermore, we require a clock on each node, but the clocks must only be loosely synchronized. These primitives can be expected even on very tiny sensor boards.

Whenever a new node wants to join the tree, it initializes the fields, that store the IDs of the root and the parent node, with its own ID. Afterwards, it waits to receive a TreeMessage from a neighbor node. TreeMessages are broadcasted periodically. They serve as *heartbeats* and allow to detect failed nodes and to refresh the tree's state. They contain the ID of its sender, the ID of the root node as seen by the sender, and the sender's distance to the root node measured in hops. If a TreeMessage contains a root node with a higher ID, the new node selects the sender as parent and updates its fields appropriately (cf. line 16). Furthermore, among all suitable parent nodes from which a heartbeat is received, one is chosen that provides the shortest path to the root (cf. line 19). Heartbeats sent by the parent node are forwarded by re-broadcasting an own TreeMessage.

However, if the new node does not hear of another node having a higher ID, it finally times out and issues a broadcast claiming itself as new root node. Since its ID is then the highest within the network, its neighbors will accept it as the new tree's root and propagate its claim. Interestingly, much the same thing happens, if a heartbeat times out on any other node. The node then assumes that an existing tree is broken or a network partition occurred. Therefore, it simply resets its fields, broadcasts a TreeMessage, and starts over again. In consequence, the root node repeatedly times out and resets itself, because no higher ID exists within the network. Nevertheless, no harm is done, since resetting its fields to its own ID is an idempotent operation in this case. Please notice the usage of a shorter timeout ($\frac{1}{\mathsf{p}} \cdot \mathsf{TIMEOUT}$) for the root node that forces it to initiate heartbeats with a higher frequency (cf. line 31). This allows any other node to miss a heartbeat without running into the risk of a timeout causing a

Listing 1. Spanning tree algorithm

```
1  // The node's ID and predetermined timeout values as builtin constants.
2  const uint ID; const long TIMEOUT; const int p ← 3, t ← 10;
3  // Assumed root node, current parent node, and number of hops to the root.
4  uint Root ← ID, Parent ← ID, Distance ← 0;
5  // Timestamp of the last heartbeat received.
6  long Timestamp ← GetCurrentTime();
7
8  void Main() {
9    Message msg;
10   while ( true ) {
11     // Wait for an incoming message (or a timeout).
12     if ( (msg ← Receive( TIMEOUT/t )) ≠ null ) {
13       if ( msg is TreeMessage ) {
14         // Check, whether a new root or better parent is found.
15         if ( msg.Root > Root ) {
16           Root ← msg.Root; Parent ← msg.Sender; Distance ← msg.Distance+1;
17         }
18         else if ( msg.Root = Root ∧ msg.Distance+1 < Distance ) {
19           Parent ← msg.Sender; Distance ← msg.Distance + 1;
20         }
21         // Process a received heartbeat.
22         if ( msg.Root = Root ∧ msg.Sender = Parent ) {
23           Timestamp ← GetCurrentTime();
24           OnHeartbeat();
25           Send( new TreeMessage( ID, BROADCAST, Root, Distance ) );
26       } }
27       else if ( IsAddressedToNode( ID, msg.Receiver ) ) {
28         OnProcessMessage( msg );
29     } }
30     // Do a reset, if the heartbeat timed out.
31     long timeout ← ( Root = ID ? TIMEOUT/p : TIMEOUT )
32     if ( ¬(0 ≤ GetCurrentTime() − Timestamp ≤ timeout) ) {
33       Root ← ID; Parent ← ID; Distance ← 0;
34       Timestamp ← GetCurrentTime();
35       OnHeartbeat();
36       Send( new TreeMessage( ID, BROADCAST, Root, Distance ) );
37 } } }
```

subsequent reset. This makes the constructed tree more robust with regard to unreliable communication links. However, as Receive() just blocks for at most $\frac{1}{t} \cdot \mathsf{TIMEOUT}$, it is guaranteed, that a failed parent node is detected at least after $\frac{t+1}{t} \cdot \mathsf{TIMEOUT}$, when the current time is compared to the timestamp of the last heartbeat received (cf. line 32). This also allows to test, whether the timestamp lies in the future (i.e., the difference to the current time is negative) in order to achieve self-stabilization even in case of arbitrary memory perturbations that possibly affect any stored value.

The other layers hook up with the spanning tree algorithm in two ways. First, the OnHeartbeat() method is called whenever a TreeMessage is sent. This enables the node to perform certain tasks periodically, e.g., to trigger the routing table propagation of the publish/subscribe algorithm. Second, the spanning tree provides the means for a hop-wise communication along its edges. Therefore, all messages contain a field with the ID of the intended receiver. To address all direct children in a tree, we set the highest bit of the receiver address. To simply broadcast a message, we set all bits of the whole address. This way, a node can send a message to its parent (ID = Parent), to its children (ID = 0x80000000), to its parent and its children (ID = Parent|0x80000000), or to all nodes within broadcast range (ID = 0xFFFFFFFF). The IsAddressedToNode() method tests whether a received message is intended for the current node according to the introduced addressing scheme. If successful, the message is propagated upwards in the algorithm stack via the OnProcessMessage() method.

3.2 Publish/Subscribe Algorithm

To send information across the network, we use a self-stabilizing hierarchical publish/subscribe algorithm [14]. It enables nodes to address messages to a certain role without knowing the device on which the role is currently executed. The publish/subscribe algorithm offers essentially three functions:

- Subscribe(type): subscribes to messages of a given type usually belonging to a certain role.
- Publish(type,message): publishes a message of a given type that is forwarded to all matching subscribers.
- PublishSingle(type,message): sends a message of a given type to just one subscriber provided that at least one exists.

Nodes interested in messages related to a certain role subscribe to a corresponding message type. Their subscriptions are forwarded along the edges of the tree towards the root. Along their way, they install the necessary routing entries to deliver matching publications on the reverse path back to interested subscribers. Therefore, each node has a routing table which stores its own subscriptions as well as subscriptions issued by its direct and indirect children. A routing entry contains the ID of the node the subscription was received from, the type of messages the subscriber is interested in, and an additional timestamp to detect stale entries. Figure 2 provides an example. Nodes are shown together with their IDs and routing tables. A routing entry of (80,A) means that node 80 has subscribed for messages of type A. We omitted the timestamp for reasons of clarity. Nodes can be direct and/or indirect subscribers. Direct subscribers (e.g., node 80) consume the messages they are subscribed for as indicated by a routing entry with their own ID. Instead, indirect subscribers (e.g., node 62) issue a subscription on behalf of an interested child in order to propagate this interest to the root.

Whenever a published message is received, a node searches its routing table for matching subscriptions and forwards the publication to its children provided

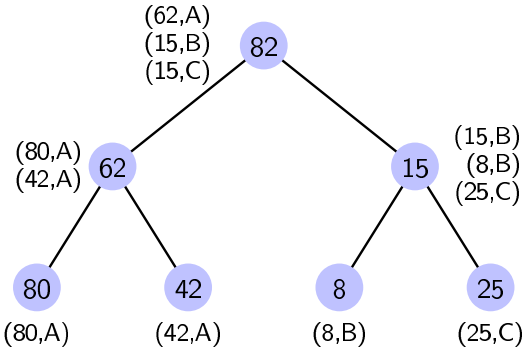


Fig. 2. Routing tables

that at least one child is subscribed for this type of message. If the node itself is interested, the publication is also delivered locally to the subscribing role and consumed there. If the message comes from a child or is published on the node itself, it is additionally addressed to the parent node in order to reach subscribers in different subtrees, too. For example, a message of type A published on node 8 first bubbles up the tree towards the root, before it is disseminated in the left subtree down to its subscribers 80 and 42. When a message is published by the PublishSingle() method, the nodes behave different. Instead of broadcasting the publication to all interested children, only one is directly addressed until a node is reached that consumes it. This function is heavily exploited by the role activation algorithm.

The timestamp in each routing entry is used to detect expired subscriptions. Subscriptions are only leased and must be renewed periodically. By applying leasing, the algorithm is able to recover from corrupted routing tables and to insert missing entries, and, thus, achieves self-stabilization. Since corrupted entries are not refreshed by subscribers anymore, they will finally time out and get removed. All three aspects, removing expired subscriptions, inserting missing ones, and refreshing the remaining ones, are handled by the OnHeartbeat() method as shown in Listing 2.

First, the role activation algorithm is forced to renew its subscriptions by invoking RoleActivation(). Afterwards, each routing entry is inspected (cf. line 4). If it possesses a valid timestamp, the entry's message type is added to a set s. Otherwise, the routing entry is removed. Finally, every node (except the root node) sends one Subscription message to its parent that contains all message types stored in s (cf. line 11). Only this way, new subscriptions bubble up the tree and existing ones get refreshed. When the parent node receives the Subscription message, it invokes OnProcessMessage(). Subscriptions are handled by updating the timestamp of the sender's routing entry for each message type the subscription contains. If a corresponding routing entry does not exist yet, it is simply added. Additionally, OnProcessMessage() also handles publications. First, it checks, whether the node itself is interested. In this case, the message is delivered to the subscribing role by OnConsumeMessage(). If the message was

Listing 2. Self-stabilizing publish/subscribe algorithm

```
1  void OnHeartbeat() {
2    RoleActivation();
3    Set s ← new Set();
4    foreach ( Entry e ∈ RoutingTable ) {
5      if ( 0 ≤ CurrentTime()−e.Timestamp ≤ TIMEOUT ) {
6        s.add( e.Type );
7      } else {
8        RoutingTable.remove( e );
9    } }
10   if ( Root ≠ ID ) {
11     Send( new Subscription( ID, Parent, s ) );
12 } }
13
14 void OnProcessMessage( Message msg ) {
15   if ( msg is Subscription ) {
16     RoutingTable.Update( msg.Sender, msg.SubscribedTypes );
17   }
18   else if ( msg is Publication ) {
19     if ( RoutingTable.ContainsEntry( ID, msg.Type ) ) {
20       OnConsumeMessage( msg );
21       if ( msg.IsPublishedSingle ) return;
22     }
23     [...]   // Propagate the message to subscribed children
24 } }
```

published by the PublishSingle() method, its propagation then ends. Otherwise, it is propagated according to the forwarding scheme already discussed above. We omit the particular code for reasons of brevity.

3.3 Role Activation Algorithm

The role activation algorithm is layered on top of the publish/subscribe algorithm. Since hierarchical routing is used, the root node knows whether a subscription for a given message type exists somewhere in the tree. However, it does not know who exactly subscribed to this message type as it only stores the next hop to the subscriber(s) in its routing table. This knowledge about existing subscriptions is exploited by the role activation algorithm. It enables the root node to decide if it is possible to assign a role and whether a role is currently active. The algorithm's structure can be divided into different phases which we describe in the following.

Phase 1. The devices analyze their capabilities and compare them with the capabilities required by a certain role. We assume that devices are not uniform: some might have special sensors, for example, others have special actuators. Furthermore, their memory, CPU, and energy resources can vary. Imagine an

application A with roles $R_A \subseteq R$, while a device D can play roles $R_D \subseteq R$. Depending on the mode of deployment different approaches are possible. Option one (the preferred one) is to execute phase 1 once upon deployment. The result, i.e., $R_D^A = R_D \cap R_A$ for every device D and application A is stored in non-volatile memory (i.e., flash memory) of D. Thus, it cannot be altered by RAM errors. The second option is to calculate R_D^A upon startup and store it in RAM. The drawback is that R_D^A has to be recalculated periodically to achieve self-stabilization in the face of RAM errors.

Phase 2. After having determined their capabilities, the devices signal which roles they can activate. For every role $r \in R_D^A$ the device subscribes in our example to a message of type $t = (A|r|\mathsf{possible})$. The GUID A uniquely identifies the application, r is encoded as an integer, and the flag possible indicates that the device is able to execute this role, but it has not yet activated it. The subscriptions are periodically renewed by the RoleActivation() function (cf. Listing 3), which is in turn invoked on every heartbeat (cf. function OnHeartbeat() in Listing 2). It is important that the subscription is refreshed directly before the publish/subscribe algorithm sends its subscriptions towards the root node to avoid running into a timeout which would signal an error.

Phase 3. The root node waits until it has received at least one subscription of type $(A|r|\mathsf{possible})$ for all roles $r \in R_A$. If this is the case, then it is possible to find one device for each role. Subsequently, the root nodes proceeds to phase 4.

Phase 4. Now, the root node assigns the roles to individual nodes. Therefore, it publishes a message of type $(A|r|\mathsf{possible})$ for all roles $r \in R_A$ using PublishSingle(). The published message has a boolean payload: true means "activate this role", while false means "deactivate this role". Role deactivation is used if more than one node is activated for one role. This can happen due to duplicated or delayed messages or because of corrupted data structures and is part of the self-stabilizing mechanism of the role assignment algorithm.

Phase 5. The recipient of the message $(A|r|\mathsf{possible})$ will activate role r of application A. Therefore, the recipient subscribes to the message type $(A|r|\mathsf{active})$. Notice that the node is now subscribed to two message types: $(A|r|\mathsf{possible})$ and $(A|r|\mathsf{active})$. If the root node sees a subscription for every $r \in R_A$ and $(A|r|\mathsf{active})$ then all roles are correctly assigned and activated.

Phase 6. This phase is the runtime loop of the application. To send a message to the node executing role r it suffices to simply publish a message of type $(A|r|\mathsf{active})$. The publish/subscribe infrastructure will then forward it to the node executing role r. This is a key advantage of using the publish/subscribe infrastructure for role assignment.

Figure 3 illustrates which subscriptions the algorithm issues and which messages are published. First, two nodes subscribe to role r_1 (phase 2, message m_1). If at least one subscription for every role is available (after phase 3), the root node publishes a message matching the subscription (phase 4, message m_2). The

Listing 3. Role activation algorithm

```
1  void RoleActivation () {
2    foreach ( GUID A ∈ APPLICATIONS ) {
3      foreach ( Role r ∈ R_A ) {
4        // Signal ability to execute role r.
5        if ( r ∈ R_D^A ) {
6          Subscribe( (A|r|possible) ); // Phase 2.
7          // Signal activation of role r?
8          if ( IsActive ( r ) ) {
9            Subscribe( (A|r|active) ); // Phase 5.
10       } }
11       // Reset multiple activations of the same role, when detected.
12       if ( |RoutingTable.GetSubscriptions ( A, r, active )| > 1 ) {
13         Publish( (A|r|possible), false );
14     } }
15     // Role assignment is only performed by the root node.
16     if ( ID ≠ Root ) continue;
17     // Check, if at least one device is available for every role. Phase 3.
18     if ( RoutingTable.HasSubscriptions( R_A, possible ) ) {
19       // (Re)assign roles, if they are not yet active.
20       foreach ( Role r ∈ R_A ) {
21         if ( ¬RoutingTable.HasSubscriptions( A, r, active ) ) {
22           PublishSingle ( (A|r|possible), true );
23       } }
24     } else {
25       // Deactivate whole application, if one of its roles cannot be reassigned.
26       foreach ( Role r ∈ R_A ) {
27         if ( RoutingTable.HasSubscriptions( A, r, active ) ) {
28           Publish( (A|r|possible), false );
29 } } } } }
30
31 void OnConsumeMessage( Message msg ) {
32   foreach ( GUID A ∈ APPLICATIONS ) {
33     foreach ( Role r ∈ (R_D^A ) {
34       // (De)activate role r on receipt of a matching message.
35       if ( msg.Type = (A|r|possible) ) {
36         if ( (bool)msg.Data ) Activate(r); else Deactivate(r);
37       }
38       // Deliver the message's content to an active role r.
39       if ( msg.Type = (A|r|active) ) {
40         OnHandleMessage( r, msg.Data );
41 } } } }
```

usage of PublishSingle() makes sure that only one of the two nodes activates role r_1. Finally, the node that consumed the published message activates role r_1 (phase 5) by subscribing to $(A|r_1|\mathsf{active})$ (message m_3). Any message of the type $(A|r_1|\mathsf{active})$ will now be forwarded to the node executing role r_1.

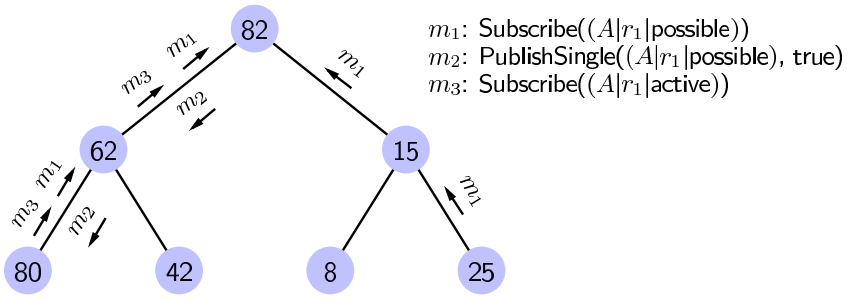


Fig. 3. Example message exchange in the role activation algorithm

Considering self-stabilization, the role activation is in a correct state, if every role needed for an application is assigned to exactly one node that has the capabilities to perform it. If no capable node exists for a particular role, the other roles must also stay deactivated to prevent partially running applications to behave arbitrarily. However, due to faulty messages or memory perturbations, nodes may start or stop executing a role without a correct (de)activation by the root. Nodes also fail and stop working, when they run out of energy. When a role r of application A gets deactivated because of any reason, its subscription $(A|r|\mathsf{active})$ is not refreshed anymore and will vanish after a constant time due to the self-stabilizing publish/subscribe layer. Eventually, the root node detects the missing subscription and returns to phase 4 to reassign role r. If a subscription $(A|r|\mathsf{possible})$ is also missing, no node is left that is able to perform r. The root node then falls back to phase 3 and stops the whole application. Therefore, a deactivation message is sent for all remaining active roles of the respective application. Afterwards, a legal state is reached again.

