Lecture Notes in Computer Science 2584

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

André Schiper Alex A. Shvartsman
Hakim Weatherspoon Ben Y. Zhao (Eds.)

Future Directions in Distributed Computing

Research and Position Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

André Schiper
École Polytechnique Fédérale de Lausanne, Faculté Informatique et Communication
IN-Ecublens, 1015 Lausanne, Switzerland
E-mail: andre.schiper@epfl.ch

Alex A. Shvartsman
University of Connecticut, Computer Science and Engineering
Unit 3155, Storrs, CT 06269, USA
E-mail: aas@cse.uconn.edu and alex@theory.lcs.mit.edu

Hakim Weatherspoon
Ben Y. Zhao
University of California at Berkeley, Computer Science Division
447/443 Soda Hall, Berkeley, CA 94704-1776, USA
E-mail: {hweather, ravenben}@cs.berkeley.edu

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): C.2.4, D.1.3, D.2.12, D.4.3-4, F.1.2

ISSN 0302-9743
ISBN 3-540-00912-4 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergraphik
Printed on acid-free paper SPIN: 10872417 06/3142 5 4 3 2 1 0

Preface

Every year we witness acceleration in the availability, deployment, and use of distributed applications. However building increasingly sophisticated applications for extant and emerging networked systems continues to be challenging for several reasons:

- Abstract models of computation used in distributed systems research often do not fully capture the limitations and the unpredictable nature of realistic distributed computing platforms;
- Fault-tolerance and efficiency of computation are difficult to combine when the computing medium is subject to changes, asynchrony, and failures;
- Middleware used for constructing distributed software does not provide services most suitable for sophisticated distributed applications;
- Middleware services are specified informally and without precise guarantees of efficiency, fault-tolerance, scalability, and compositionality;
- Specification of distributed deployment of software systems is often left out of the development process;
- Finally, there persists an organizational and cultural gap between engineering groups developing systems in a commercial enterprise, and research groups advancing the scientific state-of-the-art in academic and industrial settings.

The objectives of this book are: (1) to serve as a motivation for defining future research programs in distributed computing, (2) to help identify areas where practitioners and engineers on the one hand and scientists and researchers on the other can improve the state of distributed computing through synergistic efforts, and (3) to motivate graduate students interested in entering the exciting research field of distributed computing.

The title of this volume, *Future Directions in Distributed Computing*, captures the unifying theme for this collection of 38 position and research papers dealing with a variety of topics in distributed computing, ranging from theoretical foundations to emerging distributed applications, and from advanced distributed data services to innovative network communication paradigms. Thirty-one of the 38 papers in this volume are based on the preliminary reports that appeared in the proceedings of the *2002 International Workshop on Future Directions in Distributed Computing*, held on 3–7 June 2002 at the University of Bologna Residential Center located in the medieval hilltop town of Bertinoro, Italy. The workshop was organized by Özalp Babaoğlu, Ken Birman, and Keith Marzullo, and this book was envisioned by them. Without their efforts in organizing the Bertinoro workshop this book would not exist.

The plan to compile this volume was developed during the workshop. The numerous technical discussions at the workshop inspired many of the workshop participants to

substantially revise their position papers, and several decided to produce completely new papers. This volume is the culmination of this development and, in addition to the refined workshop papers, it includes seven completely new contributions. All papers were anonymously cross-refereed; the editorial team accepted papers following their revision based on the referee comments.

The papers are grouped into four parts, preceded by two papers with insights into technological and social processes that are a part of the development of distributed computing technology and its applications. The first paper presents a historical perspective on the development of group communication technology. Group communication services is an example of reusable and sophisticated building blocks that have been successfully used in developing dependable distributed applications and that improve our ability to construct such applications. The second paper discusses the social interactions along the boundary between academic distributed systems research and the real-world practice of distributed computing, and tackles the question of the academic research impact on the real world.

The other papers are grouped in the following four parts. Part I includes 12 papers dealing with the theoretical foundations of distributed computing, selected current research projects, and future directions. Part II consists of eight papers presenting research on novel communication and network services for distributed systems. Part III includes eight papers dealing with data and file services, and coherence and replication in network computing. Part IV includes eight papers dealing with system solutions, pervasive computing, and applications of distributed computing technologies.

We are certain that the papers in this volume contain numerous research topics that ought to be addressed in future research or that are already being addressed by forward-looking research in distributed computing. We are also certain that it is impossible to include all worthwhile future research topics in a single volume, and that some topics identified in this volume will turn out to be only moderately interesting. At the same time, we consider it necessary to identify the research topics that a broad cross-section of the research community views as important for the future of distributed computing.

January 2003

André Schiper
Alex A. Shvartsman
Hakim Weatherspoon
Ben Y. Zhao

Table of Contents

1. Practical Impact of Group Communication Theory

André Schiper

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
andre.schiper@epfl.ch

1.1 Introduction

Group communication is nowadays a well established topic in distributed computing. It emerged over the years as a topic with a strong synergy between theory and practice: group communication is highly relevant for building distributed systems, and is also of theoretical importance, because of the difficult problems it addresses. The paper presents an – inevitably subjective – retrospective of the main milestones that led to our current understanding of group communication, with the focus on theoretical contributions of practical relevance. Some open issues are discussed at the end of the paper.

What is theory? In the context of group communication, theoretical contributions can be defined as contributions to *abstractions* and *paradigms*, contributions to *system models* and *problem specifications*, and of course contributions to *algorithms*.

What is of practical importance in the context of group communication? We can mention *efficiency*, *clean structure (or architecture) of the system*, *flexibility of the system*, *correctness*, and *clear understanding of the properties of the system*. Efficiency is clearly important: no one would like to have an inefficient group communication system. *Algorithms* and *paradigms* may contribute to efficiency: a new algorithm, or the application of a new paradigm, may increase efficiency. *Abstractions* and *algorithms* contribute to a clean and flexible architecture of the system. Consider for example atomic broadcast. The algorithm in [1.11] solves atomic broadcast by reduction to consensus. This identifies consensus as a key abstraction, and directly influences the architecture of a group communication system by introducing a *consensus* component. A system with an architecture that reflects key *algorithmic abstractions* has more chance to be flexible, i.e., adaptable to a changing environment. Correctness, which obviously is of practical importance, is related to *specifications* (what is the system supposed to do), to *system models* (what are the assumptions about the system) and to *algorithms* (which are proven to meet a given specification in a given system model). Clear understanding of the properties of the system can be seen as a subset of correctness. When a system is deployed, it is important to know the conditions under which the system can deliver the services it is supposed to provide. This requires understanding of the *specifications* of the system, of the *system models* and *algorithms*, and also of *theoretical results* (e.g., the FLP impossibility result [1.26], or the result about the weakest failure detector for solving consensus [1.10]).

To summarize, the retrospective addresses the practical impact of group communication theory by focusing on key contributions to *abstractions*, *algorithms*, *paradigms*,

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 1–10, 2003.

specifications, *system models* and *theoretical results*. Contributions are grouped into three periods[1]:

- *Prehistory* (1972 - 1985)
- *Early years* (1985 - 1991)
- *Maturity and confusion* (1992 - 2002)

We start our retrospective in 1972, the year of the publication of the first paper that attempts to propose a completely software-based approach to fault-tolerance [1.44]. We end the "prehistorical" period in 1985 with the publication of the FLP impossibility result in the Journal of the ACM [1.26]. The year 1985 marks also the beginning of the "early years" period, with the publication of the first Isis paper [1.9]. We end this period in 1991, with the publication of the failure detectors paper [1.11], an important step in the history of group communication. The "maturity and confusion" period starts with the publication of the first partitionable group membership paper [1.3].

1.2 Prehistory (1972–1985)

We give only a brief overview of the main contributions the first period 1972-1985. In the context of *abstractions* and *specifications*, it is interesting to note that fundamental abstractions are identified: *interactive consistency* [1.36], *consensus* [1.25], *atomic broadcast* [1.13, 1.19]. The focus at that time was on Byzantine failures [1.45, 1.33], which shows that the complexity of solving problems with crash failures only was underestimated. The early focus on Byzantine failures had a major impact on problem specification, i.e., it led to consider *non-uniform* specifications, and the interactive consistency problem.

The *state machine paradigm* is also identified in the prehistorical period, and its implementation discussed in various system models: with no crashes [1.30], in the context of crash failures [1.42] and in the context of Byzantine failures [1.31]. The *rotating coordinator paradigm* is also already mentioned [1.40], even though it became well known only later (see Section 1.3.2).

Strong system *models* are considered in that period, e.g., the *synchronous round* model [1.36] and the *fail-stop* model [1.43], which corresponds to the asynchronous model with perfect failure detection. Models with randomization, for solving consensus, are also proposed [1.6, 1.39]. Finally, the FLP impossibility result [1.26], stating that there is no deterministic algorithm for solving consensus in an asynchronous system if one single process may crash, ends this initial period. The result will move the focus from Byzantine failures to crash failures.

1.3 Early Years (1985–1991)

1.3.1 Abstractions and Specifications

Process groups, *virtual synchrony* and *group membership* appear in the period 1985-1991. The notion of *process group* was proposed initially in an operation systems paper,

[1] The chronology of the published papers (conference or journal version) sometimes differs from the chronology of the corresponding technical reports. If the interested authors send me the historical TR references, I will add them in an extended version of this paper.

in the context of the V System [1.17]. In the paper, Cheriton and Zwaenepoel mention operations such as *join group*, *leave group*, *send message to group*, etc. It is interesting to notice that the paper also mentions the *publish-subscribe* paradigm. The concept of *virtual synchrony* was introduced in the paper by Birman and Joseph, as a specification that encompasses atomic broadcast (abcast), causal broadcast (cbcast) and group broadcast (gbcast) [1.8]. The paper does not give any precise specification of these group communication primitives, but stresses on the benefit of using these abstractions to develop fault-tolerant software:

> *We argue that this approach to building distributed and fault-tolerant software is more straightforward, more flexible and more likely to yield correct solutions than alternative approaches.*

The *group membership* abstraction and its specification appears in two papers in 1991: [1.41, 1.20]. The two papers give specifications for what is called today the *primary partition* group membership. [1.41] focusses on *processor* membership (i.e., the members of the group are processors) in an asynchronous system model, and advocates group membership as a mean to provide consistent failure notification. The specification was later shown to be flawed (see Section 1.4.2). [1.20] discusses *processor* and *process* membership in a synchronous system model. The process membership information is obtained from the processor membership. The goal of group membership is here to solve the continuous leader election problem.

1.3.2 Paradigms

We have already mentioned the *rotating coordinator* paradigm in the period 1972-1985. However, the paradigm became really known in the "early years" period, thanks to two papers [1.23, 1.11], which use the paradigm to solve consensus. In the rotating coordinator paradigm, the coordinator role moves from one process p to another process q, in a predetermined order, whenever p is suspected. Once all processes have been suspected, the coordinator role returns to the first coordinator, etc. A process can thus become the coordinator more than once. The paper by Chang and Maxemchuck [1.14], which uses a rotating token to implement atomic broadcast, is sometimes mentioned in the context of the rotating coordinator paradigm. However, token passing in [1.14] is not related to failure suspicions, and thus does not correspond to the definition.

1.3.3 System Models

The synchronous round model and the fail-stop model mentioned in Section 1.2 are very constraining from a practical point of view: they do not allow wrong failure suspicions. From a practical point of view one would like to have a model that allows the system to mistakenly suspect correct processes. Two such system models were proposed in that period: the *partially synchronous* model [1.23] and the *failure detector* model [1.11].

The partially synchronous model considers bounds on the message transmission delay and on the relative speed of processes. There are two variants of the model: (1) the bounds exist, but are not known, and (2) the bounds are known, but hold only from some unknown point on. The second variant is usually considered, but the first variant is actually more appealing from a practical point of view.

The failure detector model can be seen as a refinement of the partially synchronous model. A failure detector is defined by a *completeness* property and by an *accuracy* property. The completeness property defines the behaviour of the failure detectors with respect to faulty processes, i.e., processes that crash. The accuracy property defines the behavior of the failure detectors with respect to correct processes.

1.3.4 Algorithms

Many group communication algorithms were published in the period 1985-1991, and it is impossible to mention them all. In chronological order, the two first papers to mention are related to the group membership abstraction [1.21, 1.1]. The first paper defines *dynamic voting* (or *dynamic quorums*) [1.21], the second *view-based quorums* [1.1]. The two papers introduce some flavor of "primary partition group membership". In [1.21], each site maintains some information, which allows the site to determine who is in the same partition. In [1.1], the authors write: *"Each site s maintains a set called its view, the set of sites s assumes it can communicate with. Associated with each view is a view-id and two sites are said to be in the same view if their views have identical views-ids"*. Read and write quorums are based on views. The two papers show the algorithmic benefit of a dynamic membership information. This became less clear later . . . (see Section 1.4.2).

While the benefit of group communication abstractions is discussed by Birman and Joseph in [1.8], the same authors present the algorithms that implement these abstractions in [1.7]. The paper presents algorithms for causal broadcast (based on piggybacking on every message m the messages that are in the causal past of m) and for atomic broadcast, based on an idea by Skeen (for each message m a sequence number $sn(m)$ is computed, and messages are delivered in the order of their sequence number). The paper discusses also global broadcast (gbcast), with a rather complicated algorithm. Even though the paper lacks of rigorous specifications for group communication primitives, and does not address liveness, it constitutes a major milestone in the history of group communication.

Consensus algorithms are the key algorithmic contributions in the period 1985-1991: (1) the consensus algorithm for the partially synchronous model based on the rotating coordinator paradigm [1.23], (2) the Paxos algorithm based on leader election [1.32], and (3) consensus algorithms for the asynchronous system augmented with failure detectors [1.11], specifically the consensus algorithm using the $\diamond\mathcal{S}$ failure detector and based on the rotating coordinator paradigm. These three algorithms have in common the separation of correctness into safety and liveness properties, where the safety properties hold no matter how asynchronously the system behaves. Termination (i.e., liveness) requires some additional condition.

It is interesting to read Lamport's comments about Paxos [1.29]: the algorithm was submitted for publication in 1990, but not published before 1998. It was initially not understood, and the reviewers of found the paper *"mildly interesting, though not very important"*. . . Currently it is still an open question whether it is more efficient to solve consensus using a rotating coordinator or leader election.

Another key contribution is the atomic broadcast algorithm by Chandra and Toueg [1.11] based on reduction to consensus: the atomic broadcast algorithm inherits the properties of the consensus algorithm, e.g., safety holds no matter how asynchronous the

system is. The algorithm also shows the practical importance of consensus. References of other atomic broadcast algorithms can be found in [1.22].

1.3.5 Summary

To summarize, in the period 1985-1991, the notion of group membership appears (implicitly and explicitly), the partially synchronous and the failure detector system models are defined, and the rotating coordinator paradigm becomes popular. In terms of algorithms, the main contributions are consensus algorithms whose safety properties hold even if the system behaves completely asynchronously, and the atomic broadcast algorithm solved by reduction to consensus.

1.4 Maturity and Confusion (1992–2002)

The contributions from 1985 to 1991 establish the theoretical basis for group communication and lead to substantial progress in the field during the next ten years. New topics continue to emerge during this period, and some of these topics serve as a catalyst for further research on theoretical foundations.

1.4.1 Maturity

The elegance and the power of failure detector model have influenced a large number of researchers, and generated an important literature. Other important results were also established.

Abstractions, Specifications and System Models. In the context of abstractions and specifications, the paper by Hadzilacos and Toueg [1.27] has played an important role: it has contributed in part by motivating subsequent authors to pay more attention to a careful specification of group communication. A new group communication primitive, called *generic broadcast* [1.38], which takes the message semantics into account, was also proposed. The primitive is parametrized by a *conflict* relation, and ensures that two conflicting messages are delivered in the same order by all processes, while non-conflicting messages need not to be ordered.

In the context of system models, the failure detector model, initially defined in a model where processes do not recover after a crash, has been extended to a model with process recovery [1.2]. The paper introduces the notion of *good* process, a process that is always up or eventually permanently up, and of *bad* process, a process that is eventually permanently down or unstable (i.e., permanently crashing and recovering). The paper also introduces the failure detector $\diamond\mathcal{S}_u$, based on epoch numbers, for solving consensus in the crash-recovery model.

The increasing maturity of the field is also witnessed by the comprehensive survey on group communication specifications by Chockler, Keidar and Vitenberg [1.18]. The "consensus" survey by Barborak, Malek and Dahbura [1.5] may also be mentioned here, even though the paper is written from a *system diagnosis* perspective, rather than from a *distributed computing* point of view.

Theoretical Results. Two important theoretical results were established in this period. One, by Chandra, Hadzilacos and Toueg, shows that $\diamond\mathcal{S}$ (or equivalently $\diamond\mathcal{W}$) is the weakest failure detector that allows us to solve consensus in an asynchronous system.

This has a very practical importance: a system that uses a $\diamond\mathcal{S}$-based consensus algorithm, solves consensus in the maximum possible runs. Another result is the impossibility of solving the group membership problem in an asynchronous system [1.12]. Before this paper it was sometimes claimed that the group membership problem was not subject to the FLP impossibility result, because the model allowed processes to crash (and to kill other processes).

The following works are also worth mentioning. One is the paper by Malkhi and Reiter about *Byzantine* quorums, which significantly extended previous results and renewed the research on quorum systems, e.g., [1.34]. The other work is by Fekete, Lynch and Shvartsman [1.24] in which the authors formally specify and *prove* a view-oriented group communication service (the proofs were subsequently checked by a theorem prover).

Algorithms. Many algorithms were published in this period, e.g., consensus algorithms based on failure detectors[2] and algorithms for solving other agreement problems by reduction to consensus (atomic commitment, atomic multicast, group membership). The most important algorithm to mention is probably the algorithm for solving consensus in the static system model with process recovery, based on the failure detector $\diamond\mathcal{S}_u$ [1.2].

1.4.2 Confusion

Despite the maturity of the domain, group communication is far from being a solved problem. Many contributions, including theoretical contributions, are still needed in order to clarify some controversial issues, mostly in the context of the group membership problem. We mention first the CATOCS controversy (causally and totally ordered communication support), which generated lots of discussions some years ago.

The CATOCS Controversy. Cheriton and Skeen expressed several criticisms to the use of group communication to build distributed applications [1.16]. These criticisms, which have led to the so-called CATOCS controversy, can be summarized as follows: (1) group communication cannot guarantee total ordering between operations that correspond to groups of messages (e.g., transactions), (2) there is no efficiency gain over state-level techniques, (3) semantic ordering cannot be expressed, and (4) group communication does not ensure end-to-end guarantees. According to these criticisms, group communication has not brought anything useful to distributed computing. This is incorrect.

First, group communication define new abstractions, e.g. atomic broadcast. Abstractions are important, and the whole development of computer science consists of identifying new abstractions that contribute to the understanding of the field. Moreover, abstractions allow the development of more complex applications, and decrease the risk of errors. Do abstractions lead to inefficient solutions? This is an old story[3]. While early implementation of group communication where known to be slow, this is no longer

[2] It is interesting to note that already in 1983 (!), Fischer writes *"We survey the considerable literature on the consensus problem that was developed over the past few years and give an informal overview of the major theoretical results in this area"*. I don't know what should be said today . . .

[3] In the early seventies, some people were claiming that it will never be possible to write programs more efficiently than using assembly languages. I don't know whether these people would be eager to program a modern RISC processor at the assembler-language level!

the case with current LAN technology. For example with 100Mbits/s Ethernet, atomic broadcast becomes extremely fast, e.g., around 500-1000 atomic broadcasts per second.

It is true that group communication as such does not guarantee total ordering between operations that correspond to groups of messages. However, this does not prevent providing higher level ordering guarantees using group communication. A good example are the recent results showing the benefit of using atomic broadcast to implement transactions over a replicated database [1.37, 1.28, 1.46]: the use of atomic broadcast greatly simplifies the solution, while leading to better performances over standard database replication techniques. It is also false to claim that semantic ordering constraints cannot be expressed using group communications. Generic broadcast is an example of message ordering communication primitive that uses message semantics [1.38].

The comment about the absence of end-to-end guarantees available from group communication is maybe the most interesting. First, it is true that the absence of end-to-end guarantees can lead to problems. For example, it has been shown that the absence of end-to-end guarantees does not allow one to implement *2-safe* transactions over a replicated database using atomic broadcast [1.46]. However, (1) end-to-end guarantees can be added to group communication primitives, and (2) the absence of end-to-end guarantees can sometimes be an advantage. For example, it can be exploited to define a new safety property for transactions called *group-safety*, weaker than 2-safety, which can be sometimes more efficient than *lazy replication* while providing much stronger guarantees [1.46].

The Group Membership Issue. The group membership issue is of a different nature than the CATOCS controversy. Here, more contributions are definitely needed to clarify important issues.

The year 1992 corresponds to the publication of the first *partitionable* membership paper [1.3]. In that paper group membership is defined as *"maintaining the Current Configuration Set in consensus among the set of machines that are connected throughout the activation of the membership protocol"*. If this first definition leaves many questions open, ten years later the fundamental questions related to partitionable membership have still not been adequately addressed. For example, in [1.18] Chockler *et al.* give the following specification in the context of the partitionable membership problem: *"If the failure detector behaves like $\diamond\mathcal{P}$, then for every stable component S there exists a view $V = S$ such that every process in S installs V as its last view"*. This looks like the detection of a global (stable) property. Is the partitionable group membership problem an agreement problem, or does it rather correspond to the detection of a stable property?

While the specification of the partitionable membership problem is one issue, the role of this abstraction is still unclear. Convincing examples are needed. While the Isis *processor* membership abstraction [1.41] has clear limitations (one single group, all processors member of this group, progress only in the majority partition of the network), this limitation can be left by relying on multiple *process* groups, with one group for every set of replicated servers. Partitionable membership is not needed for solving agreement problems when the network can partition.

These comments may suggest that the simpler primary partition membership problem is fully understood. Unfortunately, this is not the case. Many specifications have been published, but none of them is satisfactory: it has been shown that current specifications

(partitionable, but also primary partition) admit trivial solutions or allow for undesirable behavior [1.4].

The question of the algorithmic role of the primary partition membership problem requires also some clarification. One role was clearly identified about 15 years ago (see Section 1.3.4). The role that is mentioned nowadays is failure detection, e.g., the notification of the crash of a process. However, failure notification does not need to be consistent to be able to solve agreement problems, e.g., consensus or atomic broadcast, as known from the failure detector approach [1.11]. Using group membership for solving atomic broadcast is an overkill. Moreover, since atomic broadcast does not require the consistent failure notification provided by group membership, one can wonder whether group membership should not be solved on top of atomic broadcast, as in [1.35], rather than the opposite (as done currently in many implementations). Group membership also allows the system to discard messages from output buffers. However, having one mechanism for solving two problems (failure notification and discarding messages from output buffers) is not a good solution [1.15].

Finally, the last issue that needs to be better addressed is the specification of *dynamic* group communication, i.e., group communication in an environment where processes can be added and removed during the computation[4]. One typical example is the *view synchronous multicast* primitive in the context of primary partition membership. The primitive appears close to *reliable broadcast*. However, both specifications are far away. The same holds for (static) *vs.* (dynamic) atomic broadcast. This is not satisfactory.

1.5 Conclusion

Results of more than twenty years of research have contributed to a very good understanding of many issues related to group communication. The need to ensure safety *and* liveness in the context of agreement problems (e.g., consensus, atomic broadcast) and the practical importance of consensus are now recognized. Nevertheless, group communication is not yet a solved problem. More work is needed on some issues, e.g., to come with convincing specifications of the membership problem (primary partition and partitionable), and with better specifications for dynamic group communication. The algorithmic benefit of the membership abstraction needs also to be better understood. Strong contributions are needed, to build on more solid grounds. Only so will group communication be better accepted outside of our community.

References

1.1 A. El Abbadi and S. Toueg. Availability in Partitioned Replicated Databases. In *ACM SIGACT-SIGMOD symp. on prin. of Database Systems*, pages 240–251, March 1986. See also *ACM Trans. on Database Systems*, 14(2):264-290, June 1989.

1.2 M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Symposium on Distributed Computing*, LNCS, pages 231–245. Springer, September 1998.

[4] [1.27] addresses the specification of *static* group communication primitives.

1.3 Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

1.4 E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Department of Computer Science, Cornell University, August 1995.

1.5 M. Barborak, M. Malek, and A. Dahbura. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

1.6 M. Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In *proc. 2nd annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

1.7 K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.

1.8 K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138. ACM, Nov 87.

1.9 K. Birman, T. Joseph, T. Räuchle, and A. El Abbadi. Implementing Fault-Tolerant Distributed Objects. *IEEE Trans. on Software Engineering*, 11(6):502–508, June 1985.

1.10 T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of ACM*, 43(4):685–722, 1996.

1.11 T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *proc. 10th annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991. See also *Journal of the ACM*, 43(2):225-267, 1996.

1.12 Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, Pennsylvania, USA, May 1996.

1.13 J-M. Chang. Simplifying Distributed Database Systems Design by Using a Broadcast Network. In *Proc. of the ACM SIGMOD Int. Conference on Management of Data*, pages 223–233, June 1984.

1.14 J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.

1.15 B. Charron-Bost, X. Défago, and A. Schiper. Broadcasting Messages in Fault-Tolerant Distributed Systems: the benefit of handling input-triggered and output-triggered suspicions differently. In *21st IEEE Symp. on Reliable Distributed Systems (SRDS-21)*, pages 244–249, Osaka, Japan, October 2002.

1.16 D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communications. In *14th ACM Symp. Operating Systems Principles*, pages 44–57, Dec 1993.

1.17 D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. on Computer Systems*, 2(3):77–107, May 1985.

1.18 G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

1.19 F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *IEEE 15th Int Symp on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, June 1985.

1.20 Flaviu Cristian. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4(4):175–187, April 1991.

1.21 D. Davcec and A. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 87–96, 1985.

1.22 X. Défago, A. Schiper, and P. Urban. Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey. TR DSC/2000/036, EPFL, Communication Systems Departement, October 2000.

1.23 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–323, April 1988.

1.24 A. Fekete, N. Lynch, and A.A. Shvartsman. Specifying and Using a Group Communication Service. *ACM Trans. on Computer Systems*, 19(2):171–216, May 2001.
1.25 M. Fischer. The Consensus Problem in Unreliable Distributed Systems (A Brief Survey). In *Int. Conf. on Foundations of Computation Theory (FCT)*, pages 127–140, 1983.
1.26 M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
1.27 V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.
1.28 B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, ETH Zurich, Switzerland, 2000. Number 13864.
1.29 L. Lamport. Web home page. http://lamport.org/.
1.30 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 1978.
1.31 L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. on Progr. Languages and Syst.*, 6(2):254–280, April 1984.
1.32 L. Lamport. The Part-Time Parliament. Technical Report 49, Digital SRC, September 1989. See also *ACM Trans. on Computer Systems*, 16(2):133-169, May 1998.
1.33 L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Progr. Languages and Syst.*, 4(3):382–401, July 1982.
1.34 D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.
1.35 P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *IEEE 11th Intl. Conf. Distributed Computing Systems*, pages 480–488, May 91.
1.36 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of ACM*, 27(2):228–234, 1980.
1.37 F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, December 1999. Number 2090.
1.38 F. Pedone and A. Schiper. Generic Broadcast. In *13th. Intl. Symposium on Distributed Computing (DISC'99)*, pages 94–108. Springer Verlag, LNCS 1693, September 1999.
1.39 M. Rabin. Randomized Byzantine Generals. In *Proc. 24th Annual ACM Symposium on Foundations of Computer Science*, pages 403–409, 1983.
1.40 R. Reischuk. A New Solution for the Byzantine general's problem. Technical Report RJ 3673, IBM Research Laboratory, November 1982.
1.41 A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
1.42 F.B. Schneider. Synchronization in Distributed Programs. *ACM Trans. on Progr. Languages and Syst.*, 4(2):125–148, April 1982.
1.43 R.D. Schlichting and F.B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. on Computer Systems*, 1(3):222–238, August 1983.
1.44 J.H. Wensley. SIFT - Software Implementated Fault Tolerance. *FJCC*, pages 243–253, 1972.
1.45 J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shotak, and C.B. Weinstock. SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proc. IEEE*, 66(10):1240–1255, 1978.
1.46 M. Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2002. Number 2577.