The case of an unintended role activation is treated differently. If two devices activate the same role r, they both subscribe to $(A|r|\mathsf{active})$, which is detected by their first common ancestor in the tree, when two different subscriptions for the same role arrive. Thereafter, the detecting node causes a reset by publishing a deactivation message telling every node to unsubscribe for this role (cf. line 13). Eventually, the root node detects the missing role and reassigns it as described in the first case.

4 Analysis

In the following, we analyze the three algorithms presented in the previous section. We analyze the worst case message complexity (i.e., the number of calls to Send()) and determine the *stabilization time* Δ, i.e., the time it takes for a system to reach a legal state starting from an arbitrary one. Therefore, let n be the *number of nodes* within the network and d its *diameter* in network hops. Assuming a uniform, planar distribution of nodes, d is in the size of $\mathcal{O}(\sqrt{n})$. Regarding the stabilization time, we introduce π as the *refresh period* a heartbeat message is sent ($(\frac{1}{\mathsf{p}}+\frac{1}{\mathsf{t}})\cdot$ TIMEOUT in Listing 1) and τ as the maximum *timeout*

after which stale routing entries are removed ($\frac{t+1}{t} \cdot \mathsf{TIMEOUT}$ in Listing 1). For the *message delay* δ between neighboring nodes, we require that $\delta \ll \pi$ holds to prevent that heartbeat messages congest the radio channel. Considering the role assignment, $|R|$ denotes the *number of possible roles* within the system.

4.1 Spanning Tree Algorithm

Whenever a node broadcasts a TreeMessage claiming itself as root, the message is only accepted and forwarded by nodes with a smaller ID provided that these do not already know a better candidate. Nodes having a higher ID always ignore the claim. Therefore, a TreeMessage containing the $\mathsf{ID} = x$ is forwarded by at most $x - 1$ smaller nodes. Hence, the worst case regarding the message complexity occurs as depicted in Fig. 4, if all nodes in ascending order time out and issue a TreeMessage, that is forwarded by the smaller ones. Thus, we encounter $n \cdot (n - 1)/2$ messages within the system, which is $\mathcal{O}(n^2)$.

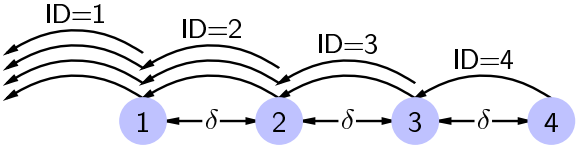


Fig. 4. Worst-case scenario for the self-stabilizing tree algorithm

Concerning self-stabilization, we have just to wait until the highest node times out after τ and issues its TreeMessage. Then, it takes no more than $d \cdot \delta$ to reach every node within the network. Afterwards, it is guaranteed that all nodes know the root which has the highest ID and that every node has selected a parent node closest to the root. Hence, for the stabilization time Δ_{tree} of the spanning tree algorithm the following holds:

$$\Delta_{\text{tree}} \leq \tau + d \cdot \delta = \mathcal{O}(\tau). \tag{1}$$

The memory consumption is constant because every node only stores its children, its parent, the last timestamp, and the root ID plus the distance from the root.

Load Distribution. The spanning tree that results from the algorithm is a shortest path tree since every node chooses the node as its parent that is closest to the root. This reduces the average number of messages required by the publish/subscribe algorithm. However, the number of children is not evenly distributed among the nodes. To illustrate this effect, we simulated the algorithm with a large number of nodes as shown in Fig. 5. The "rings" that emerge here have a distance equal to the *maximum range* of the radio $r_{\max}$. Due to the shortest path property, nodes are more likely to have many children if their distance to the root is a multiple of $r_{\max}$. In conjunction with our publish/subscribe algorithm this can cause an inhomogeneous energy consumption among the nodes,

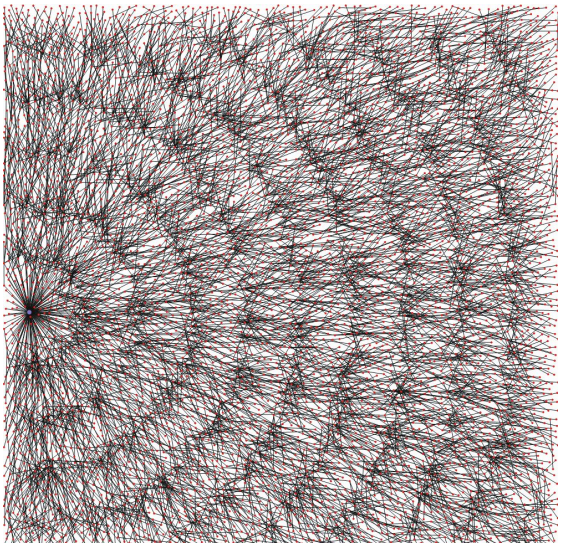


Fig. 5. Simulation of the spanning tree algorithm

since nodes with many children will on average spend more energy on forwarding messages and maintaining routing tables than nodes with less children.

However, this problem can be fixed. In the original version of the algorithm, a child always selects the node with the shortest distance to the root node as parent. Instead of broadcasting the real distance in hops to the root, potential parent nodes in an improved version divide their real distance by the percentage of energy left. Thus, the lower the energy reserves are, the less interesting is the node for potential children. The drawback of this solution is that the resulting tree will not necessarily have the shortest path property. Therefore, the overall energy consumption might increase.

4.2 Publish/Subscribe Algorithm

The periodic update of subscriptions requires in the worst case $n-1$ messages for n nodes. The leaf nodes send their subscriptions to their parent which in turn forwards the aggregated subscriptions to its parent and so on. Thus, we send one message per edge of the tree. The worst case message complexity in forwarding a publication arises in the case where every leaf is subscribed to a filter matching the publication. An invocation of Publish() then results in at most $n-1$ messages as the publication has to be sent to every node. An invocation of PublishSingle() causes at most $2 \cdot h$ messages if the sender and the receiver are both leafs on the lowest level of the tree, where $h \leq d$ is the *height of the tree*. A call to Subscribe() causes at most h messages.

Stale subscriptions will be removed after $h \cdot (\tau + \delta)$ since the propagation of a refresh takes at most time δ and it takes at most time τ until a node removes

the entry if it has not been refreshed meanwhile. Interestingly, new subscriptions need at most $h \cdot \pi + 2 \cdot \delta$ to finally arrive at the root. With every heartbeat, they bubble up one level in the tree towards the root. Additionally, the last heartbeat needs δ to reach a node next to the root, which then responds by refreshing its subscriptions that takes another δ. Alternatively, we could forward subscriptions immediately as discussed earlier [14]. This would decrease the time until the subscription becomes active on the cost of h additional messages for every issued subscription. In total, the publish/subscribe algorithm has a stabilization time $\Delta_{\mathrm{p/s}}$ of

$$\Delta_{\mathrm{p/s}} \leq \max\{h \cdot \pi + 2 \cdot \delta, h \cdot (\tau + \delta)\} = \mathcal{O}(h \cdot \tau). \tag{2}$$

The memory consumption is limited by the size of the routing table which depends on the *maximum degree* D of a node and is, thus, not greater than $D \cdot |R|$, since each child node can issue a subscription for every possible role. This upper bound in the size of the routing table is important because many nodes will have restricted memory resources in practice.

4.3 Role Activation Algorithm

A node that subscribes to $(A|r|\mathsf{possible})$ causes at most h messages. The actual role assignment in phase 4 (which is the invocation of PublishSingle()) results in at most h messages for one role and, thus, in at most $|R| \cdot h$ messages for all roles together. Since it then takes at most time $h \cdot \pi + 2 \cdot \delta$ until all subscriptions $(A|r|\mathsf{active})$ for active roles arrive at the root (which again takes at most h messages per role). Meanwhile the root node times out at most h times, each time calling PublishSingle() again. Hence, the total message complexity of phase 4 and 5 results in at most $|R| \cdot h^2 + |R| \cdot h$ messages, which in fact is $\mathcal{O}(|R| \cdot h^2)$. The messages needed for subscriptions of $(A|r|\mathsf{possible})$ are already counted in the analysis of the periodic costs of the publish/subscribe algorithm.

The role activation algorithm monitors the correct assignment of all required roles and will detect errors, if the publish/subscribe routing tables are stable. It then takes no more than $h \cdot \delta$ (i.e., one call to PublishSingle()) to assign and activate a role, i.e., to make a node subscribe accordingly. These subscriptions need an additional time of at most $h \cdot \pi + 2 \cdot \delta$ to reach the root. Thus, the stabilization time of the role activation algorithm sums up to:

$$\Delta_{\mathrm{ra}} \leq h \cdot \delta + h \cdot \pi + 2 \cdot \delta = \mathcal{O}(h \cdot \pi). \tag{3}$$

The additional memory consumed by the role activation algorithm is limited by the number of possible roles ($\mathcal{O}(|R|)$) provided that at least a flag has to be set indicating that a role is active.

4.4 Role Assignment Algorithm Stack

For layered self-stabilizing algorithms the overall worst case stabilization time is not greater than the sum of the stabilization times of the individual algorithms [5]. The stabilization time Δ of our algorithm stack is thus:

$$\Delta \leq \Delta_{\text{tree}} + \Delta_{\text{p/s}} + \Delta_{\text{ra}} \quad (4)$$

$$= \mathcal{O}(\tau) + \mathcal{O}(h \cdot \tau) + \mathcal{O}(h \cdot \pi) \quad (5)$$

$$= \mathcal{O}(h \cdot \tau). \quad (6)$$

The stabilization time Δ of the whole algorithm stack is bounded by the timeout period τ and the shape of the tree. With $h \leq d$ and $d = \mathcal{O}(\sqrt{n})$, we finally get

$$\Delta = \mathcal{O}(\sqrt{n} \cdot \tau). \quad (7)$$

If no fault occurs, the algorithm stack produces $2 \cdot (n-1)$ messages per refresh period τ: $n-1$ messages for the heartbeat mechanism and another $n-1$ messages for subscription propagation. As long as all required roles are active and the routing tables are consistent, the role assignment stack does not send additional messages.

The overall memory consumption of the three layers consisting of the spanning tree, the publish/subscribe, and the role activation algorithm sums up to $\mathcal{O}(D + |R| + D \cdot |R|) = \mathcal{O}(D \cdot |R|)$. It is thus linear in the number of possible roles and does not depend on the actual size of the network. This is ideal for small devices since they cannot store a global view of all available nodes and their capabilities.

5 Related Work

Self-organization in sensor networks has been an active research field in the past. Algorithms for several specific problems have been presented, ranging from routing and election [18], over data aggregation [19], coverage [17], and lookup services [9], to backbone infrastructures [12] built by using a fixed set of roles.

In [6,15] the authors present an algorithm for generic role assignment in sensor networks. However, their aim is different from ours, as their algorithm is not self-stabilizing and intended for another scenario. They use role assignment to determine cluster heads, or in-network aggregation nodes. For example, they consider clustering to be a special discipline of role assignment. Furthermore, they have a rule-based language to define which properties a node must fulfill to be assigned a certain role. A cluster head must, for example, not have another cluster head in a distance of two hops. This is a very powerful approach, but the authors do not prove that the algorithm always terminates, especially in the face of cyclic dependencies. These can occur, for example, if node A states that it becomes a cluster head if node B is not a cluster head and vice versa. The authors counter this using heuristics and evaluation cycles. In [7] they show how to map a role assignment specification for a given network to an integer linear program formulation in order to better analyze its properties at design time.

The TinyCubus [13] project uses application-specific role assignment for deployment. When sending code updates across the sensor network, they determine which sensor could execute which role. Their code distribution algorithm leverages this knowledge to route code updates only through the set of nodes that really need it, i.e., that execute the matching role. However, they do not provide a concrete role assignment algorithm.

Two projects have investigated self-stabilizing publish/subscribe systems. In [14] an efficient algorithm is presented that builds on a second-chance algorithm comparable to the approach presented in this paper. However, this approach forwards subscriptions immediately causing much more network traffic. On the other hand, the stabilization time is much lower, i.e., in the order of $h \cdot \delta + 2 \cdot \tau$. Another project uses a probabilistic approach where the nodes exchange lossy compressed routing tables to fix inconsistencies [16]. However, this leads only to probabilistic self-stabilization which is weaker than the approach we have taken. Many different algorithms for self-stabilizing spanning trees have been proposed. A decent overview and comparison has been published by Gärtner [8].

We rely on the publish/subscribe communication paradigm to disseminate data in the network. Other approaches have been proposed, too, most notably directed diffusion [11]. However, to the best of our knowledge no role assignment algorithm based on directed diffusion has been devised yet.

The aim of our work is to provide algorithms and tools for self-organizing and self-stabilizing pervasive applications. BASE [4] and PCOM [3] represent a component-based approach to this problem domain. Devices such as PDAs and other appliances build an ad-hoc network and a configuration algorithm tries to compose an application using the components available on the devices. Their approach has three drawbacks. First, it is not self-stabilizing. Second, the configuration algorithm is very expensive since it must solve a constraint satisfaction problem. Finally, the dependencies between the components must build a tree. Our approach does not have these limitations and is more efficient.

6 Conclusion and Outlook

We presented an algorithm stack for self-organizing role assignment in actuator and sensor networks. The contribution of this paper is threefold: First, the entire algorithm stack is self-stabilizing. We can recover from any transient faults including temporary network and memory faults. Furthermore, the algorithms can adapt to a changing network topology and they can handle adding and removing of nodes at runtime. Second, our role assignment algorithm builds directly on a data dissemination algorithm, i.e., a publish/subscribe system. When roles are assigned, every node can immediately send messages to the executors of the roles via the publish/subscribe system. Thus, role assignment and routing to the respective nodes are solved together. Finally, the memory usage of our algorithms is linear in the number of roles and it is independent of the size of the network, which is especially important for resource restricted devices.

In the future we want to optimize the placement of roles in the tree. If we knew that two roles are tightly coupled, it would be beneficial to locate them close to each other in the tree. We are currently investigating how we could exploit the information stored in the routing tables to optimize role placement. Furthermore, we are also extending our algorithm stack. On top of the spanning tree algorithm we are building a clustering algorithm that operates just by snooping the messages exchanged by the spanning tree algorithm. Furthermore,

we are currently evaluating election and process monitoring algorithms. Both are self-stabilizing and utilize the routing table of the publish/subscribe algorithms. The process monitoring is required to ensure that each application is executing exactly once in a network. Otherwise it could happen that all roles are assigned, but the application is not active. Furthermore, due to temporary network partitions it could happen that several instances of one application run concurrently on a network. Currently, developers have to manually define the roles and implement dedicated functionality for every role. Our long term goal is to provide development tools that automatically determine roles through program analysis. Thus, developers will be able to implement self-organizing applications without having to deal with self-organization themselves. This will be supported by special development tools and our algorithm stack.

References

1. Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In J. van Leeuwen and N. Santoro, editors, *WDAG*, volume 486 of *Lecture Notes in Computer Science*, pages 15–28, New York, NY, USA, 1991. Springer.
2. C. W. Bachman and M. Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Data Bases*, pages 464–476, Tokyo, Japan, Oct. 1977. IEEE Computer Society.
3. C. Becker, M. Handte, G. Schiele, and K. Rothermel. PCOM - A Component System for Pervasive Computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, pages 67–77. IEEE Computer Society, Mar. 2004.
4. C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. BASE - A Micro-broker-based Middleware For Pervasive Computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communication (PerCom), March 23-26, Fort Worth, USA*, pages 443–451. Los Alamitos: IEEE Computer Society, Mar. 2003. ISBN 0-7695-1895.
5. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
6. C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 230–240, San Diego, CA, USA, Nov. 2005.
7. C. Frank and K. Römer. Solving generic role assignment exactly. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes, Greece, apr 2006.
8. F. C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report 200338, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, June 2003.
9. K. Herrmann, G. Mühl, and M. A. Jaeger. A self-organizing lookup service for dynamic ambient services. In *25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 707–716, Columbus, Ohio, USA, June 2005. IEEE Press.
10. K. Herrmann, M. Werner, and G. Mühl. A methodology for classifying self-organizing software systems. In *International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)*, Sept. 2006. (accepted for publication).

11. C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *ACM/IEEE Transactions on Networking*, 11(1):2–16, Feb. 2002.
12. M. Kochhal, L. Schwiebert, and S. Gupta. Role-based hierarchical self organization for wireless ad hoc sensor networks. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'03)*, pages 98–107, New York, NY, USA, 2003. ACM Press.
13. P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In E. Cayirci, S. Baydere, and P. Havinga, editors, *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289. Istanbul, Turkey: IEEE, Jan. 2005. ISBN 0-7803-8801-1.
14. G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, and A. Ulbrich. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In J. C. Cunha and P. D. Medeiros, editors, *Proceedings of the 11th European Conference on Parallel Processing (Euro-Par 2005)*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 664–674, Lisboa, Portugal, Aug. 2005. Springer.
15. K. Römer, C. Frank, P. Marrón, and C. Becker. Generic Role Assignment for Wireless Sensor Networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 7–12. Leuven, Belgium: self-publisher, Sept. 2004.
16. Z. Shen and S. Tirthapura. Self-stabilizing routing in publish-subscribe systems. In *3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, pages 92–97, Edinburgh, Scotland, UK, May 2004. IEE.
17. S. Slijepcevic and M. Potkonjak. Power efficient organization of wireless sensor networks. In *Communications, 2001. ICC 2001. IEEE International Conference on*, volume 2, pages 472–476, Helsinki, Finland, 2001.
18. K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie. Protocols for self-organization of a wireless sensor network. *IEEE [see also IEEE Wireless Communications] Personal Communications*, 7(5):16–27, 2000.
19. L. Subramanian and R. H. Katz. An architecture for building self-configurable systems. In *Mobile and Ad Hoc Networking and Computing, 2000. MobiHOC. 2000 First Annual Workshop on*, pages 63–73, Boston, MA, USA, 2000.

Towards Self-organizing Distribution Structures for Streaming Media

Hans Ole Rafaelsen, Frank Eliassen, and Sharath Babu Musunoori

Simula Research Laboratory
P.O. Box 134, 1325 Lysaker
Norway
{hans, frank, sharath}@simula.no

Abstract. Efficient multi-receiver delivery of video data over computer networks, such as the Internet is still challenging, due to heterogeneity in involved network technologies, end-node capabilities and receiver preferences. While adaptive and scalable video encoding can solve many of the problems caused by heterogeneity, media gateway systems are needed for performing video filtering and transformation in order to handle cases not solved by scalable coding. The complexity of such systems makes them difficult to manage by humans. Hence streaming media delivery systems, including media gateway systems should be self-managed only guided by policies provided by system managers and users.

In this paper, we describe a media gateway systems framework for developing binding managers for gateway-based stream bindings. These bindings construct an overlay network for media stream distribution. The framework is open, enabling insertion of specific binding policies that suits particular user preferences. Our framework contributes towards self-management of multi-user streaming services by supporting self-configuration and re-configuration of multi-receiver bindings. It also contributes to easier development of business streaming applications by supporting the plugging in of binding policies targeting particular application domains. We describe the implementation of a prototype of the framework and evaluate the framework by observing and comparing the effect on the property of bindings when different binding policies are applied. Finally we discuss issues of gateway service placement when bindings are deployed in a large scale infrastructure.

1 Introduction

Distributed computing infrastructures of today consist of a diversity of technologies ranging from powerful servers to small mobile devices. The network technology interconnecting devices also varies, ranging from high capacity backbone links to low capacity wireless or modem links. For mobile environments in particular, the performance within a single network technology domain may vary over space and time.

The capabilities of the physical devices also vary considerably. For example, screen resolution may range from HDTV quality screens, down to the relatively low resolution of PDAs. Furthermore, the operating system and application software components used on different devices, are often from different manufacturers In particular, the supported media types may vary across devices. In this context, media types are characterized by their encoding and quality of service (QoS) characteristics.

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1825–1842, 2006.
© Springer-Verlag Berlin Heidelberg 2006

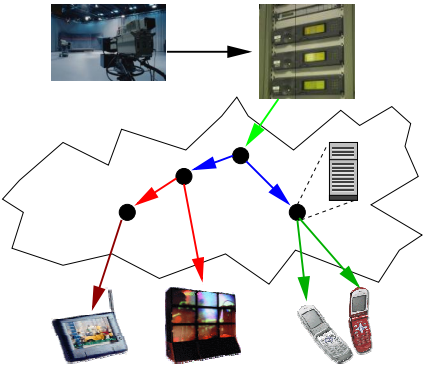


Fig. 1. Distribution graph for heterogeneous media stream

A consequence of the above trends is that supporting interoperability between multi-receiver streaming services and heterogeneous receivers is a major challenge for future media streaming systems. This includes the situation where users access services through (mobile) devices whose capabilities may not even be known at the time of development of the application service. A consequence of this is that the management of bindings[1] must be separated from the application logic and handled by the middleware platform itself. In particular, next generation distributed computing systems will have to be loosely coupled and support late binding in order to cope with this requirement. In this context, it becomes imperative that the supporting infrastructure is able to detect and resolve incompatibilities, including media formats and other capabilities, between communicating devices and services, and that it is capable of automatically adapting audio-visual information to the QoS offered by the network and the processing and display capabilities of the (mobile) devices.

While adaptive and scalable video encoding can solve many of the problems caused by heterogeneity [1], media gateway systems are needed for performing video filtering and transformation in order to handle cases not solved by scalable coding such as when the sender and receiver do not support the same streaming media types. The complexity of streaming media delivery systems for highly heterogeneous environments makes them difficult to manage by humans. Hence such systems, including media gateway systems should be self-managed only guided by policies provided by system managers and users.

In this paper, we describe the design of a gateway-based binding framework that supports multi-receiver real time streaming interoperability between devices supporting different media types. In particular, we deal with support for late binding of media streams, and the provision of middleware managed resolution of media incompatibilities between a sender and multiple receivers when binding participants do not share a common set of media types. The latter we refer to as *heterogeneous media streaming*. The bindings constructed can be seen as *overlay networks* for the distribution of

[1] In this paper the term binding is used both to mean the connection between two nodes such as a sender and a receiver, and to refer to a distribution graph connecting a set of nodes.

the media [2]. Nodes within this network performs computation on the content, thus, the overlay network acts as an *active network* [3], operating at the application level. Figure 1 illustrate a heterogeneous media binding. The binding distributes the stream from the source (camera) to the sinks (PDA, video-wall and smart-phones). To cope with the heterogeneity of the sinks, active elements are inserted at nodes to transform the stream into formats which the sinks are capable of receiving.

The binding framework is based on the concept of open bindings [4]. An open binding is a composite distributed object, used to connect multiple interfaces across a distributed computing environment. Openness allows adaptation of the binding such as inclusion of dynamic content adaptation using media gateways. A media gateway transcodes and/or scales a media stream from a sender into a stream with properties that a given receiver is capable of interpreting. Thus, media gateways located within the binding provides the functionality for supporting heterogeneous media streaming.

In general, there may be many possible configurations of a binding connecting a given sender and set of receivers. The choice may depend on factors such as user preferences, and properties of the application domain. The suggested binding framework therefore, is open in the sense that it enables insertion of different binding policies to guide the construction of bindings. Binding policies typically differ in the goals for the construction of the binding. The binding managers developed within this framework supports self-configuration, and some limited re-configuration of their bindings. Reconfiguration may be initiated when new receivers join the binding.

Our framework contributes towards self-management of streaming services by supporting self-configuration and re-configuration of multi-receiver bindings. To the users, the binding operation can be made as simple as clicking on an icon representing the content of interest. Better support of streaming application development is achieved by supporting an open framework, where domain specific policies can be plugged in. Development of such policies does not require expertise in system programming issues, since these are handled by the framework.

In this paper we also describe the implementation of a prototype of the framework. We evaluate the framework by observing and comparing the effect on the property of bindings when different binding policies are applied.

The work presented in this paper extends on our earlier work on developing a binding framework for multimedia applications These earlier works include the development of languages for specifying bindings and multimedia stream interfaces, methods for detecting incompatibilities between stream interfaces, and methods for discovering media gateways that can be used to resolve an incompatibility between two stream interfaces. In this paper, we show how these results can be exploited by binding managers to dynamically configure and reconfigure bindings for heterogeneous media streaming.

In the next section, an architectural overview of the framework is given. In section 3, the meta-level model for media devices is given, followed by the meta-level model for media gateways. In section 4, we present the binding manager and in section 5 the composition meta-level model. Section 6 present and evaluates our prototype implementation. In Section 7 we discuss issues of gateway service placement and scalability when bindings are deployed in a large scale infrastructure. Related work is given in section 8, and the last section provides our conclusion.

2 Architectural Overview

The framework supports the construction and management of bindings for heterogeneous media streams with a single sender and multiple receivers. A heterogeneous media stream is a stream where one or more receivers of the stream do not support the *media format* produced by the sender. The framework has been designed to satisfy the two following goals:

1. Autonomic configuration and re-configuration of the bindings.
2. Support for an open framework in which different policies that target different problem domains can be inserted.

The first goal states that the bindings should be self configurable and re-configurable. That is, the configuration is done dynamically based on a description of the properties of the interfaces to be bound, and the discovery, selection and composition of components, such as media gateways and media forwarding components, into a binding for heterogeneous media streaming. The discovery and selection of binding components is based on meta data describing their properties, The composition and management of bindings are guided by binding policies. Policies can be seen as rules or plans for how to construct and manage bindings. A policy might specify how the binding graph should be traversed when attempting to bind a new receiver, or how to balance preferences of some receivers against those of others.

The second goal states that binding policies should be easy to change or replace. Different applications of the framework might have different criteria for how the bindings should be constructed and managed. To make the framework usable for different applications, it should be possible to change the policies, while reusing the rest of the framework.

In order to separate and externalize configuration and reconfiguration mechanisms and policies from the code making up the functional part of the binding, The framework generates and maintains a meta-level model of the important implementation aspects of the binding. In addition the model also specifies a set of rules that constrain the interaction between the binding entities. These rules ensures *legal* configuration, and form the foundation for developing algorithms for automatic configuration and re-configuration of the bindings. The framework consists of four kinds of base-level entities:

- **Media source** is an entity capable of producing media streams. These streams might be produced live from a camera, or from a pre-recorded media file. The producer might have the option of delivering the media stream in several different media formats. However, for each active binding, the producer must select a single format for the stream. The selection of one format eliminates the usage of other formats in the same binding. If there is a requirement of a producer to produce multiple concurrent streams using different formats, this has to be resolved using multiple bindings.
- **Media sink** is an entity capable of receiving media streams produced by a media source (producer). Media sources and sinks are commonly called *media devices*.
- **Media gateway** is a network entity that provides conversion of media data carried through the network. If a sink is not capable of receiving the media produced by

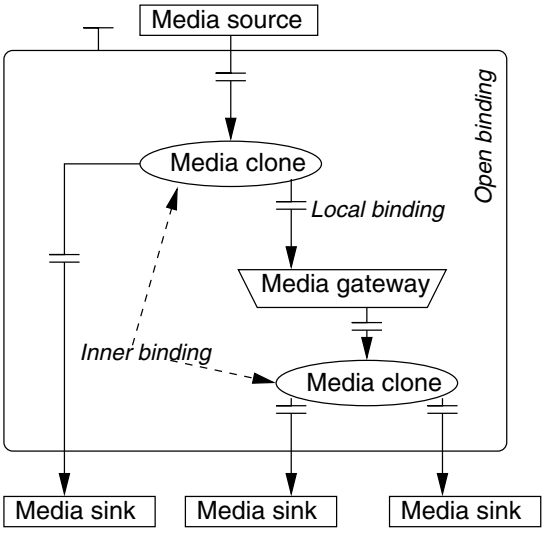


Fig. 2. A heterogeneous binding

the source in the produced format, then a gateway can be inserted in the binding between them. The gateway will do the conversion from the format used by the source to a format that the sink is capable of interpreting.

– **Media clone** is the provider of multicast support. Its task is to take a media stream as input and produce multiple copies of the stream as output. This can be achieved either by simply using underlying multicast support in the network, e.g. IP-multicast, or by using an active node in the network to make multiple copies of the stream, thus acting as a reflector.

Figure 2 illustrates the distribution of a media stream to multiple heterogeneous receivers by the use of an open media binding. The binding itself is a graph of media clones and gateways, connected by local bindings. A local binding is an association of two interfaces within a single address space. The binding in figure 2 distributes the stream from the source, through clones and gateways, to all its connected sinks.

To reason algorithmically about binding composition, the system is realized by the following components:

– **Type checker** - Media endpoints are described within the framework as their interfaces. During binding, the interfaces to be bound must be type checked for compatibility. Informally, type checking means ensuring that the properties of each source interface are as expected by the corresponding sink interface.
– **Gateway trader** - The operations performed by a gateway within a binding, can be seen as a service. The gateway trader provides mechanisms that enable gateway providers to offer their service, and allows users of such services to search for gateway offers, based on the properties of the gateways.
– **Composition meta-level model** - The task of the composition meta-level model is to maintain a graph representation of the binding. It keeps references to the base

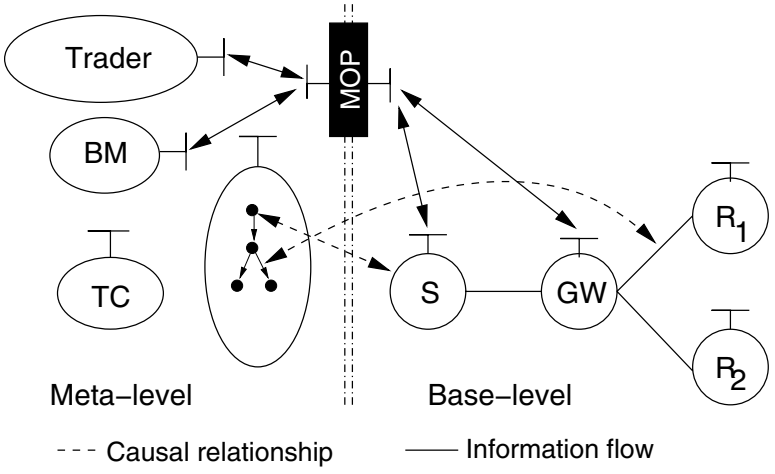


Fig. 3. Relationship between base-level and meta-level

level entities that is represented in the graph, as well as references to meta-level representation of the properties of these entities. Additionally it keeps meta data on how the base-level entities are connected to each other.

– **Binding manager** - The construction and management of the bindings is the task of a binding manager. Binding managers can be embedded with different *binding policies*. The binding manager uses the functionality provided by the components described above to conduct its work.

Figure 3 illustrates the relationship between the base-level and meta-level. The meta-level has a representation of the base-level entities. Some of the meta-level functionality is made available to base-level entities through a *meta object protocol* (MOP).

3 Specifying Properties of Media Devices and Gateways

Different media devices such as video cameras and screens often support different media formats. This might refer to the *media encoding* used, properties such as video resolution or color depth, or even the protocols supported for media streaming. In [5] we describe the Flow Interface Description Language (FIDL) that we use for the description of device media streaming capabilities. FIDL is used to describe the set of possible configurations of the input (sink) and output (source) media interfaces of the devices. In addition algorithms has been developed to determine if an input and output media interface of two devices respectively have compatible interfaces enabling them to be bound.

The properties of the media interfaces are specified as the set of alternative values they can be configured with for a given set of attributes. So if a source interface has the following alternative values for the attribute:

```
video.samplerate = {10, 15, 20, 25, 30}
```

and a corresponding sink interface has alternative values:

`video.samplerate = {10, 15, 20}`

then the two interfaces can be bound (or are compatible) with a samplerate of either 10, 15 or 20, given that the other attributes of the two interfaces also can be configured with common values. In general two media interfaces are compatible (i.e., can be bound) if they have complementary direction and for all their attributes have at least one attribute configuration in common. The framework does not impose any limitation on what flow parameters can be used, so it is possible to used parameters given by standards.

A media gateway can be seen as a component having input and output media interface. Additionally media gateways are described by specifying the casual dependency between its input and output. This dependency captures the functional behavior of the gateway and is expressed as attribute constraint relationships between attribute of the input and output interfaces. This we call the constraint part of the gateway specification.

Causal dependencies are specified using a constraint language. A constraint might be that the samplerate of the video output from a gateway must be lower or the same as that of the input. Consider for example the constraint specification

if `IN.video.samplerate` **then** `>= OUT.video.samplerate`

An effect of this specification is that if both the input and output interfaces of the gateway are able to handle sample rates of 10, 15, 20 and 25, the input interface are to be bound to a source that produces video with a sample rates of 15, then the possible sample rates of the output interface of the gateway are 10 or 15.

From the description of the behavior of a gateway, it is possible to reason algorithmically about its suitability to resolve an incompatibility between two media interfaces that otherwise could not be directly bound. In [6] we investigated how the CORBA trader could be used to select a suitable media gateway to resolve an incompatibility. However we found the expressiveness of the constraint language of the CORBA trader to be to weak to express the constraint part of a media gateway behavior specification. For that reason we have proposed a new gateway definition language called GDL, and implemented a gateway trader that uses this language to trade gateways [7].

4 Binding Manager

A binding manger has no a priori knowledge on how to construct and manage bindings. The binding manager can be configured with different binding *policies*, that define plans or rules on how to construct and manage the bindings. Dependent on the requirement for the binding, the binding manager can be configured with the required plan for the kind of bindings that are required. Binding plans usually deals with issues related to making choices from multiple alternatives. Also strategies on how to search for alternative configurations can be given by the plan. However, the rules determining the legal configurations for the binding, such as the conformance rules of media types, are hard coded into the binding framework, and can not be overridden.

At least two operations are supported by all binding managers: `bind()` and `unbind()`. The bind operation will bind a new receiver to the existing binding, while the unbind operation will remove a receiver. The new receiver requests the binding manager to be bound to the sender. The binding manager first attempts to bind it directly to the sender. If this is not possible, due to incompatibility or the policy given by the binding plan, the

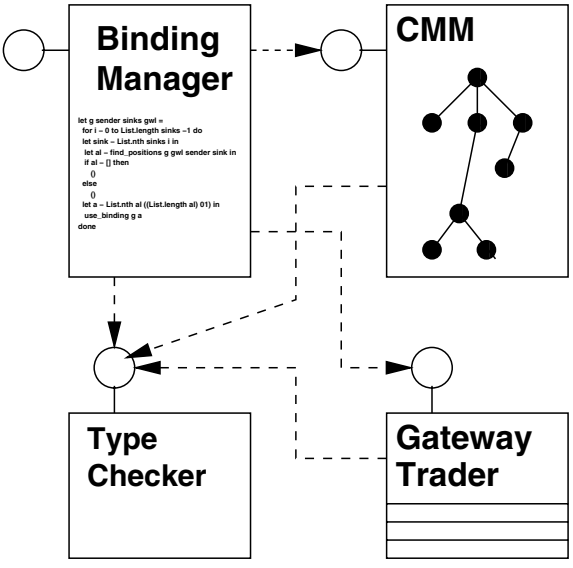


Fig. 4. Binding manager and support modules

binding graph will be searched, to find other positions in the graph, where it is possible for the new receiver to bind.

A *meta* representation of the binding is used by the binding manager. The meta level model capture important properties of the *base-level* computing system. This is represented in a format which enable the binding manager to reason about the binding. The base-level computing system are entities such as network links and computing devices. The configuration at the meta-level can be mapped down to the base-level realization.