2. On the Impact of Academic Distributed Systems Research on Industrial Practice

Michael D. Schroeder

Assistant Director
Microsoft Research Silicon Valley

Introduction

This article is based on the remarks I was invited to make at the 2002 Workshop on Future Directions in Distributed Computing on the impact of academic distributed systems research on the real world. In this short piece I document the gist of the presentation.

2.1 Good Ideas Eventually Have Impact

My overall assessment is positive. Good ideas do have an impact, independent of where those ideas were developed and tested, although the path from academic research to industrial impact is often indirect and can take a long time. One thing that academic research is good at and that industry finds especially useful is a complete model of a topic or area. Such a model defines a design space and makes clear the full range of possible mechanisms along with the functional properties, feature interactions, tradeoffs, and limits. Research provided such a model for parsing, for example, in the 1960s. A more recent example is the fairly complete understanding we now have of consensus. Work to develop a comprehensive understanding of security technology is nearing this level of completeness. For other sorts of research results, convincing demonstrations of the ideas working in a system along with careful measurements and assessments of the effectiveness can speed the impact.

2.2 Technology Transfer to Product Groups Is Hard

There is only a short window of opportunity during a product cycle when a product group is open to outside ideas. It is difficult for someone outside a company to connect with such a window. The best way to do technology transfer from an academic venue is to get involved in a startup (but see below the negative effect of this) or get involved in a custom system effort. As an example of the latter, the Isis technology from Cornell is used in European air traffic control systems.

Another issue for product groups is matching technology to business opportunities. Most good computer science research tends to produce solutions looking for problems. This tendency is a result of the higher probability that such bottom up work will generate creative results and advances in the state of the art. Top down work to develop a methodology, a tool, or a system that solves an explicitly articulated problem or market need

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 11–13, 2003.

tends to generate good engineering, not creative research or good theses. Unfortunately (for academic researchers) product groups think of their problems top down. They often will have limited patience for bottom up solutions because they will find it hard to figure out if the technology solves the problem at hand.

2.3 Impedance Match Academic Research with Product Groups Is Low

Many groups like the industrial systems research lab where I work are built out of the same sort of people as academic systems research groups. We also have the same top level goals: to contribute to the state of the art in computing. This is particularly so in the area of distributed systems. The result is that industrial research groups often have a good impedance match with academic research groups.

In the case of Microsoft Research, we work hard to track and connect with the windows of opportunity in product groups. When these opportunities are engaged we bring to the table both internally developed technology and an awareness of the work going on in outside groups. Thus we can provide a conduit between academic research and the company's product groups.

2.4 Product Groups Want Help in Unpopular and Hard Problem Areas

Right now there are three areas in distributed computing where I think academic research could help product efforts:

First, improved technology for the software engineering process and for testing. This area is hard because researchers do not have an accurate appreciation of the size of the software artifacts involved, of the size of the test sets, nor of the impact of backward compatibility requirements.

Second, good programming models for web-based services. Web-based services will be where distributed systems programming goes mainstream. Until now the software engineering profession has been unsuccessful in getting most applications programmers to properly use the RPC-based programming model for distributed systems. It is widely agreed that this model will be inadequate for web-based services. What model will replace it? How will this new model be embodied and taught? How will it encompass failure, debugging, and performance considerations?

Finally, better solutions to the system management problem. System management is a large component in the cost of ownership for most customers. System management is also a primary culprit in unreliability and insecurity of systems, especially distributed systems. Most academic researchers think this is a boring problem.

2.5 Academic Barriers to Interaction with Industry

In addition to the generally difficulty of doing technology transfer, universities have in the recent past erected two new barriers to collaboration.

The first was the wave of startups formed by research faculty. These startups remove people and ideas from the pool that could interact with industry.

The second was the attempt by many universities to make money from the intellectual property (IP) generated by faculty and students. The attempts to exploit IP raised the hassle factor for all sorts of interactions between industry and academia. Many collaborations have foundered simply because it was too much trouble to sort out the IP issues. This barrier has also had the effect of reducing the amount of financial support for research that industry channels to academia. I suspect a bottom line analysis would show that the universities lose more in research support than they gain from selling IP.

Recently the wave of startups has diminished and several universities have started to relax their IP views.

2.6 Fresh Ph.D.s often Lack Some of the Skills that Industry Values

The most important technology transfer that the university research community does to industry is via the students that take industrial jobs. While I am not a consumer of software developers and architects, I am a consumer of Ph.D. researchers. Generally speaking I think universities could do a better job of training these young researchers than they do. Of course the very best students will train themselves. It's pretty hard to do anything that will help or hinder them. Perhaps five percent of the output from the top university computer science departments fall into this category. This is the category we mine most diligently. The place where better training might matter is in the five percent to twenty five percent band. Can better training make more of these students into top-notch researchers? I think so.

When screening candidates for research positions we look for students who have identified an interesting problem, produced a creative solution, and concisely characterized the insights gained. We evaluate a thesis both as an example of the quality of the research skills a candidate has developed and as an example of the quality and impact of the technical advance of which the candidate is capable.

A good candidate will have intellectual spark, drive to succeed, independence, and a good awareness of technical context. The good candidate will have a broad range of technical interests and have flexibility to work on a variety of problems. They will also realistically assess the impact of their work. Inflated claims of the impact do not help a candidate's case.

Ocassionally we see symptoms of what we call "advisor failure." This most often manifests itself in a student lacking technical perspective, a skill we value highly. Such a student usually has done a good job of drilling down into a particular technical issue but does not understand what else has been done before, how the "new" research relates to problems and solutions already published, and whether the line of work is worth pursuing further. With respect to the last point, I sometimes believe that a candidate feels a need to claim interest in doing future work on the thesis topic just to validate that the topic is a deep and interesting one, even if it isn't. Often I'd rather that a student not want to follow up, but instead be ready for new problems and have thought about some such problems that would be worth pursuing.

Part I
Foundations of Distributed Systems: What Do We Still Expect from Theory?

Over the last 20 years, the impact of theory on the development of distributed systems has been invaluable. For example, it is hard to imagine today a world without the Fischer-Lynch-Paterson (FLP) impossibility result[1]. Theoretical results transform pragmatic know-how into a science. While engineers have developed expertise for building systems, theory is helping them to understand the properties of the systems they are constructing.

FLP is a negative result, but theory has also generated numerous positive results. Important contributions have been made in the context of the specification of problems (e.g., atomic broadcast) and in the definition of system models (e.g., the partially synchronous model, the failure detector model). Theory also yielded algorithms for important problems, and has contributed to our understanding of the difficulty of problems by establishing lower and upper bounds. Numerous theoretical results reached beyond the engineering know-how gained over the years and shaped the distributed systems area into a science.

What do we yet expect from theory? Have all the important results been established? Asking such questions is provocative, but essential. Indeed, it does happen that a research topic is pursued in the area where all important or worthwhile problems have been solved. In this respect, the papers in Part I offer very interesting insight. They show that more work on the foundations of distributed systems is definitively needed, and suggest relevant topics. Fault-tolerance is the common denominator in these papers. This is not really surprising, since distributed systems are of limited use without fault-tolerance, and fault-tolerance in distributed systems is a difficult problem with many facets.

Part I starts with seven papers related to consensus and other agreement problems. The first paper by Friedman *et al.* (page 17) relates consensus to error correcting codes, a domain that has been extensively studied. This approach might allow one to reuse results from the coding theory in distributed computing. The second paper by Lamport (page 22) classifies the process roles in the consensus problem into *proposers*, *acceptors* and *learners*, and addresses lower bounds results for asynchronous consensus. These roles, which might be considered in other agreement problems as well, may lead to interesting new insights. The third paper by Junqueira and Marzullo (page 24) proposes to replace the traditional "t faulty processes out of n" failure assumption by an abstraction that takes into account dependencies between failures. The forth paper by Charron-Bost (page 29) compares two of the most important agreement problems in distributed computing, *consensus* and *atomic commitment*. It shows that characterizing precisely the failure detector needed for solving an agreement problem leads to a better understanding of the fundamental nature of the different agreement problems. The next two papers are about

[1] The result, established in 1983, says that the consensus problem (one of the most basic problems in distributed fault-tolerant computing) is not solvable in an asynchronous system by a deterministic algorithm, if one single process may crash.

a topic that was largely neglected until now: the evaluation of algorithms, beyond the classical message and time complexity analysis. Many algorithms for solving agreement problems have been published, but too little is known about their performance. The paper by Keidar and Rajsbaum advocates the performance analysis of algorithms in the so called *well-behaved runs* (page 35). The paper by Keidar (page 40) discusses the problem of evaluating distributed algorithms, including in runs with failures, using relevant metrics. While it is important to understand and to characterize the performance of distributed algorithms, it is equally important to be able to improve their performance. The paper by Jiménez and Patiño-Martínez (page 45) advocates optimistic approaches with safeguards as the means for improving performance.

Failures in distributed systems can be grouped into two categories: benign failures (e.g., crash failures) and malicious failures (also called Byzantine failures). Dealing with malicious failures is more difficult than dealing with benign failures, which may explain why malicious failures have been less studied in recent years. However modern systems will need to be even more resilient to malicious attacks. Malicious failures are part of the model in the paper by Lamport (page 22) in the context of the consensus problem. Reaching agreement in the presence of malicious processes is not the only issue. Equally important is to ensure confidentiality of the information accessed by replicated servers. This issue is addressed by the paper by Yin *et al.* (page 51), which proposes the design of robust firewalls. Malicious failures are also considered in the paper by Cachin (page 57), but from the point of view of security and its enabling technology, i.e., cryptography. The observation is that whereas the formal model of modern cryptography is based on computational complexity theory, the formal model of distributed computing does not usually deal with bounds on time complexity. This makes it impossible to analyze Byzantine agreement protocols that use cryptography. The paper suggests a new formal model to bridge this gap. Presenting the considerable work done in distributed systems in such a framework is a challenging task.

Many distributed applications have been deployed in recent years, yet we are still at the beginning of the deployment of large-scale distributed applications in terms of the number of nodes and in terms of the geographic distances. Future systems will include mobile components, but they will also be more complex and more dynamic. Fundamental work is needed to accompany these developments. The paper by Lynch and Shvartsman (page 62) discusses research issues related to communication and data sharing in such highly dynamic distributed environments, with the goal of developing a coherent theory. The paper by Guerraoui (page 68) discusses the need for new abstractions for building such systems. Finally, consistency in large systems is a key issue. The paper by Ahamad and Raynal (page 73) suggests a new characterization of consistency criteria based on timeliness and ordering. It provides a new look on known consistency criteria, and opens the way for new exploration.

To summarize, the papers below represent a cross-section of open questions related to current and emerging topics in distributed computing theory. They witness that new theoretical results are still needed to consolidate the foundations of distributed computing.

André Schiper

3. Using Error-Correcting Codes to Solve Distributed Agreement Problems: A Future Direction in Distributed Computing?

Roy Friedman[1], Achour Mostéfaoui[2], Sergio Rajsbaum[3], and Michel Raynal[2]

[1] Department of Computer Science, The Technion, Haifa, Israel
roy@cs.technion.ac.il
[2] IRISA - Campus de Beaulieu, 35042 Rennes Cedex, France
achour,raynal@irisa.fr
[3] HP Research Lab, Cambridge, MA 02139, USA and Inst. Matem. UNAM, Mexico
Sergio.Rajsbaum@hp.com

3.1 Introduction

The *Consensus* problem lies at the heart of many distributed computing problems one has to solve when designing reliable applications on top of unreliable distributed asynchronous systems. There is a large literature where theoretical and practical aspects of this problem are studied[1], that can be informally stated in terms of three requirements. Each process proposes a value, and has to decide on a value (termination) such that there is a single decided value (agreement), and the decided value is a proposed value (validity). One of the most fundamental impossibility results in distributed computing says that this apparently simple problem has no deterministic solution in an asynchronous system even if only one process may crash [3.9]. To circumvent this impossibility, known as FLP, two main approaches have been investigated. One of them consists of relaxing the requirements of the problem, by either allowing for probabilistic solutions (e.g., [3.4]), or for approximate solutions (ε-*agreement* [3.8], or k-*set agreement* [3.6]). Another approach consists of enriching the system with synchrony assumptions until they allow the problem to be solved [3.7]. This approach has been abstracted in the notion of unreliable failure detectors [3.5]. There have also been studies of hybrid approaches, like combining failure detection with randomization [3.2, 3.21].