The binding manager is dependent on the *composition meta-level model* (CMM). This module keeps a graph representation of the current stream binding. The CMM provides methods to navigate and manipulate the graph. In addition it enforces the invariants that always have to hold for the graph. That is, all manipulating operations on the graph are checked against these rules. Any operation that breaks some invariant, will be rejected.

Figure 4 shows the relationship between the binding manager, composition meta-level model, gateway trader and type checker. The type checker is used by the other components to ensure legal configurations. The binding manager uses the CMM to keep the meta-level representation of the binding, and the gateway trader to locate gateways.

5 Composition Meta-Level Model (CMM)

The binding manager keeps track of the current binding through the *composition meta-level model* (CMM). This model represents the bindings in a *stream-graph*, and supports the necessary operations to manipulate the graph. Currently, our binding model supports stream bindings between a sender and multiple receivers.

The sender, gateways and receivers are represented as vertexes in the graph. *Clones* are another kind of vertex. A clone is essentially an abstraction of multicast. A clone is inserted into the graph, whenever the stream from a vertex needs to be delivered to multiple receiving vertexes. A clone can be implemented by the usage of a multicast address, given that the infrastructure supports this. In cases where this is not supported, some active node will have to be inserted. This node takes the stream from the source vertex as input, makes multiple copies of this stream and forwards each copy to the receiving vertexes. Clones see the stream as a bit stream and is therefore able to operate on any stream, regardless of its media properties.

Bindings between the stream interfaces of the vertexes are represented as directed edges in the graph. The direction of the edge represents the direction of the stream. These edges contain the set of possible configurations of the stream between the two vertexes. It is required that the stream has at least one possible configuration. During binding of two vertexes, the set of possible configurations of their interfaces have to be computed. How they are computed is dependent on what kind of vertexes gets bound.

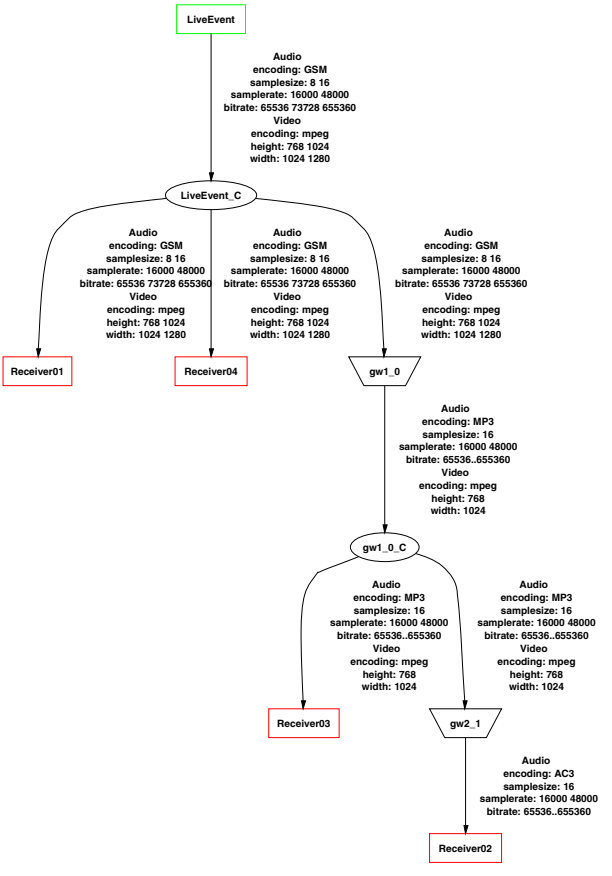


Fig. 5. Dump of a simple stream graph

An API to bind and unbind receivers is provided by the CMM. The binding can be direct or through some given gateway. In situations where insertion of a new vertex will result in a non clone vertex having more than one output edge, a new clone will be created and inserted. The out stream of the source vertex binds to the input of the clone. The receiving vertexes will then get bound to this clone. Binding or unbinding a vertex to a clone, might result in a change in the set of possible configurations of the stream going from this clone. These changes might result in changes in the set of configurations on the bindings downstream from this clone. Whenever the set of possible configurations for the binding coming in to a clone is changed, CMM will traverse the graph downstream. During this traversal all bindings in the downstream are re-computed, in order to reflect the changes done upstream.

In addition to the base level operations for binding and unbinding receivers, CMM support some meta level functions. These functions are mostly used for debugging. They give access to inspect and manipulate the graph directly. Visualization of the graph is also supported. Figure 5 shows the visualization of a graph, using the meta level function. The set of possible configurations between two nodes are printed to the right of the edges connecting the two nodes.

6 Prototype

To demonstrate the soundness of our approach, a prototype has been implemented. It has been written in the OCaml programming language [8], and consists of roughly 10000 lines of code. The implementation contains 4 major parts. The type checker for media interfaces, gateway trader, composition meta model and binding manager. In addition it supports parsing of media interface description (FIDL) and gateway descriptions (GDL).

The goal for the implementation has been to provide the basic functionality required for the construction and maintenance of stream bindings. The experiments are done to demonstrate that different policies can be implemented at top of the mechanisms provided, and that these policies gives the binding manager different behavior.

6.1 Binding Construction Examples

We have implemented a set of proof of concept binding managers. Each binding manager uses different policies for the construction and maintenance of the bindings. Binding policies determine how the bindings are constructed and maintained. The goal for these prototypes is to demonstrate the flexibility of our binding framework.

The first binding manager is quite simple in its approach to binding new receivers. It first attempts to bind the receiver directly to the sender. If this fails, it searches the gateway trader for gateways which will enable to bind the receiver to the sender through a gateway. Should also this fail, it traverse the graph, trying to bind the receiver to each node it visits. First by binding directly, and secondly by the use of a gateway. If the new receiver is unable to bind at any point in the graph, then it fails to bind.

The remaining binding managers use *selection criteria* to determine the insertion point of a new receiver. A selection criterion is a policy that decides which alternative position to choose, given there are multiple choices. We experimented with two different selection criteria:

The first selection criterion attempt to maximize the number of possible interface configurations for the receivers in the binding. Given two receivers R_1 and R_2. Let C_1 and C_2 denote the set of possible interface configurations for R_1 and R_2 respectively. Assume that $C_1 \neq C_2$ and $C_1 \cap C_2 \neq \emptyset$. That is, they have at least one configuration in common, but not all. Let S denote a sender with configuration $C_s = C_1 \cup C_2$. The sender has all configurations as both receivers. Let $C_\Delta = C_1 \cap C_2$. This is the set of configurations that the receivers have in common. If only R_1 binds to S the binding can have the set C_1 as possible configurations, and similarly if only R_2 binds to S the binding can have the set C_2 as possible configurations. If both R_1 and R_2 binds to S at the same time, then the set of possible configurations will be C_Δ. As can be seen, $C_\Delta \subseteq C_1$ and $C_\Delta \subseteq C_2$.

Hence sharing a sender with other receivers might result in a reduction of the set of possible configurations a receiver might have in the binding. The policy computes the set of possible configurations of the existing binding, and the new set of configurations if a new receiver is inserted into different positions in the binding. The position where the reduction of configurations is the lowest is then selected.

The second selection criterion is based on *QoS profiles* [9]. These profiles use *utility functions* to specify how *good* the receiver thinks an attribute value is. Finding the best utility is generally an NP-hard problem [10], so the problem has been simplified. All attributes have values ordered from worst to best. When computing the utility, the best possible value is selected, and the utility for this is computed. The selection criterion computes the total utility for the existing binding by selecting the best available value for all attributes of the receivers participating in the binding. Then all positions in the binding where the new receiver can be inserted get located. The total utility for the whole binding if the new receiver gets inserted at these possible positions gets computed. The position in the binding where the reduction in utility is the lowest when inserting the new receiver is selected for insertion of the new receiver.

Armed with the two selection criteria, two more binding managers were implemented. These binding managers work by locating all positions in the graph where it is possible for a new receiver to bind. The binding managers then use the selection criteria to compute the cost of binding at these positions. The position in the binding with the lowest cost is selected for binding the new receiver. Should there be more than one position with the same lowest cost, the binding managers prefer not to use a gateway. Otherwise, they just select one of the positions. The metric for cost for the first binding manager is the position where the reduction of possible configurations is the lowest, while for the second is the position where the reduction in utility is the lowest.

A last binding manager that combines the policies of the previous binding managers is also implemented. The policy of this binding manager is to first attempt to bind at the first possible position. However, once it fails to bind a new receiver this way, it will break and rebuild the existing graph. The idea is that at this point more information about the receivers is available to the binding manager. Under normal operation, the binding manager knows the current configuration of the binding, and the capability of the next receiver. At the rebind point, the binding manager knows the configuration of the binding and the capabilities of the set of receivers that participated in the previous binding. This knowledge can be used when rebinding the receivers. At first the new

binding only contains the sender. Rebinding is done by successively selecting the receiver which keeps the set of possible configurations the highest when bound. This is repeated until the all receivers have been rebound. Since the new graph has been created while trying to keep the set of configurations as high as possible, there is a better chance that the new receiver will be able to bind to this new binding.

6.2 Evaluation

To evaluate the performance of the implemented binding managers, a simple experiment was conducted. The time, in milliseconds, spent during binding, the number of receivers able to participate in the binding, and the total utility for these receivers was measured. A sender, 4 gateways and 16 receivers where used. The interfaces of the nodes had a complexity of 6-7 attributes in average. The attributes had between 1 and 10 attribute values. For each binding manager, the binding was built 10 times. The order of arrival for the receivers was randomized for each time. The prototype was compiled using the native compiler for OCaml. It ran on a AMD Athlon(TM) XP 1900+ CPU. During the experiment no other CPU demanding processes were run.

Table 1. Performance of binding managers

	First fit			**Most configurations**			**Best utility**			**First fit or rebind**		
	Res.	Time	Util	Res.	Time	Util	Res.	Time	Util	Res.	Time	Util
1	14	19.1	0.4738	14	35.8	0.4738	14	35.8	0.4738	13	428.6	0.4936
2	4	4.3	0.2186	14	46.1	0.4738	14	43.5	0.4738	13	239.2	0.4936
3	14	15.7	0.4866	14	32.6	0.4866	14	32.6	0.4866	14	258.3	0.4866
4	4	5.3	0.2186	14	34.9	0.4866	14	36.7	0.4866	14	146.2	0.4866
5	4	4.5	0.2186	14	30.8	0.4866	14	36.4	0.4866	14	267.0	0.4866
6	3	5.1	0.8638	14	88.7	0.4738	3	4.6	0.8638	14	94.1	0.4738
7	4	4.6	0.2186	14	32.4	0.4866	14	33.0	0.4866	14	464.8	0.4866
8	3	4.5	0.8638	13	29.1	0.4936	3	4.0	0.8638	13	196.9	0.4936
9	13	16.2	0.4936	13	30.2	0.4936	13	31.0	0.4936	13	272.3	0.4936
10	14	18.5	0.4738	14	33.8	0.4866	14	35.9	0.4738	13	86.7	0.4936
Avg	7.7	9.8	0.453	13.8	39.4	0.4842	11.7	29.3	0.5589	13.5	245.4	0.4888

Table 1 shows the result of the experiment. For all experiments, the first column shows the number of receivers able to participate in the binding. The second column show the binding time spent constructing the binding. The last column shows the utility obtained for the receivers participating in the binding.

As can be seen from the table, these prototype implementations and experiments show how the different binding policies effect the resulting binding architecture and the performance of the binding manager.

7 Base-Level Gateway Selection and Binding

The first issue when realizing the above model in a base-level system is the willingness to provide the gateway hosting service. Here we see two possible scenarios. The

first is that it is provided by the infrastructure provider, i.e. the ISP. As part of buying the network service, the gateway service will be part of the solution. Providing gateway services might actually gain the infrastructure provider, in that it might reduce the total bandwidth requirements. The second possible solution is that service is provided through collaboration by the end users themselves. In such a solution, the protocol must be resilient to the frequent joining and leaving of users, and for that reason also gateways. Work on end-user streaming services might help on finding solutions here [11] [12], but this will be an area that requires more research.

Another important aspect for realization of the system, that is currently not part of the proposed solution, is to determine a directed tree from a sender to all its receivers as a base-level realization. If the gateways are already fixed to a set of computing nodes, the selection of appropriate gateways and binding is somewhat simplified as the solution space is limited. However if the gateways are components that can be uploaded to nodes [13], the possibility of placing multiple gateways on to a set of nodes and selecting appropriate bindings make the solution space more complex to search for. A most common factor when selecting one or more gateway hosting nodes is the cost of using them with regard to the network resources and the application performance. That is, resource usage such as communication cost, delay and bandwidth usage should be taken into account when selecting a particular gateway node over another.

When a media streaming application involves a sender and a receiver (i.e., unicast), the selection of the nodes and gateway placement in order to produce a path or a route from source to destination is a complex task. In its initial form this problem is similar to the well known combinatorial optimization problem, vehicle routing problem (VRP) [14] which is proven to be NP-hard. There has been several different solution approaches proposed (For example, [15] [16]) to solve this problem. Another well known traveling salesman problem (TSP) can be seen as a special variant of the basic VRP problem. Although some solution approaches perform better than other in a given application context, evolutionary algorithms [17] [18] that use different heuristic methods are proven to be promising solution approaches.

When an application involves more than one receiver (i.e., multicast), the above problem turns out to be a directed steiner graph problem [19]. In other words the problem at hand is to construct a multi-cast routing tree which is a result of node selection, gateway assignment and binding such that the overall performance of the application is maximized while the cost incurred is minimized. Fundamentally this problem has been shown as NP-complete. Several solutions have been proposed to construct a multicast tree with different application objectives [20] [21] [22] [23]. Further [24] also addresses the reconfiguration of a constructed multicast tree as often the users enter or leave the application triggering a reconfiguration request.

In our heterogeneous multimedia streaming application service context, it is likely that new users may join the binding, and often the underlying resources (gateways, communication links) connected over the Internet change. Therefore one would not be interested in determining an optimal multicast tree (directed steiner tree) but a feasible one. Some advancements in multi-objective multi-constraint optimization came through the evolutionary algorithms [25] that use more advanced and scalable solution approaches like Ant system and simulated annealing, and learning automata. Recently some of

these techniques have been successfully applied to solve the application service placement in a grid like environment [26] [27]. Besides several similarities identified among these problems, it is worthwhile to investigate and extend the current existing solution approaches to address the multicast tree construction problem.

Apart from the overview of the multicast tree construction problem and different possible solution approaches, the performance of any given solution approach depends on the algorithm used and also on the design of different decision making elements such as binding managers. Often it is the case that there exist an entity or a binding manager which is responsible for making all the global decisions that take place during the process of a multicast tree construction. In such a centralized setting, the coordinating agent is informed of different gateway service offers and the resource availability. In addition to its simplicity, centralized approaches are well suited for the systems of limited size. For large systems, it would be useful to adapt a decentralized scheme in which multiple agents (binding managers) work more independently to accomplish their assigned task [28]. In other words, an agent is responsible for making at least one or more decisions. The convergence of the overall system is seen as a combined behavior of all agents involved in the multicast tree construction process. For this, agents communicate with other through a coordination technique. Though such a multi agent decentralized methods scale well, their success depends on the coordination technique used in the algorithm.

8 Related Work

In recent years, there has been much work done on the topic of live content distribution. The goal is to distribute the same content to a number of receivers in an efficient manner. One approach is to build an overlay multicast network, such as "MBONE" [2]. Special routers within the network participate in the multicast protocol. Commercial streaming software usually does not rely on multicast. Instead a component called reflector is used. A reflector receives a copy of a stream, and then forwards multiple copies of the stream to other reflectors, or media servers. A client connect to one of the servers. Another trend is towards peer-to-peer technology in order to provide end system multicast [12]. Each host participating in the multicast plays the role as an client, reflector or server, in order to form a multicast tree.

To cope with heterogeneity the content might be encoded in or transcoded to different formats [29] [30]. Another approach is to use layered encoding [31], and let the clients subscribe to the multicast streams for the layers which they need.

Supporting media processing within the network has been proposed as a solution for solving the heterogeneity problem. Use of video gateways have been suggested by Turletti and Bolot [32], as an alternative to layered coding, when the receivers have different bandwidth requirements. In [33] Yeadon et. al. a set of filters are created which implements and demonstrate the idea. Amir et. al. [34] design and implement a video gateway to address the problem of heterogeneity, in particular difference in bandwidth.

Work on locating media gateways have been an area of research. In the AGLP control protocol proposed by Ooi and van Renesse [13], programmable Internet servers which process multimedia stream are discovered. The protocol requires clients to upload the

media processing gateway program. A multicast extension is proposed, in which several clients can take advantage of the same media gateway. The search criteria when locating a media gateway is its processing capabilities, and placement along the source to sink path. Placement is of interest in order to take advantage of bandwidth reduction as much as possible. The work of Xu et. al. [35] also deals with locating media gateways in the MeGaDip protocol. This protocol takes location and resource situation into account when discovering gateways.

The BMA protocol proposed by Rosenberg and Schulzrinne [36] deals with issues in locating gateways to the public switched telephone network, within the Internet. It is tailored toward voice over IP applications. The Service Location Protocol [37] propose standards for locating services within the Internet.

Microsoft's DirectShow [38] and Sun's Java Media Framework [39] uses the concept of filters. They are components which can be plugged together to process media streams. They are mainly targeted used at the endpoints, as part of the application. Evaluation of the usefulness of a particular filter is done by the application itself, and not treated as an additional service provided by the binding framework.

Work that considers QoS specifications and/or negotiations includes [40] [41]. These works have their focus on network and OS level QoS. In [42] Dini and Hafid propose a formal model for service parameters. The model is used for automation of QoS negotiation of multimedia. In many respect, the goal of this work is similar to our work on FIDL, though the algorithms and implementation are different. QML is a QoS specification language [9]. Its semantics is similar to our stream and flow type model, and from our judgment should be capable of specifying quality properties of stream interfaces. However, its applicability has been demonstrated for operational interfaces only.