It is instrumental to view the set of initial input values to an agreement problem, one for each participating process, as an input vector to the problem. As elaborated below, we have recently studied the feasibility of solving agreement problems by restricting the allowed set of input vectors to various error correcting codes. Our work so far focused on Consensus, *interactive consistency* [3.22], and k-set agreement [3.6]. In this paper we describe our approach, the results we obtained so far, and some open problems in this direction. Our work is an extension of the *condition-based approach*. Thus, we start with an overview of that approach in Section 3.2. We then describe our error-correcting based results in Section3.3, and conclude with a list of open problems in Section 3.4.

[1] The distributed computing textbooks [3.3, 3.14] devote several chapters to this problem.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 17–21, 2003.

3.2 The Condition Based Approach

We have recently introduced a new *condition-based approach* to tackle the consensus problem [3.15]. This approach focuses on sets of input vectors that allow n processes to solve the consensus problem despite up to f process crashes, in a standard asynchronous model. Let an *input vector* be a size n vector, whose i-th entry contains the value proposed by a process p_i. A *condition* (which involves the parameters f and n) is a set C of such vectors that can be proposed under normal operating conditions. The aim is then to design f-fault tolerant protocols that (1) are always safe (i.e., the protocol has to always guarantee agreement and validity, whether the proposed input vector is allowed by the condition or not), and (2) are live at least when certain assumptions are satisfied (e.g., when no process is faulty, or when the input vector belongs to the condition). This is the best we can hope for, since the FLP impossibility result says we cannot require that a consensus protocol terminates always, for every input vector. But, by guaranteeing that safety is never violated, the hope is that such a protocol should be useful in applications. For example, consider the condition "more than a majority of the processes propose the same value." It is not hard to see that consensus can be solved when the inputs satisfy this condition, when $f = 1$. It is plausible to imagine an application that in some real system satisfies this condition most of the time; only when something goes wrong, the processes proposals get evenly divided, and only then should the protocol take longer to terminate (or even not terminate).

A characterization of the conditions that admit a consensus protocol with the above properties is presented in [3.15]. That is, we identify a specific set $\mathcal{C}$ of conditions and prove that there is a consensus protocol for a condition C if and only if $C \in \mathcal{C}$. It is shown in [3.18] that the classes of conditions that permit to solve consensus define a hierarchy. An efficient condition-based consensus protocol is described in [3.19].

3.3 Distributed Agreement and Error Correcting Codes

3.3.1 Code-Based Characterization of Agreement Problems

Codes for Consensus. In general, for each protocol solving consensus there has to be at least one input vector that can lead to different decision values in different runs of the protocol. However, it turns out that this is exactly why the problem is unsolvable. Hence, if we restrict the allowed input vectors such that each vector can only lead to a single predefined decision value, then consensus becomes solvable (under the condition-based approach). Each such allowed vector can be viewed as a code word made up of n digits (one per process) that encodes a single decision value. This observation had led to another way to characterize the set of conditions that permit to solve the consensus problem. This idea is developed in [3.10, 3.11] where it is shown that a condition permits to solve the consensus problem iff its input vectors correspond to a code whose Hamming distance between any two code words leading to different decoded values is at least $f+1$. Similarly, Byzantine consensus, in which processes may fail arbitrarily, can be solved iff the Hamming distance is at least $2f + 1$. This characterization, which shows that a well defined subset of error correcting codes corresponds to the consensus problem, establishes a first connection relating distributed agreement and error correcting codes.

Interactive Consistency. In the interactive consistency problem [3.22], each process proposes a value and the correct processes have to agree on the vector of proposed values. Interactive consistency with the condition-based approach (denoted CB_IC) has been investigated in [3.11, 3.16] which provides a characterization of the set of conditions that allow to solve CB_IC in the presence of f_c crashes and f_e erroneous proposals. Those are the conditions including input vectors such that their Hamming distance is at least $2f_e + f_c + 1$. Therefore, these conditions correspond exactly to the error-correcting codes, where the errors can be erasures or modified values. So, this result establishes a second connection relating distributed agreement and error correcting codes.

3.3.2 Interest of the Approach

Results from coding theory can be useful in distributed computing. For instance, any of the known (f_c, f_e) codes yields a condition C for which CB_IC can be solved. And if the code is known to be maximal, adding any input vector to the corresponding condition makes the CB_IC problem unsolvable. Similarly, one can derive coding theory results from distributed computing. As a simple example we have shown how to derive the fact that there are no perfect codes that tolerate erasures [3.16]. Additionally, it is clear from this work that no error correcting information can be computed by a fully asynchronous distributed system prone to failures.

Interestingly, the condition-based treatment permits to formalize a sense in which CB_IC is harder than consensus. Each condition C has an associated graph, whose vertices are all possible inputs to the condition: the input vectors of C together with their subvectors where at most f_c entries are replaced by $\perp$ (to represent initially crashed processes). Two vectors are joined with an edge if they differ in at most f_c entries. A connected component of a legal condition for CB_IC can contain only one (full) vector, while this is not the case for consensus. In both cases connected components are at least distance $f_c + 1$ apart.

Veríssimo introduced the notion of *wormholes* in [3.23]. Wormholes are essentially synchronous connections between a few nodes in an otherwise asynchronous distributed system. The model calls for most messages to be sent asynchronously, and using the wormholes only when needed, since they are expensive and can only support limited bandwidth. The condition-based approach can be used to derive efficient schemes for using wormholes. That is, in error correcting codes, typically the value of some digits, known also as the *protection digits*, are defined as a function of other digits. There seems to be an inherent tradeoff between the number of protection digits and the number of digits a protection digit must depend on. Moreover, when relating back to distributed computing, there might be additional considerations, such as the desire to avoid long haul wormholes. (The two extremes are parity bit in the case of erasures and Hamming code in the case of corruption on one hand, and simply duplicating or triplicating each bit on the other hand [3.10].) Thus, by choosing the right error correcting code, it is possible to solve agreement related problems using the optimal number of wormholes for the given environmental constraints.

3.4 Open Problems and Future Directions

The previous connections relating agreement problems and error correcting codes can open new research opportunities to address the way agreement problems are tackled and solved in distributed systems [3.11]. We list here a few of them.

- The condition-based approach can be combined with more traditional approaches (e.g., failure detectors) to provide more efficient agreement protocols. First steps in this direction are described in [3.10, 3.17].
- Study the condition-based approach in synchronous and partially synchronous systems. For instance, consider the following questions. It is well known that $f+1$ rounds are necessary and sufficient to solve consensus in a synchronous system. What is the set of conditions that allow to solve consensus in c rounds, $c \leq f+1$? This question can be asked when f is the actual number of failures in an execution or when it is a bound on the total number of possible failures. Is there an interesting relation to coding theory in this setting?
- The design of a general framework allowing to design a condition-based protocol that could be customized to solve either consensus or interactive consistency still remains to be defined.
- Can the condition-based approach be used to solve other agreement problems such as renaming, primary partition group membership, or atomic commit? Finding the conditions for k-set agreement is still an open problem (some progress appears in [3.1, 3.20]).
- A grander challenge is finding a general characterization of the relation between agreement problems and error correcting codes. For example, in recent years several works were published on the topological characterizations of distributed agreement problems that can be solved in asynchronous environments prone to failures (e.g., [3.12, 3.13]). It remains to be seen whether it is possible to find a direct mathematical relation between the corresponding topological structures and error correcting codes, or information theory in general.

In general, it seems that finding a linkage between coding theory and agreement problems in distributed computing is an important issue. Coding is an area that was studied extensively. By applying results from coding theory, it could be possible to find simpler proofs of existing results or even to obtain new results in distributed computing.

References

3.1 Attiya H. and Avidor Z., Wait-Free n-Consensus When Inputs are Restricted. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 326-338, Toulouse (France), 2002.

3.2 Aguilera M.K. and Toueg S., Failure Detection and Randomization: a Hybrid Approach to Solve Consensus. *SIAM Journal of Computing*, 28(3):890-903, 1998.

3.3 Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw–Hill, 451 pages, 1998.

3.4 Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, ACM Press, pp. 27-30, Montréal (Canada), 1983.
3.5 Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2):225-267, 1996.
3.6 Chaudhuri S., More *Choices* Allow More *Faults:* Set consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
3.7 Dwork C., Lynch N.A. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *JACM*, 35(2):288–323, 1988.
3.8 Dolev D., Lynch N.A., Pinter S., Stark E.W., and Weihl W.E., Reaching Approximate Agreement in the Presence of Faults. *JACM*, 33(3):499-516, 1986.
3.9 Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374-382, 1985.
3.10 Friedman R., A Simple Coding Theory-Based Characterization of Conditions for Solving Consensus. *Technical Report CS-2002-06 (Revised Version)*, Department of Computer Science, The Technion, Haifa, Israel, June 30, 2002.
3.11 Friedman R., Mostefaoui A., Rajsbaum S. and Raynal M., Distributed Agreement and its Relation with Error-Correcting Codes. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 63-87, Toulouse (France), 2002.
3.12 Herlihy M. and Rajsbaum S., Algebraic Topology and Distributed Computing: a Primer. *Computer Science Today: Recent Trends and Developments*, Springer Verlag LNCS #1000 (J. van Leuween Ed.), pp. 203-217, 1995.
3.13 Herlihy M. and Shavit N., The Topological Structure of Asynchronous Computability. *JACM*, 46(6):858-923, 1999.
3.14 Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, 872 pages, 1996.
3.15 Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM Symposium on Theory of Computing (STOC'01)*, ACM Press, pp. 153-162, 2001.
3.16 Mostefaoui A., Rajsbaum S. and Raynal M., Asynchronous Interactive Consistency and its Relation with Error Correcting Codes. *Research Report #1455*, IRISA, University of Rennes, France, April 2002, 16 pages.
http://www.irisa.fr/bibli/publi/pi/2002/1455/1455.html.
3.17 Mostefaoui A., Rajsbaum S. and Raynal M., A Versatile and Modular Consensus Protocol. *Proc. Int. Conference on Dependable Systems and Networks*, Washington D.C., pp. 364-373, June 2002.
3.18 Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., A Hierarchy of Conditions for Consensus Solvability. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press pp. 151-160, Newport (RI), August 2001.
3.19 Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Efficient Condition-Based Consensus. *8th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'01)*, Carleton Univ. Press, pp. 275-291, Val de Nuria (Catalonia, Spain), June 2001.
3.20 Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-Based Protocols for Set Agreement Problems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS, #2508, pp. 48-62, Toulouse (France), 2002.
3.21 Mostefaoui A., Raynal M. and Tronel F., The Best of Both Worlds: a Hybrid Approach to Solve Consensus. *Proc. IEEE Int. Conf. on Dependable Systems and Networks (DSN'00, previously FTCS)*, pp. 513-522, June 2000.
3.22 Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *JACM*, 27(2):228-234, 1980.
3.23 Veríssimo P., Traveling Through Wormholes: Meeting the Grand Challenge of Distributed Systems. *Proc. Int. Workshop on Future Directions in Distributed Computing (FuDiCo)*, pp. 144-151, Bertinoro (Italy), June 2002.

4. Lower Bounds for Asynchronous Consensus

Leslie Lamport

Microsoft Research

Consensus is usually expressed in terms of agreement among a set of processes. Instead, we characterize it in terms of three classes of agents:

Proposers A proposer can propose values.
Acceptors The acceptors cooperate in some way to choose a single proposed value.
Learners A learner can learn what value has been chosen.

In the traditional statement, each process is a proposer, an acceptor, and a learner. However, in a distributed client/server system, we can also consider the clients to be the proposers and learners, and the servers to be the acceptors.

The consensus problem is characterized by the following three requirements, where N is the number of acceptors and F is the number of acceptors that must be allowed to fail without preventing progress.

Nontriviality Only a value proposed by a proposer can be learned.
Safety At most one value can be learned.
Liveness If a proposer p, a learner l, and a set of $N - F$ acceptors are non-faulty and can communicate with one another, and if p proposes a value, then l will eventually learn a value.

Nontriviality and safety must be maintained even if at most M of the acceptors are malicious, and even if proposers are malicious. By definition, a learner is non-malicious, so the conditions apply only to non-malicious learners. A malicious acceptor by definition has failed, so the $N - F$ acceptors in the liveness condition do not include malicious ones. Note that M is the maximum number of failures under which safety is preserved, while F is the maximum number of failures under which liveness is ensured. These parameters are, in principle, independent. Hitherto, the only cases considered have been $M = 0$ (non-Byzantine) and $M = F$ (Byzantine). If malicious failures are expected to be rare but not ignorable, we may assume $0 < M < F$. If safety is more important than liveness, we might assume $F < M$.

The classic Fischer, Lynch, Paterson result [4.4] implies that no purely asynchronous algorithm can solve consensus. However, we interpret "can communicate with one another" in the liveness requirement to include a synchrony requirement. Thus, nontriviality and safety must be maintained in any case; liveness is required only if the system eventually behaves synchronously. Dwork, Lynch, and Stockmeyer [4.3] showed the existence of an algorithm satisfying these requirements.

Here are approximate lower-bound results for an asynchronous consensus algorithm. Their precise statements and proofs will appear later.

Approximate Theorem 1 If there are at least two proposers, or one malicious proposer, then $N > 2F + M$.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 22–23, 2003.

Approximate Theorem 2 If there are at least two proposers, or one malicious proposer, then there is at least a 2-message delay between the proposal of a value and the learning of that value.

Approximate Theorem 3 (a) If there are at least two proposers whose proposals can be learned with a 2-message delay despite the failure of Q acceptors, or there is one such possibly malicious proposer that is not an acceptor, then $N > 2Q + F + 2M$.

(b) If there is a single possibly malicious proposer that is also an acceptor, and whose proposals can be learned with a 2-message delay despite the failure of Q acceptors, then $N > \max(2Q + F + 2M - 2,\ Q + F + 2M)$.

These results are approximate because there are special cases in which the bounds do not hold. For example, Approximate Theorem 1 does not hold in the case of three distinct processes: one process that is a proposer and an acceptor, one process that is an acceptor and a learner, and one process that is a proposer and a learner. In this case, there is an asynchronous consensus algorithm with $N = 2$, $F = 1$, and $M = 0$.