Our work emphasizes on the description of media devices and gateways. In particular we build our work on a type model for describing flow types. The properties of media devices and gateways are separated from the device and gateway implementations them selves, and treated as meta-information. This increases the flexibility of the binding framework in that it enables reasoning about the usefulness of gateways without having access to their implementation. Thus, the plan for binding configurations can be created at the meta-level, before the binding protocols start to install/activate the particular components of the binding. Gateways can be inserted anywhere in the distributed computing environment, given that technology domain provides *active service* [43] facilities. End-system multicast can be supported by letting the end-systems participate in the binding as media gateway or clones. The binding framework enables construction of binding managers which provide a simple interface to application programmers or users, while the binding manager takes care of the details of binding construction.

9 Conclusion and Further Work

In this paper we have presented a framework for developing binding managers. These binding managers support the creation and management of distributed heterogeneous media bindings. The framework provides the basic mechanisms while new binding policies that exploit these mechanisms can be plugged into the framework. Another goal of this work is to construct binding managers which are easy to use for end user

or applications developers. The bind operation can be made as easy as clicking on an icon representing the content of interests. Unbind is similarly simple. The complexity of the operations is handled by the binding manager. Self-configurable computing systems are seen as important factor in the future. These systems manage themselves given high-level objectives. We have demonstrated the feasibility of such an approach within the limited problem domain of creation and management of heterogeneous distributed multimedia bindings.

The scalability of the framework is limited in the sense that the meta information of the binding manager becomes a central component. Some degree of scalability can be achieved by using multiple binding managers. Each operates within a administrative domain. Cross domain traffic goes between domain gateways. The domain gateway has the role as a sender node within the domain. All binding management is kept within the domain. Another binding manger takes care of management of cross domain bindings.

In the current implementation the binding is constructed in the meta level without paying attention to base level issues. Base level issues are concerns regarding physical location of nodes and the topology of the underlying network. In addition, resource availability is not considered in the current version. Resources of interest to a binding manager will typically be available computing capacity at the nodes, bandwidth capacity and delays on links between nodes.

For future versions of the binding framework, we plan to incorporate awareness of location, topology and resource availability. Furthermore the current framework is not resilient to failure of intermediate nodes. This is something we plan to address in further versions of the framework.

References

1. Eide, V.S.W., Eliassen, F., Michaelsen, J.A.: Exploiting content-based networking for fine granularity multi-receiver video streaming. In Chandra, S., Venkatasubramanian, N., eds.: 12th Annual Multimedia Computing and Networking (MMCN'05), SPIE, San Jose, California, USA (2005) 155–166
2. Eriksson, H.: Mbone: the multicast backbone. Commun. ACM **37**(8) (1994) 54–60
3. Tennenhouse, D.L., Smith, J.M., Sincoskie, W.D., Wetherall, D.J., Minden, G.J.: A survey of active network research. IEEE Communications Magazine **35**(1) (1997) 80–86
4. Fitzpatrick, T., Blair, G., Coulson, G., Davies, N., Robin, P.: Supporting Adaptive Multimedia Applications through Open Bindings. In: 4th International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, Maryland, USA (1998)
5. Eliassen, F., Mehus, S.: Type Checking Stream Flow Endpoints. In: Middleware'98, The Lake District, England, Chapman & Hall (1998) 305–322
6. Rafaelsen, H.O., Eliassen, F.: Trading Media Gateways with CORBA Trader. In: International Symposium on Distributed Objects and Applications (DOA'01), Rome, Italy (2001)
7. Rafaelsen, H.O., Eliassen, F.: Design and Performance of a Media Gateway Trader. In: Proceedings of Distributed Objects and Applications (DOA'02), IEEE, Irvine, CA, USA (2002)
8. INRIA: Objective Caml. http://caml.inria.fr/ (2003)
9. S. Frølund and J. Koistinen: Quality-of-Service Specification in Distributed Object Systems. Distributed Systems Engineering Journal **5**(4) (1998)

10. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: Practical solutions for qos-based resource allocation problems. In: IEEE Real-Time Systems Symposium, Madrid, Spain (1998) 296–306
11. Padmanabhan, V., Wang, H., Chou, P., Sripanidkulchai, K.: Distributing streaming media content using cooperative networking, in ACM/IEEE NOSSDAV, Miami, FL, USA, May 12-14 (2002)
12. hua Chu, Y., Rao, S.G., Zhang, H.: A case for end system multicast (keynote address). In: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, ACM Press (2000) 1–12
13. Ooi, W.T., van Renesse, R.: An adaptive protocol for locating programmable media gateways. In: ACM Multimedia 2000, Los Angeles, California, USA (2000)
14. Lenstra, J.K., Kan, A.R.: Complexity of vehicle routing and scheduling problems. Networks **21** (1981) 221–227
15. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. Lecture Notes in Computer Science **1520** (1998) 417–??
16. Thangiah, S.: Vehicle routing with time windows using genetic algorithms (1995)
17. Machado, P., Tavares, J., Pereira, F.B., Costa, E.: Vehicle routing problem: Doing it the evolutionary way. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (2002) 690
18. Tan, K., Lee, L., Zhu, Q., Ou, K.: Heuristic methods for vehicle routing problem with time windows (2001)
19. Chlebik, M., Chlebikova, J.: Approximation hardness of the steiner tree problem on graphs (2002)
20. Shukla, S., Boyer, E., Klinker, J.: Multicast tree construction in network topologies with asymmetric link loads (1994)
21. Klinker, J.: Multicast tree construction in directed networks (1996)
22. Ramanathan, S.: Multicast tree generation in networks with asymmetric links. IEEE/ACM Transactions on Networking **4**(4) (1996) 558–568
23. Chen, S., Nahrstedt, K., Shavitt, Y.: A QoS-aware multicast routing protocol. In: INFOCOM (3). (2000) 1594–1603
24. Griwodz, C., Vik, K.H., Halvorsen, P.: Multicast tree reconfiguration in distributed interactive applications. In: the IEEE Consumer Communications and Networking Conference (CCNC'06). (2006) 1219–1223
25. Coello, C.A.C., Veldhuizen, D.A.V., Lamont, G.B.: Evolutionary Algorithms for Solving Multi-Objective Problems. Kluwer Academic Publishers, Norwell, MA, USA (2002)
26. Musunoori, S., Horn, G.: A Fixed Structure Learning Automaton Solution to the Quality Aware Application Service Configuration in a Grid Environment. In: Proceedings of 17th International Conference on Parallel and Distributed Computing and Systems (PDCS 2005), Phoenix, AZ, USA. (2005)
27. Musunoori, S.B., Horn, G.: Ant-based approach to the quality aware application service partitioning in a grid environment. In: IEEE Congress on Evolutionary Computation (CEC 2006), IEEE (2006)
28. Jennings, N.R.: Coordination techniques for distributed artificial intelligence. In O'Hare, G.M.P., Jennings, N.R., eds.: Foundations of Distributed Artificial Intelligence. John Wiley & Sons (1996) 187–210
29. Mohan, R., Smith, J.R., Li, C.S.: Adapting multimedia internet content for universal access. IEEE Transactions on Multimedia **1**(1) (1999) 104–114
30. Fox, A., Gribble, S., Chawathe, Y., Brewer, E.: Adapting to network and client variation using active proxies: Lessons and perspectives (1998)

31. McCanne, S., Jacobson, V., Vetterli, M.: Receiver-driven layered multicast. In: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications, ACM Press (1996) 117–130
32. Turletti, T., Bolot, J.C.: Issues with multicast video distribution in heterogeneous packet networks. In: Proc. 6th International Workshop on PACKET VIDEO, Portland, Oregon (1994) F3.1–3.4
33. Yeadon, N., Garcia, F., Hutchison, D., Shepherd, D.: Filters: QoS Support Mechanisms for Mulipeer Communications. IEEE Journal on Selected Areas in Computing (JSAC) special issue on Distributed Multimedia Systems and Technology **14**(7) (1996) 1245–1262
34. Amir, E., McCanne, S., Zhang, H.: An Application Level Video Gateway. In: ACM Multimedia 95, San Francisco, California (1995) 255–266
35. Xu, D., Nahrstedt, K., Wichadakul, D.: MeGaDiP: A Wide-Area Media Gateway Discovery Protocol. In: 19th IEEE International Performance, Computing, and Communications Conference (IPCCC 2000), Phoenix, AZ, USA (2000)
36. J. Rosenberg and H. Schulzrinne: Internet Telephony Gateway Location. In: Proceedings of the Conference on Computer Communications (IEEE Infocom), San Francisco, California, USA (1998)
37. Veizades, J., Guttman, E., Perkins, C., Kaplan, S.: Service Location Protocol. IETF RFC-2165. (1997)
38. Microsoft Corporation: DirectX 9, DirectShow (2002)
39. Sun Microsystems: Java Media Framework, version 2.1.1 (2001) Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
40. Nahrstedt, K., Smith, J.M.: The qos broker. IEEE Multimedia **Spring 1995, Vol. 2 No. 1** (1995) 53–67
41. Vogt, C., Wolf, L.C., Herrtwich, R.G., Wittig, H.: HeiRAT – Quality-of-Service Management for Distributed Multimedia Systems. Special Issue on QoS Systems of ACM Multimedia Systems Journal (3) (1998) 152–166
42. Dini, P., Hafid, A.: Towards Automatic Trading of QoS Parameters in Multimedia Distributed Applications. In: IEEE/IFIP ICODP/ICDP Conference, Toronto, Canada (1997) 166–179
43. Amir, E., McCanne, S., Katz, R.: An active service framework and its application to real-time multimedia transcoding (1998)

Bulls-Eye – A Resource Provisioning Service for Enterprise Distributed Real-Time and Embedded Systems

Nilabja Roy, Nishanth Shankaran, and Douglas C. Schmidt

Vanderbilt University, Nashville TN, USA 615-343-8197
{nilabjar, nshankar, schmidt}@dre.vanderbilt.edu

Abstract. Middleware is increasingly used to develop and deploy components in enterprise distributed real-time and embedded (DRE) systems. A key challenge in these systems is devising resource management algorithms that deploy application components properly onto target nodes. To provide an accurate view of system resource utilization, these algorithms need monitor resources at runtime. Runtime resource monitoring is also needed to make redeployment or reconfiguration decisions triggered by various factors, such as failures, attacks, overloads, or changes in quality of service (QoS) requirements. DRE systems with a diverse range of applications can therefore benefit from a common resource provisioning service capable of monitoring resource data and enabling proper resource allocation in a timely manner.

This paper provides two contributions to the study of runtime resource provisioning for enterprise DRE systems. First, it describes the challenges in developing *Bulls-Eye*, which is an open implementation of the OMG standard Target Manager specification that provides a reusable service for provisioning distributed resources in enterprise DRE systems. Second, it presents the results of experiments that applied *Bulls-Eye* to the multi-layer resource management subsystem of a shipboard computing environment. Our results show that provisioning resources at runtime in a DRE system via *Bulls-Eye* simplifies resource management and helps automate adaptations in the face of dynamic changes in operating conditions.

Keywords: Resource Provisioning, Component Technology, Dynamic Resource Management, CORBA Component Model.

1 Introduction

Problem → Resource provisioning in component-based enterprise DRE systems. Applications in the domain of enterprise distributed and real-time embedded (DRE) systems, such as shipboard computing environments, satellite constellations, and surveillance and reconnaissance systems, are characterized by stringent quality of service (QoS) requirements and operate in dynamic and resource-constrained environments. The operating modes of these systems may dynamically vary in response to changes in policies or input loads, and they often execute across heterogeneous platforms. Certain enterprise DRE system characteristics, such as their longevity and complexity, motivate the use of component-based development. In this context, components are units of implementation and composition that have well-defined QoS

R. Meersman, Z. Tari et al. (Eds.): OTM 2006, LNCS 4276, pp. 1843 – 1861, 2006.
© Springer-Verlag Berlin Heidelberg 2006

requirements and resource consumption profiles. In enterprise DRE systems, applications consist of groups of domain-related tasks that can be implemented by parameterized and executable software components using QoS-enabled component middleware platforms, such as the OMG Lightweight CORBA Component Model (CCM) [3] and PRiSM [14].

Although component technologies can help enhance software reuse, maintenance, and extensibility [19], they also introduce new *deployment and configuration* challenges [15] stemming from the need to shield applications and users from the complexities of heterogeneous and dynamically changing hardware/software environments. The process of deploying enterprise DRE systems involves creating a *deployment plan* that allocates available computing and communication resources (e.g., memory, CPU, and network bandwidth) to the components and establishes connections between them. To prepare an effective deployment plan, a DRE system needs to know the resources available in the target domain so that resource consumption profiles of the components can be mapped properly to the available *computing nodes* (such as servers, workstations, and PDAs) and c*ommunication links* (such as Gigabit Ethernet or Fibrechannel). It is also important to track resource usage at runtime so that components can be redeployed and/or reconfigured to adapt to changes in application operating conditions caused by policy changes or failures, which must be detected quickly so the system can adapt with minimum disruption.

Solution approach → Resource provisioning service. One way to address the challenges described above is to create *a resource provisioning service* that (1) monitors the resources available in the target domain, (2) supplies this information to human and/or automated planners who prepare a deployment plan using the current resource profile, (3) dynamically allocates resources to deployed components and releases resources when the components are terminated, and (4) facilitates component redeployment and reconfiguration based on resource availability and constraints. Developing such a resource provisioning service for enterprise DRE systems is hard due to the need to handle platform heterogeneity, ensure responsiveness and scalability, and enable dynamic updates within time constraints.

This paper describes the design and application of *Bulls-Eye*, which is an implementation of the Lightweight CCM Target Manager specification [6] that is tailored to the needs of enterprise DRE systems. In particular, we designed *Bulls-Eye* to optimize its CPU and I/O usage to provide fast/predictable access to resource information and enable its use to provision enterprise DRE systems with a range of QoS requirements. The resulting object-oriented framework has been integrated with the *Component-Integrated ACE ORB (CIAO)* (www.dre.vanderbilt.edu/*CIAO*), which is an open-source[1] implementation of Lightweight CCM that has been applied to several enterprise DRE systems, including a shipboard computing system and a prototype of a NASA science mission.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes a case study that motivates the need for a resource provisioning framework

[1] The implementation of *Bulls-Eye* is freely available as open-source software and can be downloaded along with the *CIAO*, DAnCE, and *RACE* open-source middleware from www.dre.vanderbilt.edu/*CIAO*.

in shipboard computing systems; Section 3 discusses the structure and functionality of the *Bulls-Eye* Target Manager; Section 4 explains the design challenges that we overcame while developing *Bulls-Eye* and applying it to the shipboard computing domain; Section 5 summarizes the results of experiments that measures the overhead of *Bulls-Eye* and demonstrates its utility in the context of a prototype shipboard computing system; Section 6 compares our work on *Bulls-Eye* with related research; and Section 7 presents concluding remarks and outlines our lessons learned during this project.

2 Case Study: An Enterprise DRE System for Shipboard Computing

This section describes the structure and functionality of the *Multi-Layer Resource Management (MLRM)* subsystem for shipboard computing, which we use as our running case study in the paper to motivate our work on *Bulls-Eye*. A shipboard computing environment is a metropolitan area network of computational resources and sensors that provides on-demand situational awareness and actuation capabilities for human operators, and responds flexibly to unanticipated runtime conditions. To meet such demands in robustly and predictably, the shipboard computing environment uses services in the *MLRM* subsystem to (1) bridge the gap between shipboard applications and the underlying operating systems and middleware infrastructure and (2) support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization. To support the operational tempo in modern shipboard computing systems, the *MLRM* software must adapt to dynamic conditions for the purpose of utilizing the available computer and communication resources to the highest degree possible to meet changing mission needs.

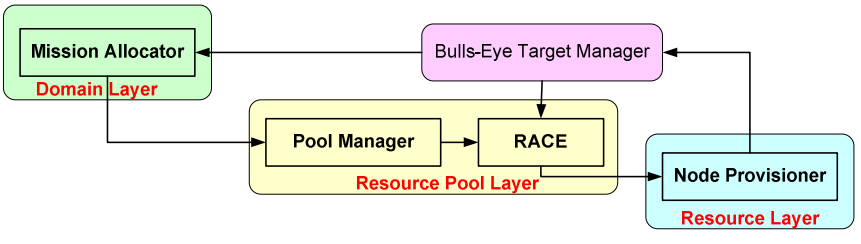


Fig. 1. Layered Architecture of the *MLRM* Subsystem

The *MLRM* subsystem described in this paper consists of the three layers shown in Figure 1. The command and policy inputs flow in a top-down manner, whereas the resource status information moves in a bottom-up manner. At the top is the *Domain Layer*, which consists of the *Mission Allocator* that collects command and policy inputs and passes them onto the *Resource Pool* Layer, which represents a set of computing resources managed by a *Pool Manager*. The Pool Manager in turn interacts with the *Resource Allocation and Control Engine (RACE)* [13], which a reusable framework that separates resource allocation and control algorithms from the underlying middleware deployment, configuration, and control mechanisms so that different algorithms

can reuse common middleware mechanisms to (re)deploy components onto nodes and manage the node's resources among competing applications. The bottom layer is the *Resource Layer*, which contains the entire set of hardware elements in the shipboard computing environment, known as the *Target Domain*. Each node in turn contains a *Node Provisioner* that receives commands from *RACE* to create and destroy applications on the node.

The *MLRM* subsystem is built using *CIAO*, which combines *Lightweight CCM* [5] mechanisms (such as standards for specifying, implementing, packaging, assembling, and deploying components) and *Real-time CORBA* [7] mechanisms (such as thread pools and priority preservation policies). The *MLRM* subsystem has scores of different types and instances of CCM components written in ~500,000 lines of C++ code and residing in ~1,000 files developed by five teams at different locations. The scale, complexity, longevity and multiple QoS requirements of shipboard computing necessitates that its components be deployed and allocated using effective resource management techniques [5]. This requirement, in turn, motivates the need for accurate information on resource availability in the domain. The *Bulls-Eye* Target Manager shown in Figure 1 provides this functionality for the *MLRM* subsystem by capturing runtime information on resource usage that helps *RACE* optimize component allocation and meet end-to-end QoS requirements.

Bulls-Eye is used during initial system deployment when *RACE* runs algorithms to allocate components to nodes in a resource pool. These algorithms interact with *Bulls-Eye* to obtain information regarding resource utilization in the target domain, which is used to produce the deployment plan needed to deploy the system via *DAnCE*, which is *CIAO*'s implementation of the OMG Deployment and Configuration (D&C) specification [1]. This specification standardizes many aspects of deployment and configuration for component-based distributed systems, including component configuration, assembly, and packaging; package configuration/deployment; and repository management of component implementations. *Bulls-Eye* is also used at runtime to extract dynamic resource data and update component implementations dynamically, e.g., in response to damage or to handle changing workload levels.

A particularly important function of the resource allocation and control algorithms in the *MLRM* subsystem is the (re)deployment and (re)configuration of components based on their operational context. For example, a shipboard computing environment may need to switch rapidly from crew entertainment mode to ship defense mode, which necessitates updating and/or migrating many computing services. *Bulls-Eye* therefore provides mechanisms to retrieve the resource availability data across the entire target domain by monitoring and dynamically updating component resource usage. *RACE* uses data provided by *Bulls-Eye* and the requirements of each component to generate an optimized deployment plan and to ensure that the allocated components conform to the characteristics of each node's hardware, OS, middleware, and programming language(s), which can be highly diverse.

3 The Design of the Bulls-Eye Target Manager

Bulls-Eye is a resource provisioning service designed to enable software developers and applications in enterprise DRE systems to (1) retrieve a list of the initial available

resources in a target domain, thereby enabling the preparation of a deployment plan that meets the allocation and connection requirements of each component, (2) allocate resources for a particular deployment plan and release resources when the components or the entire deployment is removed, (3) obtain runtime resource available in the system, and (4) dynamically update the resource consumption data. This section describes how the structure and functionality of *Bulls-Eye* supports these capabilities in the context of Lightweight CCM's Target Manager specification.

3.1 Structure of Bulls-Eye

Figure 2 shows the architecture of *Bulls-Eye*, which implements the CORBA interface in the Target Manager specification. *Bulls-Eye*'s architecture has two parts: (1) a logically centralized[2] service, known as the Target Manager Core (`TM-Core`), which is used by applications and system services to allocate/release resources and (2) multiple monitors (`TM-Monitors`) distributed in the domain to perform resource monitoring and update the `TM-Core`'s model of the amount of resources available.

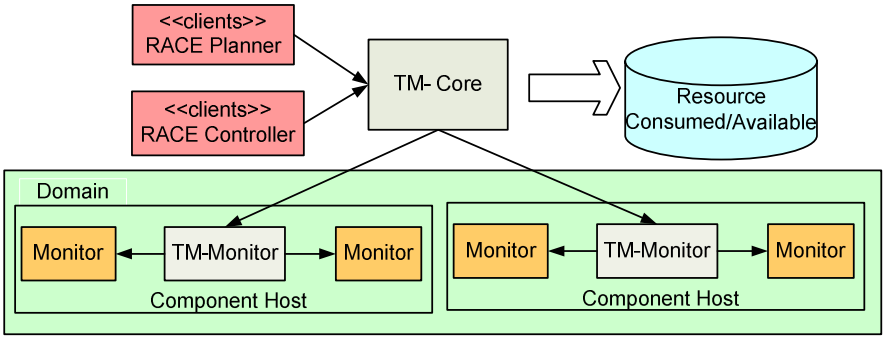


Fig. 2. The *Bulls-Eye* Target Manager Architecture

The *Domain* contains all the elements of a target environment, including the nodes, interconnects between nodes, bridges connecting interconnects, and the set of resources belonging to these elements. A *Domain* is a logical concept, i.e., a single resource or node element can be part of more than one target domain. *Domains* can therefore be structured hierarchically, with a top-level domain containing other *Domains*. Each *Domain* has a `TM-Core` that accumulates the resource information for elements in the target domain.

The `TM-Core` provides a standard set of operations that applications and system services can use to provision available resources *statically* (i.e., prior to system launch) as well as *dynamically* (i.e., during system runtime) in the form of a generic structure known as the *DomainStruct* [4]. This structure describes the contents of the entire target environment by composing data related to available nodes in the

[2] There is only one logical instance of the `TM-Core` in the domain, though it can be replicated to enhance availability and prevent a single point of failure.

network, the connections between nodes, connection between networks, the shared resources among them, and the resources for each element.

A `TM-Monitor` is placed on each node in the target domain to monitor the resource usage in that node. The `TM-Monitor` sends updates periodically to the `TM-Core`, with the current resource utilization/availability on that node. Upon receiving the updates, the `TM-Core` aggregates the data received with previous data and updates its content. *Bulls-Eye* uses standard CORBA request/response calls for this communication.

Bulls-Eye maintains a top-level *Domain* element that contains all the elements of a target domain and is identified by a universally unique identifier (UUID). This *Domain* element is designed so that all possible elements in the target domain can be incorporated, thereby alleviating the need to create separate structures for different types of resources, such as processors, memory, storage, and/or network bandwidth. This design also makes client code flexible by alleviating the need for any specific type of resource in the target domain since it can handle all the varieties of resource elements present.

`TM-Monitors` collect data pertaining to their sub-domain and update the `TM-Cores` with fresh data. Clients are interested in data across sub-domains, so the data from different `TM-Monitors` are aggregated and presented uniformly. To avoid latency issues, the distributed monitors exchange only the data that changed from the previous update. This data is aggregated with the remaining target domain data that are already present.

3.2 Functionality of Bulls-Eye

Bulls-Eye provides the following standard Target Manager operations that can be invoked by clients to provision system resources:

- **Querying static resources.** Developers or planner applications can use `getAllResources()` to obtain the initial static resource availability in the target domain. This operation returns the *Domain* structure that describes the entire target domain resources hierarchically.
- **Querying dynamic resources.** Dynamic resource availability can be returned by `getAvailableResource()`. This operation returns the same *Domain* structure as above, except that the resources reflect their remaining capacity.
- **Committing resources.** A planning application can call `createResourceCommitment()` to commit (i.e., allocate) resources for a deployment plan. This operation creates a `ResourceCommitmentManager` that can commit and release resources for a specific plan. A pool of resources can be specified when a call to `createResourceCommitment()` is made or can be allocated after it is created. An exception is raised if a requested resource cannot be committed.
- **Releasing resources.** When an application or component is deleted all its resources must be released so they can be reallocated to subsequent applications. Applications can release resources by calling `releaseResources()` on the associated `ResourceCommitmentManager`. When a `ResourceCommitmentManager`

is itself deleted via `destroyResourceCommitment()`, all remaining committed resources it holds are released automatically.

- **Updating dynamic resource data.** The target domain data in the `TM-Core` can be updated via `updateDomain()`. The updated information is passed using the *Domain* structure, which is a subset of the higher level domain structure. An enumeration called *DomainUpdateKind* can be used to tell *Bulls-Eye* whether the subset should be added, deleted, or updated.

The *Bulls-Eye* Target Manager functionality plays a key role in the deployment and configuration of enterprise DRE systems. On startup, it reads a standard XML configuration script that describes the resources present in the target domain. The script is prepared by a human or automated domain administrator who understands the initial target domain contents, such as nodes, the interconnects that link them, and the resources contained in them (such as processor capacity, memory capacity, and disk capacity) that are available for application usage. This script is structured according to a standard *DomainStruct* described in Section 3.1. The `TM-Monitor` used to monitor component resources on a node is collocated and started together with its associated `Node Manager`, which is an entity defined by the OMG D&C specification and implemented by *CIAO* as a daemon process running on each node. The `TM-Core` finds the object reference addresses of the various `Node Managers` from a Naming Service and establishes connection with them.

At startup, the `TM-Core` is passed the subset of the entire *Domain*, which it then uses to instructs the `TM-Monitors` on each node which resources to monitor. Each `TM-Monitor` then checks the *Domain* information and reports any discrepancies (such as the hard disk capacity being smaller than the initial domain description or the node is single processor instead of a multiprocessor) to the `TM-Core`. After *Bulls-Eye* starts running, clients can use it to query information about the domain, e.g., *RACE* components can extract domain related information for preparing a deployment plan. Any entity, such as an `Execution Manager`, that deploys plans in the target domain needs to provision resources via *Bulls-Eye* to run applications successfully.

4 Resolving Bulls-Eye Design Challenges

Although the CCM specification defines the interface and the functionality of the Target Manager service it does not prescribe any design details. We therefore faced a number of design challenges when implementing *Bulls-Eye*. This section describes key challenges we encountered, presents our solutions, and outlines how we applied these solutions to the shipboard computing applications supported by the *MLRM* subsystem described in Section 2.

Challenge 1: Integrating Heterogeneous APIs of Multiple Platforms

Context. The resource utilization information provided to clients of *Bulls-Eye* should be consistent, i.e., use similar units/structures. Otherwise, the users of the data will need to convert them manually, which is tedious, error-prone, and can yield redundancies and/or inconsistencies in conversion logic.

Problem → Encapsulating diverse resource utilization APIs on different OS platforms. The target domain of DRE systems typically consists of multiple operating systems, each with its own platform-specific APIs that provide information on resource usage. For example, the Unix/Linux `/proc` file system tracks process resources usage, such as processor, memory, and byes sent/received from network devices. Windows, in contrast, has multiple ways of procuring this information, including (1) using a DLL (PDH.dll) that provides an API for querying resource consumption data or (2) extracting the raw data from the system registry using the HKEY_LOCAL_MACHINE key. The data structures and units returned by platform-specific OS APIs are sufficiently diverse that it is hard to write portable resource management algorithms. Diversity even extends to different versions of the same OS, e.g., performance data in Windows NT 4.0 is contained in a counter named '% Total processor time', which is named '% processor time' in Windows XP. Ideally, the middleware should convert the diverse OS-specific APIs and data into a uniform and consistent format that can be used portably by clients.

Solution → Use the Adapter pattern to adapt diverse API. To mitigate the problem of diverse resource usage APIs and data in *Bulls-Eye*, we used the Adapter pattern [2]. This pattern converts non-standard APIs that extract resource data into the standard interface defined by the Target Manager specification, as shown in Figure 3. The implementation of this interface converts the platform-specific data into a uniform type for storing and disseminating resource usage information to *Bulls-Eye* clients.

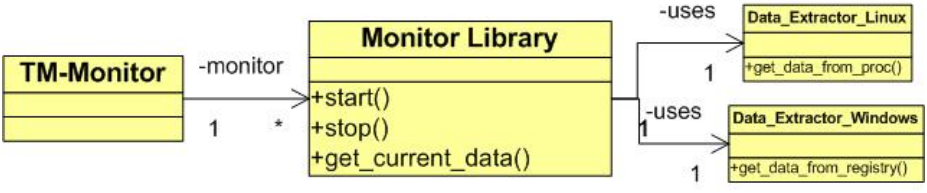


Fig. 3. Using the Adapter Pattern in *Bulls-Eye*

The extraction of resource consumption data is tricky and obtaining accurate values depends on certain optimizations [20], such as keeping `/proc` open between reads and reading data in a block rather than individual characters. Likewise, extracting raw value from the registry is faster on Windows, though it is more complicated to program since it involves parsing a text stream and extracting different structures containing performance counters. Each structure is variable length and contains headers that must be parsed to get details about the data. Although the PDH.dll mentioned above does this parsing automatically it is somewhat slower since it incurs more overhead. In future work we will choose the appropriate solution in terms of performance and simplicity by encapsulating these details in the middleware itself, thereby simplifying client applications.

Applying the solution to the MLRM case study. The current version of the *MLRM* uses nodes with various versions of Linux as the operating system. To function properly, the *MLRM* must monitor the system load as a function of the total percentage of

CPU utilized. A monitor library is therefore written that reads the "`/proc/stat`" file and calculates the processor utilization as a percentage. The `TM-Monitor` loads this monitor library and periodically extracts the system data from it.

Challenge 2: Providing a Common Access Point to Provision Domain Resources

Context. Enterprise DRE systems are often distributed across many entities. The entire application environment is arranged hierarchically, with a top-level domain containing sub-domains, which in turn contain computing nodes connected via bridges and interconnects. Any planner for a target domain will require information of resources contained in the entire domain.

Problem → Monitoring resource utilization of multiple physically distributed resources. Information regarding resource utilization and/or availability of system resources is tracked by multiple monitors that are physically distributed across the domain. If clients were responsible for determining the resource utilization and/or availability of system resources they would have to (1) obtain the location of the monitors, (2) contact the monitors to obtain the resource utilization data, and (3) process the data acquired from multiple monitors to obtain a *global view* of utilization of system resource, which is tedious, error-prone, and non-scalable for each client to perform individually.

Solution → Use distributed monitors to collect data across the domain and consolidate the information via a common access point. To provide a common access point for data in the target domain, *Bulls-Eye* uses the `TM-Monitors` to send data updates at a periodic interval (which can be configured) to a centralized location, the `TM-Core`.[3] `TM-Monitors` are located on each node in the target domain infrastructure shown in Figure 4 and use the platform-independent adapters described in Challenge 1 to extract the resource information. The `TM-Core` serves as a common access point where the clients can obtain the information regarding global resource availability/utilization in a target domain.

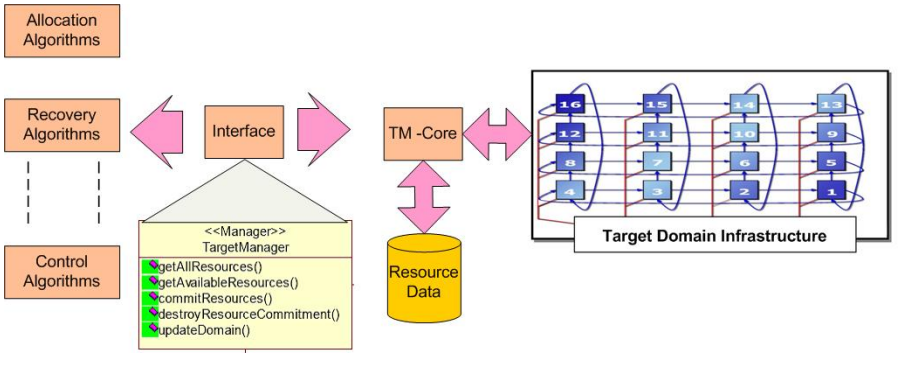


Fig. 4. Providing a Common Access Point to Domain Resource Data

[3] Although with this approach the `TM-Core` is a "single point of failure" it can be replicated to address this issue and to enhance availability.

Applying the solution to the MLRM case study. The *MLRM* `Node Provisioners` use the *DAnCE* `Node Managers` to start application components on each node. The `TM-Monitors` also startup with the `Node Managers` and receive a *DomainStruct* from the `TM-Core` specifying what are resources to monitor. Using this configuration, each `TM-Monitor` monitors the resources and notifies the `TM-Core` periodically when resource availability changes. In the *MLRM*, the `TM-Core` is started in one server and its object reference is stored in the Naming Service. Any client that requires resource data obtains the object reference from the Naming Service and makes calls to the `TM-Core`. The `TM-Monitors` are started along with the `Node Managers` on each node and periodically send data updates on resource usage and availability to the `TM-Core`.

Challenge 3: Presenting Data to Clients with Bounded Response Time in Uniform Structure

Context. An enterprise DRE system typically has many resources present in various forms. For example, a target domain may contain multiple nodes, and each node may contain multiple resources, *e.g.*, a node may contain multiple network cards and thus be connected to many other nodes in the domain. The data from different parts of a domain should be presented in a uniform and aggregated form for planners and clients to manage and use resources effectively.

Problem → Providing aggregated data of entire domain within bounded delay quickly. In a typical enterprise DRE system there may be many domain elements, the data exchanged by these elements may be large, and there significant latency may be incurred when transferring this data. Data updates from distributed monitors will reach the `TM-Core` separately and will only pertain to a subset of the entire target domain. When these updates are merged along with many other updates to the main data store, parsing and locating the data store for a particular resource may slow down response time considerably. As a result, there is a need for efficient and scalable algorithms to merge the data. Moreover, changes in target domain resource usage should be disseminated within bounded delay so that planners and clients can use the data for time-sensitive planning and resource management.

Solution → Combination of heap-sort and timer based aggregation algorithm. To solve the problem of aggregating domain data quickly, *Bulls-Eye* combines the following approaches:

- It maintains a cache of the last update sent to the `TM-Core` to optimize communication and minimize unnecessary CPU and network processing. When *Bulls-Eye* gets fresh data from the underlying component it compares the data received with the cached data and only sends an update if it detects differences. For example, no update is sent to the `TM-Core` if a particular reading informs the `TM-Monitor` that memory usage has not changed since the last update.
- The *Bulls-Eye* time-based data aggregation mechanism uses a heap-sort implementation [22] that provides O (log n) time complexity in the worst case. Each resource entity is labeled with a unique identity that is placed along with pointers to the actual data structure in a heap. Once updated data is received from the

monitors, it is stored in a cache. A timer then fires at regular intervals and the cache is examined for any outstanding data update. The resource entity id of the update is located in the heap in O (log n) time and its corresponding data structure is updated in constant time via the pointer into the cache.