The first theorem is fairly obvious when $M = 0$ and has been proved in several settings. For $M = F$, it was proved in the original Byzantine agreement paper [4.6]. The generalization is not hard. Results quite similar to the second theorem have also been proved in several settings [4.2]. The third theorem appears to be new.

The bounds in these theorems are tight. Castro and Liskov [4.1] present an algorithm satisfying the bounds of the first theorem. A future paper will describe a new version of the Paxos algorithm [4.5] that obtains agreement in two message delays under the weakest conditions allowed by the third theorem.

References

4.1 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. ACM, 1999.

4.2 Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

4.3 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

4.4 Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

4.5 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

4.6 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

5. Designing Algorithms for Dependent Process Failures*

Flavio Junqueira** and Keith Marzullo

University of California, San Diego
Department of Computer Science and Engineering
{flavio,marzullo}@cs.ucsd.edu

5.1 Introduction

Most fault-tolerant algorithms are designed assuming that out of n components, no more than t can be faulty. For example, solutions to the Consensus problem are usually developed assuming no more than t of the n processes are faulty, where "being faulty" is specialized by a failure model. We call this the *t of n assumption* (also known as threshold model). It is a convenient assumption to make. For example, bounds are easily expressed as a function of t: if processes can fail only by crashing, then the Consensus problem is solvable when $t < n$ if the system is synchronous and when $t < 2n$ if the system is asynchronous extended with a failure detector of the class $\diamond W$. [5.5, 5.1]

The t of n assumption is best suited for components that have identical probabilities of failure and that fail independently. For embedded systems built using rigorous software development this can be a reasonable assumption, but for most modern distributed systems it is not. Processes' failures can be correlated because, for example, they were built with the same software that contains bugs.

That failures can have different probabilities and can be dependent is not a novel observation. The continued popularity of the t of n assumption, however, implies that it is an observation that is being overlooked by protocol designers. In Section 5.2 we propose an abstraction that exposes dependent failure information for one to take advantage of in the design of a fault-tolerant algorithm. Like the t of n assumption, it is expressed in a way that hides its underlying probabilistic nature in order to make it more generally applicable. Our preliminary results show that our techniques are promising. We discuss briefly some of these results in Section 5.3.

There has been some work in providing abstractions more expressive than the t of n assumption. The hybrid failure model (for example, [5.8]) generalizes the t of n assumption by providing a separate t for different classes of failures. Using a hybrid failure model allows one to design more efficient protocols by having sufficient replication for masking each failure class. It is still based on failures in each class being independent and identically distributed.

Byzantine Quorum systems have been designed around the abstraction of a *Fail-prone System* [5.6]. This abstraction allows one to define quorums that take correlated failures into account, and it has been used to express a sufficiency condition for replication. Our work can be seen as generalizing this work that has been done for Quorum

* This work was developed in the context of the RAMP project, supported by DARPA as project number N66001-01-1-8933.

** The author is partially supported by a scholarship from CAPES.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 24–28, 2003.

Systems in that we give replication requirements for dependent failures for another set of problems, namely Consensus. Hirt and Maurer generalize the t of n assumption in the context of secure multi-party computation. [5.3] With *Collusion* and *Adversary* structures, they characterize subsets of players that may collaborate to either disrupt the execution of a protocol or obtain private data of other players. Because our focus is not on malicious behavior, the abstractions we propose are defined differently and fulfill a distinct set of properties.

5.2 Replacing the t of n Assumption

Let a distributed system be defined as a set Π of processes interconnected pairwise by channels. These channels are used to exchange messages and this is the only way of communication between two processes in Π. Assuming that once a process fails it does not recover, it is necessary to have enough replicas so that in every execution there are enough correct processes to accomplish some given task. An example of such a task is reaching agreement.

For such a system, we define a *core* as a minimal subset of processes such that in every execution at least one process is correct. By minimal, we mean that a core contains just enough processes to satisfy some notion of reliability. This notion of reliability can be expressed in terms of probabilities, and a core in this case contains just enough processes to make negligible the probability that every process in this subset fails in every execution. Alternatively, cores can be determined according to properties of the system. For example, one can leverage the fact that some processes are more reliable than others.

A system can have several cores. Consider a set S of processes that intersects with every core in such a system. There is some execution in which all the processes in S do not fail. If S is such that the removal of a single process makes its intersection with some core empty, then this subset is called a *survivor set*. Thus, by assumption, there is at least one survivor set that contains only correct processes in every execution.

It is important to observe that both cores and survivor sets describe the worst-case failure scenarios of a system. They are therefore equivalent to *fail-prone systems* as defined by Malkhi and Reiter in their work on Byzantine Quorum Systems. [5.6] For a given system configuration, one can obtain the fail-prone system by taking the complement of every survivor set.

Cores and survivor sets generalize subsets under the t of n assumption of size $t+1$ and $n-t$, respectively. Thus, we conjecture that fault-tolerant algorithms that have subsets of size $t+1$ and $n-t$ embedded in its characterization are translatable to our model by simply exchanging these subsets with cores and/or survivor sets. Intuitively, performance may be impacted by such a change, but correctness is preserved.

Defining these abstractions more formally, let R be a rational number expressing a desired reliability, and $r(x)$, $x \subseteq \Pi$, be a function that evaluates to the reliability of the subset x. We then have the following definitions:

Definition 5.2.1. *Given a set of processes Π and target degree of reliability $R \in [0,1] \cap Q$, c is a* core *of Π if and only if: 1) $c \subseteq \Pi$; 2) $r(c) \geq R$; 3) $\forall p \in c$, $r(c - \{p\}) < R$.*

Definition 5.2.2. *Given a set of processes Π and a set of cores C_Π, s is said to be a survivor set if and only if: 1)$s \subseteq \Pi$; 2) $\forall c \in C_\Pi$, $s \cap c \neq \emptyset$; 3) $\forall p_i \in s$, $\exists c \in C_\Pi$ such that $p_i \in c$ and $(s - \{p_i\}) \cap c = \emptyset$.*

We define C_Π and S_Π to be the set of cores and survivor sets of Π respectively.

To illustrate the utilization of cores and survivor sets, consider a six-computer system, each computer representing a process. Two of them are robust computers (ph_1 and ph_2), located in different sites and managed by different people. The other four computers (pl_1, pl_2, pl_3, pl_4) are located in the same room and are old machines managed by the same person. If machines only fail by crashing due to power outages and software bugs, then we can safely assume that ph_1 and ph_2 are reliable processes and fail independently, whereas pl_1, pl_2, pl_3, and pl_4 are less reliable and their failures are highly correlated. To simplify the analysis, we assume that the failure of process pl_i implies the failure of pl_j, $i \neq j$. In order to maximize reliability, a core is hence composed of 3 processes: ph_1, ph_2, and pl_i, $1 \leq i \leq 4$. Note that adding another process pl_j, $i \neq j$, does not increase the reliability of a core, and the core is not minimal in this case. Under these assumptions, we construct the set of cores and survivor sets for this system as follows:

Example 5.2.1. $\Pi = \{ph_1, ph_2, pl_1, pl_2, pl_3, pl_4\}$, $C_\Pi = \{\{ph_1, ph_2, pl_1\}, \{ph_1, ph_2, pl_2\}, \{ph_1, ph_2, pl_3\}, \{ph_1, ph_2, pl_4\}\}$, $S_\Pi = \{\{ph_1\}, \{ph_2\}, \{pl_1, pl_2, pl_3, pl_4\}\}$

5.3 A Summary of Results

We applied cores and survivor sets to the Consensus problem in different system models. First, we considered the synchronous model with crash process failures. In such a model, Consensus is known to be solvable for any number of failures. Assuming that processes do not recover once they fail, this is equivalent to saying that the set of cores is not empty ($C_\Pi \neq \emptyset$). For synchronous systems in which both information about cores is available and processes fail dependently, we are able to reduce latency and message complexity. [5.4]

In an asynchronous system, Consensus is known not to be solvable even with a single crash failure. Extending this model with a failure detector, however, enables a solution for Consensus. In particular, $\diamond\mathcal{W}$ is the weakest class of failure detectors which enables Consensus to be solved. [5.1] Assuming an asynchronous model extended with a failure detector of the class $\diamond\mathcal{W}$, consider the following two properties:

Property 5.3.1. (**Crash Partition**) Every partition (A, B) of Π is such that either A or B contain a core.

Property 5.3.2. (**Crash Intersection**) S_Π forms a coterie.

As shown in [5.4], these properties are equivalent, and they are necessary and sufficient to solve Consensus in such a model. Since Crash Intersection is also the requirement for a quorum system to exist, this result shows in addition that the implementability of quorums are necessary and sufficient.

The crash model, however, is not adequate for systems that are susceptible to other types of process failures besides crash. For such systems, algorithms are often designed with the assumption of arbitrary failures. By assuming a weaker failure model, the replication requirement changes. Consider the following properties:

Property 5.3.3. (**Byzantine Partition**) For every partition (A, B, C) of Π, at least one of A, B, or C contains a core.

Property 5.3.4. (**Byzantine Intersection**) $\forall s_i, s_j \in S_\Pi, \exists c_k \in C_\Pi$: $c_k \subseteq (s_i \cap s_j)$.

We show in [5.4] that these properties are equivalent, and they are necessary and sufficient to solve Consensus in both synchronous systems and asynchronous systems extended with a failure detector of the class $\diamond\mathcal{M}$, where $\diamond\mathcal{M}$ is the class failure detectors satisfying Mute Completeness and Eventual Weak Accuracy. [5.2]

In all four models, we were able to use Consensus protocols proposed in the literature by applying simple modifications to them. This is in fact an important feature of our model: it does not invalidate the work done under the t of n assumption (at least not for Consensus, as our results show).

Assuming arbitrary failures, systems in which cores do not have uniform size require less replication. Consider the following example:

Example 5.3.1. $\Pi = \{p_a, p_b, p_c, p_d, p_e\}$, $C_\Pi = \{\{p_a, p_b, p_c\}, \{p_a, p_d\}, \{p_a, p_e\}, \{p_b, p_d\}, \{p_b, p_e\}, \{p_c, p_d\}, \{p_c, p_e\}, \{p_d, p_e\}\}$, $S_\Pi = \{\{p_a, p_b, p_c, p_d\}, \{p_a, p_b, p_c, p_e\}, \{p_a, p_d, p_e\}, \{p_b, p_d, p_e\}, \{p_c, p_d, p_e\}\}$

The system configuration in Example 5.3.1 satisfies Byzantine Intersection, and hence Consensus is solvable for this system. Under the t of n assumption, at least seven processes are required ($n = 7$), because in the worst case there are two faulty processes ($t = 2$) and it is a well-known result that $n > 3t$ for synchronous systems assuming arbitrary failures.

5.4 Future Directions

There are several questions that we are still unable to answer. It is not clear, for example, that cores and survivor sets are good abstractions for real systems. In the worst case, a system has an exponential number of cores and survivor sets, and consequently extracting cores or survivor sets may require exponential time. For some system configurations, however, we observed that there are simple heuristics that determine cores in polynomial time on the number of processes. In addition, there are two important facts: 1) fault-tolerant systems often do not have a large number of processes; 2) for Consensus, it is not necessary to extract all possible cores and survivor sets, but only a sufficient number of subsets in order to satisfy replication requirements. In both cases, even if there is an exponential number of cores or survivor sets, the utilization of our abstractions is perhaps practical.

To determine the cores or the survivor sets, one can use intrinsic characteristics of a system, such as shared resources. If such characteristics are known, then the computation of cores or survivor sets can be accomplished using off-line techniques such as fault-tree analysis [5.7]. Otherwise, there has to be a mechanism to infer failure probabilities and correlations on-line. We are not aware, however, of a technique to find cores or survivor sets on the fly.

On the theoretical side, we have used cores and survivor sets to obtain results only for Consensus. Although Consensus is an important primitive, similar results for a broader set of problems are necessary to enable a wide applicability of our techniques.

References

5.1 T. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

5.2 A. Doudou and A. Schiper. Muteness Detectors for Consensus with Byzantine Processes. In *Proceedings of the 17th ACM Symposium on Principle of Distributed Computing*, page 315, Puerto Vallarta, Mexico, July 1998.

5.3 M. Hirt and U. Maurer. Complete Characterization of Adversaries Tolerable in Secure Multy-Party Computation. In *ACM Symposium on Principles of Distributed Computing*, pages 25–34, Santa Barbara, California, 1997.

5.4 F. Junqueira and K. Marzullo. Consensus for Dependent Process Failures. Technical report, UCSD, La Jolla, CA, September 2002. http://www.cs.ucsd.edu/users/flavio/Docs/Gen.ps.

5.5 I. Keidar and S. Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, MIT, May 2001.

5.6 D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *29th ACM Symposium on Theory of Computing*, pages 569–578, May 1997.

5.7 Y. Ren and J. B. Dugan. Optimal Design of Reliable Systems Using Static and Dynamic Fault Trees. *IEEE Transactions on Reliability*, 47(3):234–244, December 1998.

5.8 P. Thambidurai and Y.-K. Park. Interactive Consistency with Multiple Failure Modes. In *IEEE 7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, Ohio, October 1988.