Applying the solution to the MLRM case study. The operational context of a ship-board computing environment evolves continuously, e.g., it needs to satisfy changing mission requirements and adapt to transient overload and node failures. These changes provoke a reaction in the control algorithms that drive redeployment of components in the system based on domain resource usage. The aggregation algorithm used by *Bulls-Eye* (1) improves its responsiveness and allows it to collaborate faster with clients (such as the *MLRM* subsystem and its applications) and (2) helps reduce the costs associated with redeploying the system, thereby devoting more CPU and I/O processing to accomplishing mission tasks and meeting system deadlines.

Challenge 4: Using Multiple Configurable Monitor Components to Extract Variety of Data

Context. There are many types of elements in a target domain for an enterprise DRE system. Each element can have its own `TM-Monitor` tracking its resource usage. There can also be multiple monitors for the same resource to record data from various perspectives. For example, system utilization can be monitored via the load average value on a CPU, the percentage of CPU usage, or the average amount of slack time in each cycle.

Problem → Configuring and using different resource data extraction mechanisms. A variety of monitor elements may record resource data using different mechanisms, e.g., in some platforms (e.g., Windows) vendor-supplied software may track processor consumption, whereas in other platforms (e.g., Linux) developers may need to write code to access the data. In yet other platforms (e.g., VxWorks), developers may want to use specialized third-party hardware monitoring utilities. Since each mechanism may use different APIs *Bulls-Eye* should be capable of integrating and configuring different types of monitors with minimal developer effort.

Solution → Monitor algorithm wrapped within a shared library and configured using initial domain data. *Bulls-Eye* uses the Strategy pattern [2] to encapsulate each monitor algorithm in a DLL whose common interface defines lifecycle activities and data extraction methods. When *Bulls-Eye* starts up, the `TM-Core` sends each `TM-Monitor` the initial *Domain* data, which contains the DLL name and specific configuration details for each resource element it monitors. The `TM-Monitor` then uses the Component Configurator pattern [11] to link the DLL dynamically and update a map that associates each resource element to its library name. The `TM-Monitor` calls each library to start/stop monitoring and to periodically get current data. It also combines data from each library into one *Domain* structure before uploading it to the `TM-Core`, thereby making Bull-Eye's monitoring capabilities more extensible.

Applying the solution to the MLRM case study. The *MLRM* subsystem uses *Bulls-Eye* to monitor each node's total CPU utilization along with the CPU percentage used by each application under its control. This design uses one DLL (system monitor) that

tracks the CPU load for the entire node and another DLL (application monitor) that tracks the load on the applications controlled by the *MLRM*. The `TM-Core` creates the *Domain* data containing the corresponding monitoring library names and the process ids of the relevant applications that it receives from *DAnCE*. It passes this structure to the `TM-Monitors` that load the libraries and send the process ids to the corresponding application monitor. The `TM-Monitor` then starts the monitors, periodically collects data from the two monitors, combines them, and pushes the data to `TM-Core` if it differs from the previous update.

5 Experimental Evaluation of *Bulls-Eye*

This section first outlines the testbed that provides the infrastructure for a representative enterprise DRE system from the domain of shipboard computing. We then describe our experiments and analyze the results obtained to evaluate the performance of *Bulls-Eye*.

5.1 Hardware/Software Testbed and Experiment Configuration

Our experiments were performed on the ISISLab testbed at Vanderbilt University (www.dre.vanderbilt.edu/ISISLab), which is a testbed of nodes and bridges that can be arranged in many configurations. ISISLab contains 6 Cisco 3750G-24TS switches, 1 Cisco 3750G-48TS switch, 4 IBM Blade Centers each consisting of 14 blades (for a total of 56 blades), 4 gigabit network IO modules and 1 management module. Each blade has two 2.8 GHz Xeon CPUs, 1GB of RAM, 40GB HDD, and 4 independent 1 Gbps network interfaces. The hardware configuration for our *Bulls-Eye* experiments consists of six blade nodes acting as the target domain. Each node ran Redhat Fedora Core release 4 operating system running in real-time scheduling mode.

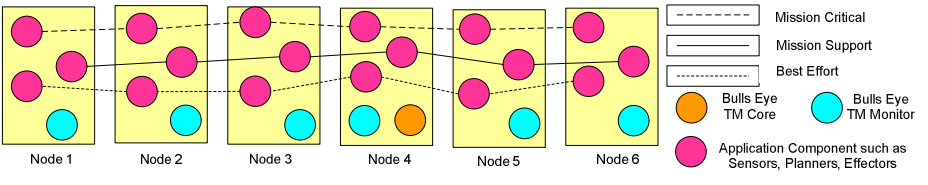


Fig. 5. Operational Strings in the Testbed

Figure 5 shows our enterprise DRE system test configuration, which contained three operational strings [13], each with six application components that are allocated to six nodes in ISISLab. The application components were implemented using component workload emlators (CoWorkErs) [18]. Real-time QoS properties and requirements of these operational strings are specified by their relative priority and end-to-end deadline, respectively. The three operational strings were composed of one mission-critical string, one mission-support string, and one best-effort operational string. The mission-critical operational string was configured with the highest priority (priority 80), followed by the mission-support (priority 60) and best-effort (priority 20)

operational strings. An end-to-end deadline of 500 ms was specified for the mission-critical operational string.

To evaluate *Bulls-Eye*, we first deployed the mission-critical operational string in ISISLab at time T = 40 sec. We deployed the mission-support operational string at time T = 110 sec. Finally, we deployed the best-effort operational string at time T = 170 sec. *Bulls-Eye* monitored the net processor utilization at each node and each component within the domain. It also monitored the end-to-end execution time of the mission-critical operational string. Since *Bulls-Eye* is implemented as a component, we monitored the resource utilization of *Bulls-Eye* itself to determine its overhead.

Our experiments used the *Resource Allocation and Control Engine (RACE)* [13] in conjunction with *Bulls-Eye* to ensure the end-to-end execution time of the mission-critical operational string was below its end-to-end deadline. *RACE* uses allocation and control algorithms whose behavior depends on the characteristics of applications it deploys. Inputs to *RACE* include (1) resource requirements of each component in the operational string, (2) end-to-end deadlines of operational strings, which are obtained from the component descriptors, and (3) runtime resource utilization information provided by *Bulls-Eye*. Based on the current resource availability and resource components' requirements, *RACE* generates a deployment plan that maps components to physical nodes. *DAnCE* then uses this plan to deploy application components to the designated nodes in the target domain.

5.2 Analysis of Results

This section analyzes the results of running the experiment described in Section 5.1 on ISISLab. We used end-to-end execution time of the mission-critical operational string as a metric to evaluate the utility of *Bulls-Eye*, as well as and the resources usaged by *Bulls-Eye* as a measure of the infrastructure overhead. Resource utilization information collected by *Bulls-Eye* for the six nodes is shown in Figures 6 and 7 as a function of time. End-to-end execution time of the mission-critical operational string is shown in Figure 8 as a function of time. Since *Bulls-Eye* was deployed on node 6, Figure 7 also captures the overhead of the *Bulls-Eye* infrastructure.

As shown in Figures 6, 7, and 8, when the best-effort operational string is deployed at T = 170 sec, the net processor utilization of the nodes increased above the utilization setpoint of 70% recommended for rate monotonic scheduling [17]. At the same time, the end-to-end execution time of mission-critical operational string increases above its 500 ms deadline, which indicates that the increase in execution time of the mission-critical operational string stemmed from over-utilization of CPU resources.

The resource utilization information collected by *Bulls-Eye* serves as input to *RACE*, which then performs adaptive system control modifications, such as modifying operating system priority, scheduler class, and/or tearing down lower priority operational strings. In our experiment, *RACE* tears down the best-effort operational string to meet the QoS requirements of the higher priority mission-critical operational strings. As a result of *RACE*'s adaptive control actions, the end-to-end execution time of the mission-critical operational string is restored to below its deadline, as shown in Figure 8. Figure 7 shows that the infrastructure overhead due to *Bulls-Eye* itself is insignificant compare to the network resource utilization.

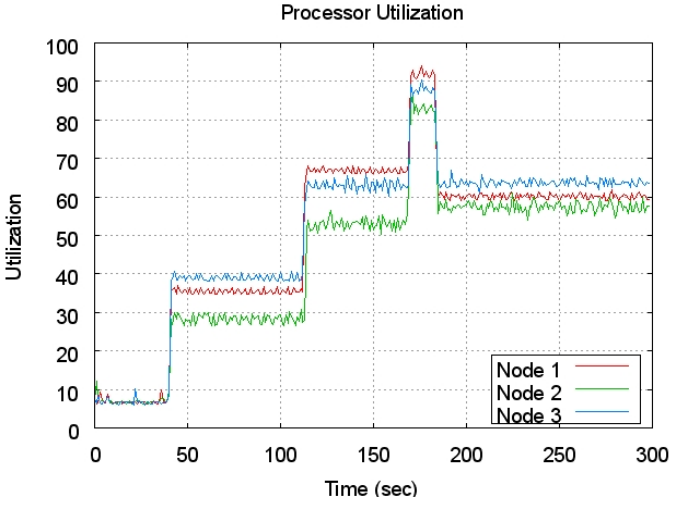


Fig. 6. Processor Utilization of Node 1, 2, and 3

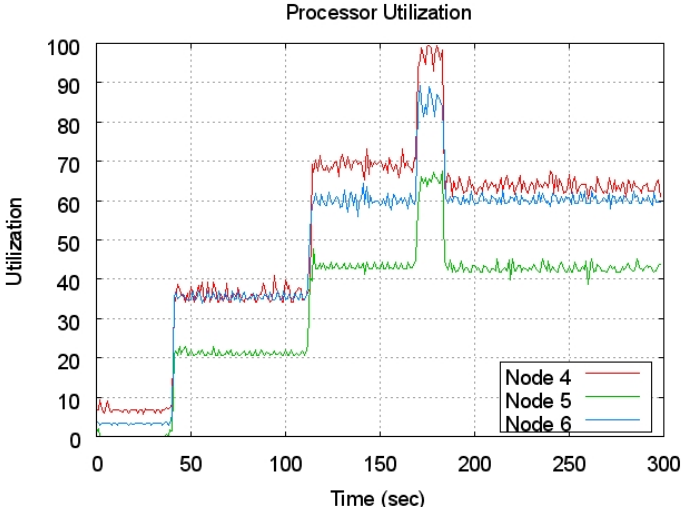


Fig. 7. Processor Utilization of Node 4, 5, and 6

Without a resource provisioning service like *Bulls-Eye*, over-utilization of system resources could go unnoticed. Resource utilization information is a key input to a control framework like *RACE*, whose algorithms require resource utilization information on multiple types and instances of resources from the domain. A resource provisioning framework such as *Bulls-Eye* is therefore essential to effective adaptive resource management for enterprise DRE systems.

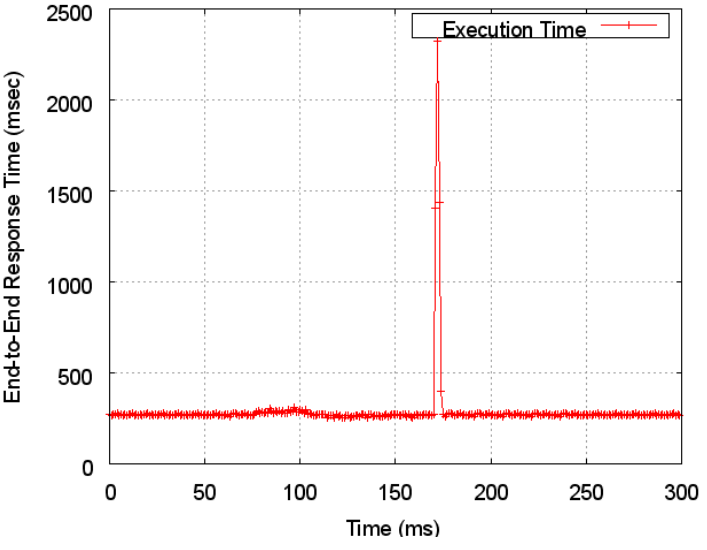


Fig. 8. End-to-End Execution Time of the Mission-Critical Operational String

6 Related Work

This section compares our work on *Bulls-Eye* with related work in the domain of resource provisioning.

The CMU Resource Monitoring System (ReMoS) [8] is service that allows network-aware applications to obtain relevant information about the bandwidth and latency of a specific flow, which is an application level connection between a pair of computation nodes. It also answers queries about the network topology. ReMoS uses two abstraction levels: *explicit management* of resource sharing and *statistical measurement*. Its flows abstraction captures the communication between nodes and its topologies abstraction provides a logical view of network connectivity. ReMoS measurements are made at the network level, so it provides information for use in sharing of resources. *Bulls-Eye*, in contrast, focuses on the resource availability for component assemblies, rather than the network level. This focus requires the aggregation of data so that decisions regarding whole units/assemblies can be taken. Resource provisioning, synchronizing multiple planners, and matching of component requirements to target domain availability are other key concerns for *Bulls-Eye*, which acts as a common service for resource provisioning at multiple layers.

The BBN Resource Status Service (RSS) [9] is another multi-layer resource monitoring service. RSS consists of monitors (known as "condition objects") that are distributed to nodes in a network and which communicate to acquire the required data. In addition, RSS aggregates data of various resources, such as processor load average and memory consumption. Whereas the RSS is based on a non-standard interface, *Bulls-Eye* supports the OMG Lightweight CCM Target Manager specification, which defines standard interfaces that third-party providers can use to integrate their monitoring mechanisms. The Lightweight CCM Target Manager specification (and thus

Bulls-Eye) also supports resource provisioning by providing a common point to commit and release resources for different plans deployed, which is not supported by RSS.

The Globus Toolkit [10] provides a number of resource provisioning services that focus on monitoring, management, scheduling, and coordination of different computations in a computing grid. It also has tools for transmitting and managing large amounts of data useful to grid-based applications. *Bulls-Eye* does itself does not manage applications of a distributed environment, relying on other services in *CIAO*, such as *DAnCE* and *RACE* for these capabilities. Instead *Bulls-Eye* focuses on the collection, aggregation, and presentation of resource information in a timely manner. *Bulls-Eye* focuses on deployment and configuration of component-based applications and has features to support real-time QoS policies for mission-critical DRE systems.

[11] proposes an integrated architecture for managing dependencies uniformly in distributed component-based systems. It allows developers to present dependencies between components; instantiates component based applications, and manages hardware resources in the distributed system. It therefore has a *resource management service* that is similar to *Bulls-Eye*, i.e., it uses distributed monitors to acquire local status information and aggregates the information on a central server. [11] focuses largely on the allocation and running of the applications, however, whereas *Bulls-Eye* is more generic and supports standard interfaces for plugging in multiple types of resource monitors. In addition, [11] does not deal with the resource provisioning aspects supported by *Bulls-Eye*.

[12] implements a resource monitoring service similar to *Bulls-Eye*, but with a focus on collecting resource data to create a forecasting model that provides process schedulers with resource trends so that they can schedule more efficiently. *Bulls-Eye* is different, however, since it collects resource information at runtime at a finer level of granularity, i.e., it collects data for each participating process and thread and passes the data to *RACE* [13], which uses multiple allocation and control algorithms to (re)deploy and (re)configure the applications with the goal of maintaining stringent Qos requirements. Since *Bulls-Eye* is targeted for enterprise DRE systems, it focuses on the latency of the data collection and provides a standard interface that can be composed with resource management frameworks, such as *RACE*.

IBM's Tivoli [21] implements a resource provisioning system that automatically triggers the dynamic provisioning of resources (such as storage and services) to processes that need them the most to handle client demand. It mainly focuses on the web-based enterprise application domain, such as applications for e-commerce websites. *Bulls-Eye*, in contrast, is designed to provision DRE system resources and thus deals with more stringent real-time QoS requirements.

7 Concluding Remarks

This paper motivated and described *Bulls-Eye*, which is an implementation of the Lightweight CCM Target Manager specification we developed to support resource provisioning for enterprise DRE systems. We discussed the design challenges faced when developing *Bulls-Eye* and applying it to a shipboard computing system and showed how our solutions helped resolve these challenges. We also presented the results of experiments that showed how *Bulls-Eye* simplified resource management and helped automate adaptations in the face of dynamic operating condition changes.

The following are lessons learned from our work on *Bulls-Eye* and its use in the Multi-Layer Resource Manager (*MLRM*) subsystem shipboard computing case study:

- Building enterprise DRE systems whose operational requirements and environment change frequently necessitates the dynamic monitoring of domain resources and requires a framework to provide resource availability information to enable the automated (re)deployment and (re)configuration of components throughout the (possibly heterogeneous) system.
- The CCM Target Manager specification strikes an effective balance between flexibility and efficiency by keeping client code considerably simpler and supporting dynamic updates and system (re)deployment and (re)configuration.
- Applying patterns to *Bulls-Eye* helped ensure that its design used best practices associated with solving recurring problems and leveraging the experience of experienced developers. Patterns applied to *Bulls-Eye* included Adaptor, Strategy, and component configurator.
- *Bulls-Eye's* aggregation algorithms helped improve overall system performance, which in turn led to clients responding to changes in application operating conditions or policies more portably and effectively.
- The decoupling of the `TM-Core` from the `TM-Monitors` helped increase the performance and scalability of *Bulls-Eye* by exploiting the computing power of distributed nodes across the target domain and distributing complex processing over multiple nodes.
- *Bulls-Eye* played an important role in allocating components to different nodes across the domain. It helped allocation algorithms generate deployment plans that optimize global resource usage. *Bulls-Eye's* monitoring capabilities are also useful for the control algorithms in *RACE* that (re)allocate component properties in response to changing operating environment.

Although our experience with *Bulls-Eye* was generally positive, we also identified the following limitations with various design/architectural issues that affect end-to-end application QoS in our *MLRM* case study:

- The `TM-Core` has only one instance in the entire target domain, which is a single point failure and a potential bottleneck in largely target domain configurations.
- *Bulls-Eye* is not yet fully scalable with respect to the resources it manages, e.g., it creates monitors for each resource type and node added to the domain, which increases the data volume of the domain proportionally, instead of being managed via sub-domains with distributed storage.
- *Bulls-Eye* serially establishes communication with the various `Node Managers` in the target domain during start up, which is not scalable and may cause deadlock.
- *Bulls-Eye* receives updates from various monitors, which can cause performance bottlenecks for node that runs the `TM-Core`.

Our future efforts therefore focus on adding the capabilities listed above to further enhance *Bulls-Eye*. In particular, we are exploring the use of group communication and distributed replica management to addresses *Bulls-Eye's* current limitations.

References

1. G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A. Gokhale, "*DAnCE*: A QoS-enabled Component Deployment and Configuration Engine", Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France, November 28-29, 2005.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994.
3. Object Group Management (2003, May), "Light Weight CORBA Component Model" Revised Submission, Ed. OMG Document realtime/03-05-05.
4. Object Management Group: "Deployment and Configuration" Adopted Submission, OMG Document ptc/03-07-08 edn. (2003).
5. Object Management Group (2002, Aug). "Real-time CORBA Specification". Ed. OMG Document formal/02-08-02.
6. D. Schmidt, M. Stal, H. Rohert, and F. Buschmann, "Pattern-Oriented Software Architecture: Patterns for Networked and Concurrent Objects", Wiley and Sons, 2000.
7. D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," CrossTalk, Nov, 2001.
8. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, D. Sutherland, "ReMoS: A Resource Monitoring System for Network-Aware Applications" Carnegie Mellon School of Computer Science, CMU-CS-97-194.
9. J. Zinky, J. Loyall, and R. Shapiro "Runtime, Performance Modeling and Measurement of Adaptive Distributed Object Applications," Proceeding of International Symposium on Distributed Object and Applications, DOA 2002, Irvine, CA, Oct 28-30, 2002.
10. I. Foster and C. Kesselman."Globus: A Metacomputing Infrastructure Toolkit" Intl. Journal of Supercomputer Applications and High Performance Computing, 11(2):115-128, 1997.
11. F. Kon, T. Yamane, C. Hess, R. Campbell, and M. Mickunas, "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems," Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas, Jan, 2001.
12. R. Wolski, "Experiences with Predicting Resource Performance On-line in Computational Grid Settings" ACM SIGMETRICS Performance Evaluation Review, Volume 30, Number 4, pp 41-49, Mar, 2003.
13. N. Shankaran, J. Balasubramanian, D. Schmidt, G. Biswas, P. Lardieri, E. Mulholland, and T. Damiano, "A Framework for (Re)Deploying Components in Distributed Realtime and Embedded Systems", Dependable and Adaptive Distributed Systems Track of the 21st ACM Symposium on Applied Computing, Dijon, France, Apr 23-27, 2006.
14. W. Roll, "Towards Model-Based and CCMBased Applications for Real-Time Systems," in Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Hokkaido, Japan, IEEE/IFIP, May 2003.
15. S. Murat, Bicer, F. Pilhofer, G. Bardouleau, and J. Smith, "Next Generation Architecture for Heterogeneous Embedded Systems", International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA). Jun 2003, Las Vegas, NV, USA.
16. S. Paunov and D. Schmidt, "RepoMan: A Component Repository Manager for Enterprise Distributed Real-time and Embedded Systems," Proceedings of the 44th ACM Southeast Conference, Melbourne, FL, Mar 10-12, 2006.
17. J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," In Proceedings of the 10th IEEE Real-time Systems Symposium (RTSS 1989), Santa Monica, Dec 1989.

18. J. Hill, J. Slaby, S. Baker, and D. Schmidt, "Evaluating Enterprise Distributed Real-time and Embedded System Quality of Service with System Execution Modeling Tools," Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia, 16-18 Aug 2006.
19. G. Heineman and B. Councill, "Component-Based Software Engineering: Putting the Pieces Together", Addison-Wesley, Reading, Massachusetts, 2001.
20. C. Smith and D. Henry, "High-Performance Linux Cluster Monitoring Using Java," Proceedings of the 3rd Linux Cluster International Conference, 2002.
21. E. Manoel et al. "Provisioning On Demand: Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator", IBM, International Technical Support Organization, Dec 2003.
22. Ronald E. Barkley and T. Paul Lee, "A Heap-based Callout Implementation to Meet Real-time Needs", Proceedings of the USENIX Summer Conference, June 1988, pp. 213-222.

Erratum

LNCS 4276 Editorial

In an earlier version by mistake the volume editors have been stated instead of the authors of each paper. The current version is the correct source for any reference made to a paper included in the OTM 2006 Proceedings LNCS 4275-4278.

Author Index

Aberer, Karl I-516
Abou-Tair, Dhiah el Diehn I. I-983
Adams, Michael I-291
Adamus, Radoslaw II-1290
Adjiman, P. I-698
Afsarmanesh, Hamideh I-91
Aït-Ameur, Yamine I-704
Albertoni, Riccardo I-1020
Alcaide, C. II-1181
Ali, Nour II-1633
Alia, Mourad II-1686
Allen, Gabrielle II-1119
Alvarez, Francisco I-855
Amann, Bernd I-327
Ardagna, Danilo I-807, II-1273

Balasubramanya, Magesh II-1576
Balis, Bartosz II-1305
Baloian, Nelson I-679
Banos, Evangelos I-975
Barros, Alistair I-145
Bassiliades, Nick I-975
Baude, Françoise II-1191
Baytelman, Felipe I-679
Behnel, Stefan II-1522
Ben Lakhal, Neila I-163
Benatallah, Boualem I-109
Benavides Navarro, Luis Daniel II-1449
Bergamaschi, Sonia I-909
Berlik, Stefan I-983
Bernard, Guy II-1668
Bertolo, Stefano II-1125
Bézivin, Jean I-863
Bhiri, Sami I-408
Bittner, Sven II-1503
Boboila, Marcela S. II-1234
Böhm, Klemens I-444, I-498
Borz, Luca II-1336
Bosque, José Luis II-1412
Botía Blaya, Juan A. II-1325
Boucelma, Omar I-377
Bouroche, Mélanie II-1722
Bouzeghoub, Mokrane I-237
Bræk, Rolv II-1613
Brandão, José Eduardo M.S. I-462
Brasileiro, Francisco II-1705
Brockmans, Saartje I-901
Bryl, Volha I-533
Bubak, Marian II-1305
Buche, Patrice I-891
Buchmann, Alejandro II-1522
Buchmann, Erik I-498

Cahill, Vinny II-1722
Campos, Maria Luiza M. I-551
Cappiello, Cinzia I-807
Caromel, Denis II-1191
Carvalho, Nuno II-1485
Casanova, Marco A. I-756
Cattaneo, Gianpiero II-1152
Chabridon, Sophie II-1668
Chao, Yi II-1119
Charfi, Anis I-183
Chatalic, P. I-698
Chateigner, Lydialle II-1668
Chung, Seokkyung I-1092
Cirne, Walfredo II-1705
Clematis, A. II-1132
Cluet, Sophie I-72
Collet, Christine I-391
Colonna, François-Marie I-377
Combi, Carlo I-201
Conforti, Eugenio I-219
Constantin, Camelia I-327
Corana, A. II-1132
Corona, Grégory I-72
Corrales, Juan Carlos I-237
Coulson, Geoff II-1522
Creel, James I-605
Cristea, Valentin II-1234
Cunningham, Raymond II-1722

D'Agostino, D. II-1132
Dagdeviren, Orhan II-1422
Daniel, Florian I-201
de Buenaga, Manuel I-855
De Martino, Monica I-1020
De Meo, Pasquale I-967

de Moor, Aldo I-738
Debarbieux, Denis I-941
Delicato, Flávia C. I-551
Deng, Yu I-1039
Dennunzio, Alberto II-1152
Di Quarto, Gabriele I-967
Díaz, Manuel II-1181, II-1351
Dibie-Barthélemy, Juliette I-891
Didonet Del Fabro, Marcos I-863
Dikenelli, Oguz I-927
Doussot, David I-891
Dragut, Eduard I-882
Dumas, Marlon I-145
Dworaczyk, Blake I-605

Edmond, David I-291
Eliassen, Frank II-1686, II-1825
Erciyes, Kayhan II-1422
Erdur, Riza Cenk I-927
Evans, Reuben I-661

Fahringer, Thomas II-1305
Falconer, Sean M. I-1075
Farina, Fabio II-1152
Felber, Pascal II-1541
Ferscha, Alois II-1434
Feschet, Fabien II-1213
Fincke, Tonio II-1388
Fireman, Daniel II-1705
Flissi, Areski II-1402
Folliot, Bertil II-1790
Fraga, Joni da Silva I-462
Francalanci, Chiara I-807
Fricke, Rolf II-1686
Friedman, Roy II-1435

Gaaloul, Walid I-408
Gal, Avigdor I-360
García Clemente, Félix J. II-1325
Garruzzo, Salvatore I-949
Geoffray, Nicolas II-1790
Gianuzzi, V. II-1132
Giorgini, Paolo I-533
Giunta, Gabriele II-1273
Goasdoué, F. I-698
Godart, Claude I-408
Gómez Skarmeta, Antonio F. II-1325
Grace, Paul II-1522
Grigori, Daniela I-237
Groppi, Annalisa I-807
Gross-Amblard, David I-327
Gruber, Olivier II-1772
Gueant, Pierre II-1203
Gümüs, Özgür I-927
Günther, Christian W. I-309
Guzy, Krzysztof II-1305

Haase, Peter I-901
Habegger, Benjamin I-941
Hacid, Mohand-Saïd I-109
Hadad, Erez II-1435
Haemmerlé, Ollivier I-891
Hakkarainen, Sari I-625
Halashek-Wiener, Christian I-722
Hall, Richard S. II-1772
Han, Xiao II-1263
Hassan, Mahbub I-109
Hauck, Franz J. II-1739
Herrero, Pilar II-1203
Herrmann, Peter II-1613
Higgins, Michael I-569, II-1576
Hinze, Annika I-643, I-661, II-1503
Holloway, Seth I-587
Holstius, David II-1576
Horn, Geir II-1686
Huang, Jin II-1376
Huedo, E. II-1143

Iannone, Luigi I-1058
Ibrahim, Noha II-1467
Ingraffia, Nunzio II-1273
Iordache, George V. II-1234

Jacob, Joseph C. II-1119
Jaeger, Michael A. II-1807
Jakob, Wilfried II-1252
Jang, Julian I-426
Jean, Stéphane I-704
Jin, Hai II-1376
Julien, Christine I-587
Jun, Jongeun I-1092
Junmanee, Saijai I-643

Kaczmarski, Krzysztof II-1290
Kantere, Verena I-17
Kapitza, Rüdiger II-1739
Kardas, Geylani I-927
Katakis, Ioannis I-975
Katz, Daniel S. II-1119
Khan, Mohammad Ullah II-1686

Kiringa, Iluju I-17
Kobayashi, Takashi I-163
Koshutanski, Hristo II-1336
Kowalski, Tomasz II-1290
Kraemer, Frank Alexander II-1613
Kropf, Peter II-1541
Kuliberda, Kamil II-1290
Kummer, Raphaël II-1541

Lamarre, Philippe I-36
Larson, Nathan I-360
Lassen, Kristian Bisgaard I-127
Lau, Lydia I-772
Lawrence, Ramon I-882
Le, Duc Minh I-772
Le Mouël, Frédéric II-1467
Leal, Luciana N. I-551
Levashova, Tatiana I-1012
Leymann, Frank I-2
Leyton, Mario II-1191
Li, Du I-605
Li, Kenli II-1263, II-1315
Li, Peggy P. II-1119
Lima, Aliandro II-1705
Liu, Min II-1315
Llopis, L. II-1181
Llorente, I.M. II-1143
Lucas, Peter I-569, II-1576
Lundqvist, Magnus I-1012

Mafra, Paulo Manoel I-462
Malgouyres, Rémy II-1213
Manaskasemsak, Bundit II-1223
Marín, Mauricio II-1388
Márquez, A. II-1181
Martinelli, Fabio II-1336
Martínez Pérez, Gregorio II-1325
Maslov, Dmitri I-1075
McArthur, Greg I-17
McLeod, Dennis I-1092
Meersman, Robert I-738
Merle, Philippe II-1402
Merlo, A. II-1132
Mezini, Mira I-183
Migliavacca, Matteo II-1594
Millán, Carlos II-1633
Mirandola, Raffaela II-1273
Modafferi, Stefano I-219
Montero, R.S. II-1143
Montes, Jesús II-1203
Morales, Mario I-679
Mori, Paolo II-1336
Mühl, Gero II-1807
Mulugeta, Mesfin II-1650
Muñoz Ortega, Andrés II-1325
Murphy, Amy L. II-1594
Musunoori, Sharath Babu II-1825
Mylopoulos, John I-17, I-533

Nagypál, Gábor I-791
Nepal, Surya I-426
Ng, Kam-Wing II-1361

O'Connor, Neil II-1722
O'Riain, Sean I-818
Obelheiro, Rafael R. I-462
Önal, Ata I-927
Ooi, Beng Chin I-54

Palmisano, Ignazio I-1058
Parsia, Bijan I-722
Parzyjegla, Helge II-1807
Pashkin, Michael I-1012
Pastor, Luis II-1412
Payli, Reşat Ümit II-1422
Pereira, José II-1485
Pérez, María S. II-1203
Pernici, Barbara II-1273
Picco, Gian Pietro II-1594
Pierra, Guy I-704
Pires, Paulo F. I-551
Pop, Florin II-1234
Porter, Barry II-1522
Pozzi, Giuseppe I-201
Proper, H.A. (Erik) I-345

Qi, Xuesheng II-1263
Qin, Yunchuan II-1315
Quattrone, Giovanni I-967
Quiané-Ruiz, Jorge-Arnulfo I-36
Quilici, Romain II-1191
Quinte, Alexander II-1252

Rafaelsen, Hans Ole II-1825
Ramos, Isidro II-1633
Redavid, Domenico I-1058
Reichert, Manfred I-273, I-309
Reichle, Roland II-1686
Reynolds, Vinny II-1722

Rice, Julia E. I-1039
Rinderle, Stefanie I-273, I-309
Robles, Oscar D. II-1412
Rodrigues, Luís II-1485
Rodríguez, Andrea II-1388
Rodríguez, Ángel II-1412
Román, Carlos II-1388
Roman, Gruia-Catalin II-1594
Romero, Sergio II-1351
Rosaci, Domenico I-949
Roth, Stuart I-569
Rouached, Mohsen I-408
Rousset, M.-C. I-698
Roussopoulos, Nick I-480
Roy, Nilabja II-1843
Rozkwitalski, Kuba II-1305
Rubio, Bartolomé II-1351
Rungsawang, Arnon II-1223

Sadaoui, Samira II-1757
Sáenz, Fernando I-855
Sala, Antonio I-909
Sam, Yacine I-377
Sánchez, Alberto II-1203
Satoh, Ichiro II-1555
Schill, Alexander II-1650
Schmidt, Andreas I-995
Schmidt, Douglas C. II-1843
Schmidt, Holger II-1739
Schmidt, Roman I-516
Schwarz, Peter I-1039
Sebei, Imen I-72
Semeraro, Giovanni I-1058
Senart, Aline II-1722
Senn, Jeff I-569
Shankaran, Nishanth II-1843
Sharifimehr, Nima II-1757
Simon, L. I-698
Simperl, Elena Paslaru Bontas I-836
Sirin, Evren I-722
Söldner, Guido II-1739
Soler, Enrique II-1181, II-1351
Spyns, Peter I-738, I-818
Storey, Margaret-Anne I-1075
Strasunskas, Darijus I-625
Stratan, Corina II-1234
Stuckenschmidt, Heiner I-901
Stucky, Karl-Uwe II-1252
Subieta, Kazimierz II-1290
Sun, Aixin I-109
Süß, Wolfgang II-1252
Südholt, Mario II-1449

Tan, Kian-Lee I-54
Tang, Xiaoyong II-1263, II-1315
Tempich, Christoph I-836
ter Hofstede, Arthur H.M. I-145, I-291
Thomas, Gaël II-1790
Troya, José M. II-1351
Truong, Hong-Linh II-1305
Tsoumakas, Grigorios I-975
Tsoumakos, Dimitrios I-480

Unal, Ozgul I-91
Ursino, Domenico I-967
Uthayopas, Putchong II-1223

Vaccarelli, Anna II-1336
Valduriez, Patrick I-36, I-863
van Bommel, P. I-345
van der Aalst, Wil M.P. I-127, I-291, I-309, I-408
van der Weide, Th.P. I-345
van Gils, B. I-345
Vandenbussche, Jan I-738
Vanderperren, Wim II-1449
Vaquero, Antonio I-855
Vargas-Solar, Genoveva I-391
Vázquez-Poletti, J.L. II-1143
Verheecke, Bart II-1449
Vieira, Tatiana A.S.C. I-756
Vlahavas, Ioannis I-975
Vodislav, Dan I-72
von der Weth, Christian I-444, I-498
Vu, Thi-Huong-Giang I-391

Weis, Torben II-1807
Widdows, Dominic I-569, II-1576
Wierzbicki, Adam II-1163
Wislicki, Jacek II-1290
Wombacher, Andreas I-255, I-273
Wong, Sze-Wing II-1361
Wu, Ji I-54

Xiao, Degui II-1315
Xie, Xia II-1376

Yahav, Inbal I-360
Yang, Jing II-1315
Yang, Yi I-605

Yokota, Haruo I-163
Yoldas, Ümit I-791

Zaha, Johannes Maria I-145
Zhang, Qin II-1376
Zhou, Qingqing I-17
Zhou, Yongluan I-54
Zhu, Chunling II-1263
Zhu, Xilu II-1263
Zic, John I-426
Zrour, Rita II-1213
Zurita, Gustavo I-679