6. Comparing the Atomic Commitment and Consensus Problems

Bernadette Charron-Bost

LIX, École polytechnique 91128 Palaiseau Cedex, France
charron@lix.polytechnique.fr

6.1 Two Agreement Problems: Consensus and NB-AC

Reaching agreement in a distributed system is a fundamental issue of both theoretical and practical importance. Consensus and non-blocking atomic commitment are two well-known versions of this paradigm. The *Consensus* problem considers a fixed collection of processors each of which has an initial value drawn from some domain V, and processors must eventually decide on the same value[1]; moreover, the decision value must be the initial value of some processor. The *non-blocking atomic commitment (NB-AC)* problem arises in distributed database systems to ensure the consistent termination of transactions. Each process that participates in the processing of a database transaction arrives at an initial opinion (vote) about whether to *commit* the transaction or *abort* it. Processes must eventually reach a common decision (commit or abort). The decision to commit may be reached only if all processes initially vote to commit. In this case, "commit" must be reached if there is no failure.

As observed by Hadzilacos in [6.16], the binary Consensus and NB-AC specifications are very similar. They only differ in their *validity conditions*, i.e., the conditions describing the decision values that are permitted: in the Consensus problem, the possible decision values are related to the initial values only, contrary to the NB-AC problem for which the decision values depend on both input values *and* failure pattern. A closer look at these validity conditions shows that they cannot be compared. So in the strict standpoint of specification, Consensus and NB-AC are incomparable. But what about their solvability and their complexity? In other words, is one problem harder to solve than the other?

This note is devoted to various results on this question I have recently obtained, most of them in collaboration, for the *crash failure model*. Firstly, I recall the ones in my joint work with F. Le Fessant [6.6] concerning synchronous systems: we have proved that in this setting, Consensus and NB-AC are two very similar problems which are both solvable whatever the environment, and with the same time complexity. Then I give some impossibility results for the NB-AC problem in non-synchronous systems in which Consensus is solvable: in such systems, NB-AC is thereby a harder problem than Consensus. Finally, I present two theorems due to S. Toueg and myself which show that, in asynchronous systems, Consensus and NB-AC are incomparable in almost all environments. I complete those theorems by examining what information about failures is necessary and sufficient to solve NB-AC in asynchronous systems.

[1] More exactly, the problem that is considered here is *uniform Consensus*: no two (correct or faulty) processes can decide differently.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 29–34, 2003.

6.2 When NB-AC and Consensus Are Similar

In synchronous systems, Consensus and NB-AC are both solvable no matter how many processes fail, and so are equivalent in terms of solvability. Actually they are equivalent in a stronger sense: in the presence of up to t failures, $t + 1$ rounds[2] are necessary and sufficient in the worst case execution to solve Consensus as well as NB-AC [6.19]. Following [6.9], one refines this comparison by discriminating runs according to the number of failures f that actually occur ($0 \leq f \leq t$). Charron-Bost and Schiper [6.7] proved that both Consensus and NB-AC require at least $f + 2$ rounds if $f < t - 1$, and only $f + 1$ rounds if $t - 1 \leq f \leq t$ (subsequently, Keidar and Rajsbaum [6.17] have given another proof of this lower bound).

One may wonder whether these lower bounds for early deciding Consensus and NB-AC algorithms are tight. A Consensus algorithm is presented in [6.7] that achieves these lower bounds. Unfortunately, this algorithm cannot be easily adapted for the NB-AC problem. As noticed by Lamport in [6.18], "a [popular] belief is that a three-phase commit protocol *à la* Skeen [6.20] is needed" for solving NB-AC. It is not exactly clear what that means, but it seems to imply that at least three rounds are required in failure-free runs for deciding. The $f + 2$ lower bound would be therefore weak in the case of NB-AC. In [6.6], F. Le Fessant and I show that this intuition is incorrect: we devise a general algorithm for both Consensus and NB-AC which achieves the general lower bounds for early deciding established in [6.7]. This proves that even when considering the question of early deciding, Consensus and NB-AC remain two equivalent problems.

6.3 When NB-AC Is Harder than Consensus

As observed by Gray [6.12], the impossibility of NB-AC in an asynchronous system can be established by quite a simple argument even if the maximum number of failures is one, and so does not require the subtle proof given in [6.11] for Consensus. The argument is based on the following two points:
(1) In an asynchronous system, a process may be so slow that it executes its first step after the other processes have decided; the decision value is the same as if the slow process had initially crashed.
(2) The decision value of any run with one initial crash is necessarily "abort".
This argument yields the following three impossibility results in systems subject to a single crash failure:
(1) Even with *initial* failure, NB-AC cannot be solved in an asynchronous system.
(2) There is no asynchronous randomized algorithm solving NB-AC.
(3) NB-AC cannot be solved in any of the partially synchronous models described in [6.10][3].

[2] In a synchronous algorithm, a run proceeds in *rounds*: in each round, every process sends messages to all processes, receives all the messages sent to it in the round, and then changes its state. Time complexity is then measured in terms of the number of rounds necessary to produce all the required outputs.

[3] Using the same simple argument, Guerraoui [6.13] showed this result in some partially synchronous systems defined in terms of unreliable failure detectors [6.3]. Obviously, the argument applies to the more general and realistic partially synchronous models of [6.10].

On the other hand, Consensus is solvable in each of these system models if a majority of processes is correct (see [6.11, 6.1, 6.10], respectively). In that sense, NB-AC is harder to solve than Consensus.

6.4 When NB-AC and Consensus Are Incomparable

In a joint work with S. Toueg [6.8], we compared the NB-AC and Consensus problems in the asynchronous setting. We formally defined the notion of *problem reduction*, and proved that:
(1) Consensus is not reducible to NB-AC except in an environment with at most one process failure, in which case Consensus is indeed reducible to NB-AC.
(2) Conversely, NB-AC is never reducible to Consensus.
The proof of the first irreducibility result consists in showing that the impossibility proof of Consensus in [6.11] still applies when processes can invoke "Vote" and "Decide" primitives to achieve NB-AC. In the same way, we show that NB-AC is not reducible to Consensus by extending the impossibility proof sketched in Section 6.2 to systems in which processes can use the "Propose" and "Decide" primitives of Consensus.

In [6.4, 6.5], I have completed these irreducibility results by showing that in an asynchronous system equipped with the *binary flag* failure detector $\mathcal{BF}$ described in the next section, NB-AC becomes reducible to Consensus.

6.5 The Weakest Failure Detector for Solving NB-AC

One way to refine the irreducibility results between Consensus and NB-AC is to compare the information about failures that is necessary to make each of these two problems solvable in an asynchronous environment. In other words, we compare the failure detectors needed to solve Consensus and NB-AC.

The papers [6.3, 6.2] focused on the Consensus problem, and introduced a failure detector denoted Ω such that the output of the failure detector module of Ω at process p is a single process. Intuitively, when q is the output of Ω at p, the failure detector module of Ω at process p currently considers q to be correct; then, we say that p *trusts* q. The Ω failure detector satisfies the following property: there is a time after which all the correct processes always trust the same correct process. Chandra and Toueg [6.3] showed that Consensus can be solved in an asynchronous system equipped with Ω if a majority of processes are correct. Conversely, Chandra, Hadzilacos, and Toueg [6.2] proved that in any environment, if a failure detector $\mathcal{D}$ can be used to solve Consensus, then Ω can be "extracted" from $\mathcal{D}$. Thus Ω is the *weakest* failure detector for solving Consensus in asynchronous systems with a majority of correct processes. But what about the NB-AC problem? In [6.4, 6.5], I addressed this question and I now sketch the results I obtained. Similar results have been independently given by Guerraoui [6.14].

Because of the validity condition, we can easily extract failure informations from NB-AC. More precisely, the NB-AC problem naturally yields the *binary flag* failure detector, denoted $\mathcal{BF}$, the output of which is the flag **NF** (for No Failure) or **F** (for Failures). This failure detector satisfies the following three properties:

(1) If all processes are correct, then no flag is ever **F**.
(2) If the flag of some process is **F**, then it remains **F** forever.
(3) If some process crashes, then there is a time after which all the flags are **F**.
In [6.4], I show that in any asynchronous environment, we can extract $\mathcal{BF}$ from any failure detector that can be used to solve NB-AC. The proof relies on a simple idea: if all processes initially vote "commit", then any NB-AC algorithm decides to commit if and only if there is no failure, and so can be used to detect whether some failures occur. Conversely, if there is at most one failure, then $\mathcal{BF}$ is sufficient to make NB-AC solvable. Thus, $\mathcal{BF}$ is the weakest failure detector for solving NB-AC in asynchronous systems with at most one failure. Note that in an environment with at most one failure, $\mathcal{BF}$ is actually equivalent to the perfect failure detector $\mathcal{P}$ (which eventually detects all the failures that occur and makes no false detection). Consequently, $\mathcal{P}$ is the weakest failure detector to solve NB-AC if there is at most one crash[4].

Even if tolerating more than one failure is obviously harder than tolerating only one failure, the latter result does *not* state that $\mathcal{P}$ is necessary to solve NB-AC in general. Indeed, as explained in the previous section, Consensus is reducible to NB-AC in a system equipped with $\mathcal{BF}$. Combining this reduction with the main result of [6.3], we deduce that if there is a majority of correct processes, then the "composition" of the Ω and $\mathcal{BF}$ failure detectors can be used to solve NB-AC. Now this composed failure detector is clearly weaker than $\mathcal{P}$ with more than one failure.

Finally, I prove in [6.5] that the NB-AC problem cannot be solved using $\mathcal{BF}$ if the maximum number of failures is greater than 1. It is not clear whether there exists a weakest failure detector for solving NB-AC in the general case, and if it exists, it is an open problem to determine it[5].

6.6 Expected Impacts of These Results

These results, somehow surprising and counterintuitive, might be interesting for future works in this area from both a theoretical and practical viewpoint.

Theoretical: The reason why NB-AC is impossible to achieve in non-perfectly synchronous systems (see Section 6.3) is rather trivial and comes from the poor statement of NB-AC. This suggests to refine the NB-AC specification, more precisely the validity condition with for example, timing or probabilities clauses.

Moreover, this work provides another instance of the lack of "stability" of results in distributed computing: slight modifications in problem statements or system models may result in quite important discrepancies in solvability and complexity. Again this demonstrates that a rigorous treatment is especially important in the area of distributed algorithms.

[4] Since NB-AC is reducible to the *terminating reliable broadcast* problem (*TRB*), this implies and so formally proves that $\mathcal{P}$ is the weakest failure detector to solve TRB if there is at most one failure.

[5] After the submission of this paper, reference [6.15] has been pointed out to me: there, Guerraoui and Kouznetsov actually determine the weakest failure detector for solving NB-AC in a restricted subclass of failure detectors. To the best of my knowledge, the problem is thus still open.

Practical: In view of Sections 6.3 and 6.4, there is no evident basic agreement block which allows to solve a whole set of agreement problems including Consensus and NB-AC in asynchronous systems. In the same way, a failure detector which makes solvable both Consensus and NB-AC must at least combine the completeness and accuracy properties of both Ω and $\mathcal{BF}$, and so is quite strong.

Concerning the synchronous case, it would be worthy to implement the optimal algorithm described in [6.6] which achieves both Consensus and NB-AC, since it makes decisions as soon as possible. In particular, decision values are available by the end of the second round in failure free runs (that is the most frequent case).

Acknowledgments

I am grateful to Leslie Lamport and the anonymous referee for their valuable comments on the first version of this note. Special thanks to Rachid Guerraoui for communicating me his recent related papers.

References

6.1 M. Ben-Or, M. (1983): Another advantage of free choice: Completely asynchronous agreement protocols. PODC, 27–30
6.2 Chandra, T. D., Hadzilacos, V, Toueg, S. (1996): The weakest failure detector for solving consensus. JACM **43**(4), 685–722
6.3 Chandra, T. D., Toueg, S. (1996): Unreliable failure detectors for asynchronous systems. JACM. **43**(2), 225–267
6.4 Charron-Bost, B. (2001): Agreement problems in fault-tolerant distributed systems. (SOFSEM, LNCS, vol. 2234), 10–32
6.5 Charron-Bost, B. (2001): The weakest failure detector for solving atomic commitment. Unpublished Notes
6.6 Charron-Bost, B., Le Fessant, F. (2002): Validity conditions in agreement problems and time complexity. Tech. Rep. RR-4526, INRIA-Rocquencourt
6.7 Charron-Bost, B., Schiper, A. (2000): Uniform consensus is harder than consensus. Tech. Rep. DSC/2000/028, EPFL
6.8 Charron-Bost, B., Toueg, S. (2001): Comparing the atomic commitment and consensus problems. Unpublished Notes
6.9 Dolev, D., Reischuk, R., Strong, H. R. (1990): Early stopping in Byzantine agreement. JACM **37**(4), 720–741
6.10 Dwork, C., Lynch, N. A., Stockmeyer, L. (1988): Consensus in the presence of partial synchrony. JACM **35**(2), 288–323
6.11 Fischer, M. J., Lynch, N. A., Paterson, M. S. (1985): Impossibility of distributed consensus with one faulty process. JACM **32**(2), 374–382
6.12 Gray, J. (1987): A comparison of the byzantine agreement problem and the transaction commit problem. (Fault Tolerant Distributed Computing, LNCS vol. 448), 10–17
6.13 Guerraoui, R. (1995): Revisiting the relationship between non-blocking atomic commitment and consensus. (WDAG, LNCS vol. 972), 87–100
6.14 Guerraoui, R. (2002): Non-blocking atomic commitment in asynchronous systems with failure detectors. Distributed Computing **15**(1)
6.15 Guerraoui, R., Kouznetsov, P. (2002): On the weakest failure detector for non-blocking atomic commit. International Conference on Theoretical Computer Science

6.16 Hadzilacos, V. (1987): On the relationship between the atomic commitment and consensus problems. (Fault Tolerant Distributed Computing, LNCS vol. 448), 201–208
6.17 Keidar, I., Rajsbaum, S. (2002): A simple proof of the uniform consensus synchronous lower bound. IPL, To appear
6.18 Lamport, L. (2000): Lower bounds on consensus. Unpublished manuscript
6.19 Merritt, M. J (1985): Unpublished Notes
6.20 Skeen, D. (1982): Nonblocking commit protocols. ACM SIGMOD Conf. on Management of Data, 133–147

7. Open Questions on Consensus Performance in Well-Behaved Runs

Idit Keidar[1] and Sergio Rajsbaum[2]

[1] Department of Electrical Engineering, The Technion, Haifa 32000, Israel
idish@ee.technion.ac.il
[2] Instituto de Matemáticas, UNAM, Mexico
rajsbaum@math.unam.mx

7.1 Consensus in the Partial Synchrony Model

We consider the *consensus* problem in a message-passing system where processes can crash: Each process has an input, and each correct process must decide on an output, such that all correct processes decide on the same output, and this output is the input of one of the processes. Consensus is an important building block for fault-tolerant systems.

It is well-known that consensus is not solvable in an asynchronous model even if only one process can crash [7.13]. However, real systems are not completely asynchronous. Some partially synchronous models [7.12, 7.10] where consensus is solvable better approximate real systems. We consider a *partial synchrony* model defined as follows [7.12][1]: (1) processes have bounded drift clocks; (2) there are known bounds on processing times and message delays; and (3) less than half of the processes can crash. In addition, this model allows the system to be *unstable*, where the bounds in (2) do not hold for an unbounded but finite period, but it must eventually enter a *stable* period where the bounds do hold. A consensus algorithm for the partial synchrony model never violates safety, and guarantees liveness once the system becomes stable. Algorithms for this model are called indulgent in [7.16].

What can we say about the running time of consensus algorithms in a partial synchrony model? Unfortunately, even in the absence of failures, any consensus algorithm in this model is bound to have unbounded running times, by [7.13]. In this paper we propose a performance metric for algorithms in the partial synchrony model, and suggest some future research directions and open problems related to evaluating consensus algorithms using this metric.

In practice there are often extensive periods during which communication is timely and processes do not experience undue delays. That is, many runs are actually stable. In such runs, failures can be detected accurately using time-outs. We are interested in the running time of consensus algorithms for partially synchronous models under such benign circumstances. We focus on runs in which, from the very beginning, the network is stable. We will call such runs *well-behaved* if they are also failure-free. Since well-behaved runs are common in practice, algorithm performance under such circumstances is significant. Note that an algorithm cannot know a priori that a run is going to be well-behaved, and thus cannot rely upon it.

[1] This is very close to the model called *timed-asynchronous* in [7.10].

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 35–39, 2003.

We will evaluate algorithm running times in terms of the number of communication *steps* (some other authors call them *rounds*) an algorithm makes in the worst case before all processes decide in well-behaved, and sometimes more generally, stable runs.

7.2 Algorithms and Failure Detectors

There are several algorithms for partially synchronous models that decide in two steps in well-behaved runs, which as we shall see, is optimal. Most notably, (an optimized version of) Paxos [7.20], and others such as [7.24, 7.17].

Many of these consensus algorithms use oracle unreliable failure detectors [7.6] that abstract away the specific timing assumptions of the partial synchrony model, instead of directly using the specific timing bounds of the underlying system. The unreliable failure detectors, in turn, can be implemented in the partial synchrony model. Our interest in understanding the performance of consensus algorithms under common scenarios takes us to questions on the performance of failure detectors in the partial synchrony model. Notice that such unreliable failure detectors can provide arbitrary output for an arbitrary period of time (while the system is unstable), but eventually provide some useful semantics (when the system becomes stable).

Chandra and Toueg [7.6] define several classes of failure detectors whose output is a list of *suspected* processes. The natural way of detecting failures using timeouts in a partial synchrony model yields a failure detector called *eventually perfect*, or $\diamond\mathcal{P}$, that satisfies the following two properties. *Strong completeness:* there is a time after which every correct process permanently suspects every crashed process. *Eventually strong accuracy:* from some point on, every correct process is not suspected. It is remarkable that although $\diamond\mathcal{P}$ is indeed sufficient to solve consensus, it is not necessary. The so called $\diamond\mathcal{S}$ failure detector has been shown to be the weakest for solving consensus [7.5], and it is strictly weaker than $\diamond\mathcal{P}$. A $\diamond\mathcal{S}$ failure detector satisfies strong completeness and *Eventually weak accuracy:* there is a correct process p such that there is a time after which p is not suspected by any correct process.

Another example of a failure detector class is the *leader election* service $\diamond\Omega$ [7.5][2]. The output of a failure detector of class $\diamond\Omega$ is the identifier of one process, presumed to be the leader. Initially, a failure detector of this class can name a faulty process as the leader, and can name different leaders at different processes. However, eventually it must give all the processes the same output, which must be the identifier of a correct process. In [7.5], it is shown that $\diamond\Omega$ is equivalent to $\diamond\mathcal{S}$, and hence weaker than $\diamond\mathcal{P}$. In a *well-behaved* run, a $\diamond\Omega$ failure detector announces the same correct leader at all the processes from the beginning of the run. We pose the following **open problems**:

- What is the weakest timing model where $\diamond\mathcal{S}$ and/or $\diamond\Omega$ are implementable but $\diamond\mathcal{P}$ is not?
- Is building $\diamond\mathcal{P}$ more "costly" than $\diamond\mathcal{S}$ and/or $\diamond\Omega$? Under what cost metric?

[2] Originally called Ω, but we add the $\diamond$ prefix for consistency with the notation of the other failure detectors.

7.3 Lower Bound

Distributed systems folklore suggests that every fault tolerant algorithm in a partial synchrony model must take at least two steps before all the processes decide, even in well-behaved runs. We formalized and proved this folklore theorem in [7.19]. Specifically, we showed that any consensus algorithm for a partial synchrony model where at least two processes can crash will have at least one well-behaved run in which it takes two steps for all the processes to decide.

This is in contrast to what happens in a *synchronous* crash failure model: in this model there are consensus algorithms that, in failure-free runs, decide within one step. More generally, early deciding algorithms have all the processes decide within $f + 1$ steps in every run involving f crash failures. This is optimal: every consensus algorithm for the synchronous crash failure model will have runs with f failures that take $f + 1$ steps before all the processes decide [7.21].

Why then do consensus algorithms for the partial synchrony model require two steps in well-behaved runs, that look exactly like runs of a synchronous system? The need for an additional step stems from the fact that, in the partial synchrony model, a correct process can be mistaken for a faulty one. This requires consensus algorithms to avoid disagreement with processes that seem faulty. In contrast, consensus in the synchronous model requires only that correct processes agree upon the same value, and allows for disagreement with faulty ones. The *uniform consensus* problem strengthens consensus to require that every two processes (correct or faulty) that decide must decide on the same value. Interestingly, uniform consensus requires two steps in the absence of failures in the synchronous model, as long as two or more processes can crash, as proven in [7.7, 7.19]. The two step lower bound for consensus in the partial synchrony model stems from the fact that in this model, any algorithm that solves consensus, also solves uniform consensus [7.15].

The observation above – that an additional step is needed in order to avoid disagreement with processes that are incorrectly suspected to have failed – suggests the use of an *optimistic* approach to relax the requirement that two correct processes never decide on different values. An algorithm solving variant of consensus that is allowed to violate agreement in cases of false suspicions, can always terminate in a single step. Such a service can be useful for optimistic replication, if false suspicions are rare, and if the inconsistencies introduced in cases of false suspicions can later be detected and reconciled (or rolled-back). Group communication systems [7.9] such as Isis [7.4] take such an optimistic approach: they implement totally ordered multicast in a single step. If a correct process is suspected, inconsistencies can occur. Isis resolves such inconsistencies by forcing the suspected process to fail and re-incarnate itself as a new process, whereby it adopts the state of the other replicas. Another example is optimistic *atomic commitment* [7.18], which can lead to *rollback* in case of false suspicions. Our observation suggests that the likelihood of inconsistencies depends on the frequency of false suspicions, and leads to the following **open problem**:

– Formalize the notion of *likelihood of inconsistencies* for an application of optimistic consensus (or of a group communication system), and quantify its cost. Then, use this measure to analyze the cost-effectiveness of employing optimism in a particular setting.

7.4 Extensions and Future Directions

As a first extension, it is interesting to look at weaker notions of well-behavedness. E.g., consider runs where the system is stable but there are f failures. All the algorithms mentioned above can take as many as $2f+2$ steps in such runs. Dutta and Guerraoui [7.11] present an algorithm that decides in $t + 2$ steps in stable runs, where t is the maximum number of possible failures, and show a corresponding lower bound. It remains an **open problem** to devise an algorithm that takes $f + 2$ steps for every $f <= t$. Such an algorithm would be optimal.

Another **future direction** is to consider runs that are initially unstable and then become stable, and to find lower and upper bounds on the time it takes to reach decision once stability is reached. A similar **open question** can be posed for asynchronous *self-stabilizing* algorithms.

So far, we have focused on algorithms and lower bounds stated in message-passing models. Similar algorithms exist in shared memory models, e.g., a version of Paxos for a shared memory model with read-write registers [7.14]; and another for a model with infinitely many processes and shared read-modify-write registers [7.8]. It is interesting to consider the meaning of our performance metric, namely, number of steps in well-behaved runs, in shared memory models. A common notion of "well-behavedness" in shared memory models is the absence of contention [7.1]. At a first glance, this may seem unrelated to our notion of well-behaved runs being synchronous failure-free runs. However, a deeper look reveals that the two notions are indeed related: in shared-memory versions of Paxos, absence of contention occurs when there is a single leader trying to propose a consensus value. This is akin to having a single leader in message-passing versions of Paxos. That is, the failure detector $\diamond\Omega$ enforces the absence of contention. Given this observation, we pose the following **open problems**:

- Are there formal generic transformations from well-behaved runs as defined herein to contention-free runs in different shared memory models?
- Are there generic complexity-preserving reductions from message passing to shared memory models? Do they preserve the communication step metric and some notion of "well-behaved runs"?

We have analyzed performance in terms of the number of steps an algorithm takes. Lower and upper bounds on the actual running time of consensus were postulated in a variant of the partial synchrony model that is stable from the outset [7.2]. An **open question** is to revisit these bounds, perhaps using layered analysis in the sense of [7.22], and to extend them to well-behaved runs.

Finally, [7.3] points out limitations of the communication steps performance measure in environments like the Internet, where message delays exhibit high variability. The running time of a communication step can depend heavily on the number of messages sent in the step, and the particular links over which messages are sent. This is due to the high variability of message delays (see, e.g., the analysis in [7.23]). A **future direction** is finding performance metrics that better capture algorithm performance in practice.

References

7.1 R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Inform. Comput.*, 126(1):62–73, 1996.

7.2 H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122–152, 1994.

7.3 O. Bakr and I. Keidar. Evaluating the running time of a communication round over the Internet. In *ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 243–252, July 2002.

7.4 K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comp. Sys.*, 9(3):272–314, 1991.

7.5 T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

7.6 T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

7.7 B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus (extended abstract). Tech Rep DSC/2000/028, EPFL, Switzerland, May 2000. Submitted to *J. Algorithms.*

7.8 G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *21st ACM Symp. on Prin. of Dist. Comp. (PODC)*, Jul 2002.

7.9 G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Comp. Surveys*, 33(4):1–43, December 2001.

7.10 F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Par. Dist. Sys.*, pages 642–657, June 1999.

7.11 P. Dutta and E. Guerraoui. The inherent price of indulgence. In *21st ACM Symp. on Prin. of Dist. Comp. (PODC)*, July 2002.

7.12 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr 1988.

7.13 M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, Apr 1985.

7.14 E. Gafni and L. Lamport. Disk paxos. In *Int'l Symp. on DISt. Comp. (DISC)*, pp. 330–344, 2000.

7.15 R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Int'l Wshop on Dist. Alg. (WDAG)*, pp. 87–100. Sep 1995. LNCS 972.

7.16 R. Guerraoui. Indulgent Algorithms. In *19th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 289–297. 2000.

7.17 M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Dist. Comp.*, 12(4), 1999.

7.18 R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and S. Arevalo. A low latency non-blocking commit protocol. In *Int'l Symp. on DISt. Comp. (DISC)*, Oct 2001.

7.19 I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 2002. To appear.

7.20 L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, May 1998.

7.21 L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Tech Rep 62, SRI Int'l, Apr 1982.

7.22 Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SIAM J. Comp.*, 31(4):989–1021, 2002.

7.23 S. Rajsbaum and M. Sidi. On the performance of synchronized programs in distributed networks with random processing times and transmission delays. *IEEE Trans. Par. Dist. Sys.*, 5(9):939–950, 1994.

7.24 A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Dist. Comp.*, 10(3):149–157, 1997.

7.25 D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD Int'l Symp. on Management of Data*, pp. 133–142, 1981.

8. Challenges in Evaluating Distributed Algorithms

Idit Keidar

Department of Electrical Engineering, The Technion, Haifa 32000, Israel
idish@ee.technion.ac.il

8.1 Introduction

Theoretical evaluation of performance, availability, and reliability of distributed algorithms is always based on models and metrics that make some simplifying assumptions. Such assumptions are needed in order to have simple abstractions for reasoning about algorithms. However, such assumptions often lead to models, metrics, and analyses that fail to capture important aspects of actual system behavior. Using realistic system models and metrics is important, since distributed algorithms and systems are often designed to optimize over such metrics.

One example is time complexity metrics. The typical theoretical metric used to analyze the running time of distributed algorithms is the number of *communication rounds* the algorithm performs, or the number of message exchange steps in case of a non-synchronous system (e.g., [8.20, 8.14, 8.15]). In Section 8.2, we illustrate the weakness of this metric.

Another example is reliability metrics. In [8.13], we highlight the fact that fault tolerant algorithms are often designed under the assumption that no more than t out of n processes or components can fail. This characterization of failures implicitly assumes that the probability of a component failing while a protocol is in progress is independent of the duration of the protocol; that all components that can fail have an identical probability of failure; and that failure probabilities of different components are mutually independent. These assumptions do not adequately reflect the nature of real-world network environments. In practice, the likelihood of t failures occurring while a protocol is running is highly dependent on the protocol's duration. Thus, while consensus protocols that execute more rounds can tolerate more faults, the occurrence of more faults with such protocols is also more likely, which can lead to reduced system availability or reliability, as observed, e.g., in [8.3, 8.10].

The rest of this white paper is organized as follows: Section 8.2 presents an example of a new research effort that tries to better understand the performance of distributed algorithms over Internet. In Section 8.3, we outline directions for future work. Section 8.4 concludes the discussion.

8.2 Example: Evaluating the Running Time of a Communication Round over the Internet

It is challenging to predict the end-to-end performance a distributed algorithm would achieve when run over TCP/IP in a wide-area network. It is also not obvious to determine which algorithm would work best in a given setting. E.g., would a decentralized

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 40–44, 2003.

algorithm outperform a leader-based one? Answering such questions is difficult for a number of reasons. Firstly, performance prediction is difficult because end-to-end Internet performance itself is extremely hard to analyze, predict, and simulate [8.7]. Secondly, end-to-end performance observed on the Internet exhibits great diversity [8.18, 8.22], and thus different algorithms can prove more effective for different topologies, and also for different time periods on the same topology. Finally, different performance metrics can be considered.

In [8.4], we look at the running time of a communication round over the Internet. We consider a fixed set of hosts engaged in a distributed algorithm. A *communication round* is essentially a black box that propagates information from potentially every host to every other host. Every round is initiated at some host, called the *initiator*. We consider the following four common implementations of a communication round:

- *all-to-all*, where the initiator sends a message to all other hosts, and each host that learns that the algorithm has been initiated sends messages to all the other hosts. This algorithm is structured like decentralized two-phase commit, some group membership algorithms (e.g., [8.15]), and the first phases in decentralized three-phase commit algorithms, (e.g., [8.21, 8.9]).
- *leader*, where the initiator acts as the leader. In this algorithm, the initiator sends a message to all hosts, and all other hosts respond by sending messages to the leader. The leader *aggregates* the information from all the hosts, and sends a message summarizing all the inputs to all the hosts. This algorithm is structured like two-phase commit [8.8], and like the first two of three communication phases in three-phase commit algorithms, e.g., [8.21, 8.12].
- *secondary leader*, where a designated host (different from the initiator) acts as the leader. The initiator sends a message to the leader, which then initiates the leader-based algorithm.
- *logical ring*, where messages propagate along the edges of a logical ring. This algorithm structure occurs in several group communication systems, e.g., [8.1].

Using the typical theoretical metric that counts message exchange steps, we get the following running times: 2 communication steps for the all-to-all algorithm; 3 for the leader algorithm; 4 for secondary leader; and $2n - 1$ steps for the ring algorithm in a system with n hosts.

In [8.4] we evaluate these four algorithms over the Internet. Our experiments span ten hosts, at geographically disperse locations – in Korea, Taiwan, the Netherlands, and several hosts across the US, some at academic institutions and others on commercial ISP networks. The hosts communicate using TCP/IP. We measure each algorithm's *overall running time*, that is, the time that elapses from when initiator initiates the algorithm, and until *all* the hosts terminate. In contrast to what the communication step metric suggests, we observe that all-to-all usually has the worst performance. In cases in which the initiator is a host with good communication links to other hosts, the leader algorithm performs best. If the initiator is a host, like the one in Taiwan, that has poor connectivity to most of the other hosts, then secondary leader algorithm achieves the best overall running time. The typical running time of ring was usually less than double the running times of the other algorithms. As an aside, we note that in case of failures, the all-to-all algorithm is the most robust of the four. The other algorithms may fail to complete

in cases of failures that occur while the algorithm is running. Thus, there is a tradeoff between performance and robustness.

Why does the standard metric fail to capture the actual algorithm behavior over the Internet? Firstly, not all communication steps have the same cost, e.g., a message from MIT to Cornell can arrive within 20 ms., while a message from MIT to Taiwan may take 125 ms. Secondly, the latency on TCP links depends not only on the underlying message latency, but also on the loss rate. If a message sent over a TCP link is lost, the message is retransmitted after a timeout which is larger than the average round-trip time on the link. Therefore, if one message sent by an algorithm is lost, the algorithm's overall running time can be more than doubled. Since algorithms that exchange less messages are less susceptible to message loss, they are more likely to perform well when loss rates are high. This explains why the overall running time of all-to-all is miserable in the presence of lossy links. Additionally, message latencies and loss rates on different communication paths on the Internet often do not preserve the triangle inequality [8.19, 8.15, 8.2], because routing policies at Internet routers often do not choose the best possible path between two sites. This explains why secondary leader can achieve better performance by refraining from sending messages on very lossy or slow paths.

One general lesson from our study is that some communication steps are more costly than others. E.g., it is evident that propagating information from only *one* host to all other hosts is faster than propagating information from *every* host to each of the other hosts.

8.3 Future Directions and Research Goals

Our goal in the *Dalgeval (distributed algorithm evaluation)* project is to develop realistic ways to evaluate distributed algorithms. We believe that in order to succeed in this endeavor, a range of research techniques must be used: from gathering of data [8.4], through empirical evaluation in real environments [8.4, 8.15] and simulation using accurate models [8.10], to theoretical modeling and analysis. These techniques complement each other, and when used together can lead to more effective results. Most importantly, obtaining data on how real environments behave can lead to more accurate simulations and more realistic theoretical system models. We propose the following general research directions:

Obtaining data about how distributed algorithms behave in realistic environments. This research effort focuses on obtaining data, and then analyzing the data to identify the factors that affect distributed algorithms' performance and availability, and how these factors come into play. Such experiments can teach us which aspects of system behavior are important and ought to be captured in a theoretical system model or metric, and which aspects have little impact and therefore can be simplified out. We gave one example of such a research effort in the Section 8.2; many others are yet to be explored.

Using the gathered data to evaluate a range of algorithms. The gathered data can be used, for example, in trace-driven simulations. Consider the results of the experiments described in the previous section [8.4]. Beyond the specific evaluation of four different distributed algorithms for propagating information, the data gathered in those experiments provides information regarding the nature of communication failures over the

Internet, and the correlation among failures over distinct communication paths. This information is useful for evaluating many different kinds of algorithms. Indeed, trace data we gathered in those experiments is currently being used by other researchers [8.11] for evaluating the stability of group membership algorithms such as Moshe [8.15] over the Internet, and the effectiveness of different scalable master-worker algorithms that use group membership, e.g., [8.6, 8.17]. It is our hope that in the future, these traces and others will be used to evaluate various other algorithms.

Formulating better theoretical complexity and reliability metrics. Ultimately, we hope that such empirical results will lead to more realistic theoretical evaluation of distributed algorithms. However, the transition from data to models is not easy; having gathered data about real systems, it is still challenging to find ways to model this data so it will be easy to reason about.

We now propose one simple example of how our empirical results can be used to improve the accuracy of theoretical complexity analysis. As noted above, our results show that propagating information from only one host to all other hosts is faster than propagating information from every host to each of the other hosts. This observation can be leveraged in order to refine the way by which one analyzes algorithm time complexity. We suggest to refine the communication step metric as to encompass different kinds of steps. One cost parameter, Δ_1, can be associated with the overall running time of a step that propagates information from all hosts to all hosts. This step can be implemented using the most appropriate algorithm for the particular setting where the algorithm is deployed. A different (assumed smaller) cost parameter, Δ_2, can be associated with a step that propagates information from one host to all other hosts. Another cost parameter, Δ_3 can be associated with propagating information from a quorum of the hosts to all the hosts[1], etc. This more refined metric can then be used to revisit known lower and upper bound results. E.g., [8.14] presents a tight lower bound of two communication steps for failure-free executions of consensus in practical models. Under the more refined metric, the lower bound is $2\Delta_1$, whereas known algorithms (e.g., [8.16, 8.5]) achieve running times of $\Delta_2 + \Delta_3$.

Improving algorithm design. Finally, we hope that focusing on the "right" metrics will lead to the design of more effective distributed algorithms and systems.

8.4 Conclusions

Gathering data about network characteristics and the behavior of distributed algorithms in different networks is extremely important. Such data can be at the basis of more realistic simulations, as well as theoretical complexity and reliability metrics. Ultimately, using better performance metrics can lead to more effective design of distributed algorithms and systems.

[1] In future experiments we intend to evaluate a primitive that waits for responses from a quorum of hosts.

References

8.1 D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Trans. Comput. Syst.*, 16(2):93–132, May 1998.

8.2 D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pp. 131–145. ACM, Oct. 2001.

8.3 Ö. Babaoğlu. On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Trans. Comput. Syst.*, 5(4):394–416, 1987.

8.4 O. Bakr and I. Keidar. Evaluating the running time of a communication round over the Internet. In *ACM Symp. on Prin. of Dist. Comp. (PODC)*, July 2002.

8.5 T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

8.6 S. Dolev, R. Segala, and A. Shvartsman. Dynamic load balancing with group communication. In *Intl. Coll. Struct. Inf. and Comm. Complexity*, 1999.

8.7 S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Trans. Networking*, 9(4):392–403, Aug 2001.

8.8 J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, LNCS 60*, pp. 393–481, 1978.

8.9 R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *IEEE Intl. Symp. on Par. and Dist. Proc. (SPDP)*, Oct 1995.

8.10 K. W. Ingols and I. Keidar. Availability study of dynamic voting algorithms. In *21st Intl. Conf. on Dist. Comp. Sys. (ICDCS)*, pp. 247–254, Apr 2001.

8.11 K. Jacobsen, K. Marzullo, and X. Zhang. Private communication, 2002.

8.12 I. Keidar and D. Dolev. Increasing the resilience of distributed and replicated database systems. *J. Comput. Syst. Sci.*, 57(3):309–324, Dec 1998.

8.13 I. Keidar and K. Marzullo. The need for realistic failure models in protocol design. In *4th Intl. Survivability Wshop (ISW) 2001/2002*, March 2002.

8.14 I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. Tech. Rep. MIT-LCS-TR-821, MIT May 2001.

8.15 I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. *ACM Trans. Comput. Syst.*, 20(3):1–48, August 2002.

8.16 L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

8.17 G. Malewicz, A. Russell, and A. Shvartsman. Optimal scheduling for disconnected cooperation. In *Intl. Coll. Struct. Inf. and Comm. Complexity*, Jun 2001.

8.18 V. Paxson. End-to-end Internet packet dynamics. In *ACM SIGCOMM*, Sep 1997.

8.19 S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, pp. 289–299, Sep 1999.

8.20 A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Dist. Comp.*, 10(3):149–157, 1997.

8.21 D. Skeen. Nonblocking commit protocols. In *ACM SIGMOD Intl. Symp. on Management of Data*, pp. 133–142, 1981.

8.22 Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of Internet path properties. In *ACM SIGCOMM Internet Measurement Wshop*, Nov 2001.

9. Towards Robust Optimistic Approaches

Ricardo Jiménez-Peris and Marta Patiño-Martínez

Technical University of Madrid (UPM), Madrid, Spain
{rjimenez,mpatino}@fi.upm.es

9.1 Introduction

Optimism is a well-known technique to enhance the performance of distributed protocols. Optimistic approaches exploit properties exhibited by the system with certain likelihood, (i.e., that certain kinds of scenarios will prevail over others) to outperform the corresponding conservative protocol. These properties are usually referred as optimistic assumptions (e.g., an optimistic assumption is that reliably multicast messages in a LAN are spontaneously totally ordered). When the optimistic assumption holds, the optimistic approach is more efficient than the conservative one. However, this gain usually implies a tradeoff. That is, if the optimistic assumption does not hold, the optimistic approach is less efficient than the conservative one. This is due to the need to undo or repair the incorrect actions and the dismissal of work already done. This is precisely the Achilles' heel of traditional optimistic approaches. Therefore, what is crucial for an optimistic approach to be successful is that the resulting gains of optimism outweigh the penalties imposed by optimism failures.

Researchers have long recognized the potential benefits of using optimism and have proposed optimistic versions of conventional distributed protocols [9.2, 9.19]. However, despite the many optimistic approaches suggested in distributed computing, they are not that common in industrial applications. The main reason for this reluctance is that whenever the optimistic assumption fails, the protocol behaves worse that the conventional protocol. This behavior might imply more messages, or undoing part of the work. It is our opinion that to increase the applicability of optimistic protocols they need to be enriched with safeguards that limit the consequences of those scenarios where the optimistic assumptions do not hold. These safeguards make optimistic approaches more robust and efficient, and therefore, more applicable. As a consequence, in those periods during which the optimistic assumptions do not hold frequently enough, the system will not degrade to unacceptable levels.

In this paper, we try to point out the possible causes of the lack of success of some optimistic protocols, and show which directions can be taken to overcome these shortcomings in order to diminish the existing reluctance in industry for this kind of protocols. We think that optimistic protocols will play a crucial role in the upcoming wide-area distributed systems. Despite that bandwidth will grow more and more, latency will always be a problem in WANs due to physical limitations.

9.2 Traditional Optimistic Approaches

Replication. Computer clusters are the hardware platform of choice for many types of distributed information systems, and more concretely for replicated databases. These

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 45–50, 2003.

systems usually have strong availability and consistency requirements. Reliable multicast [9.3] has become one of the main abstractions for building fault-tolerant distributed systems. More particularly, it has become a key building block for modern information systems. Unfortunately, and in spite of the intensive research carried out in the area, there is still a lot of reluctance in the information systems community to use such protocols on account of their performance (mainly the high latency). This reluctance has prevented its widespread use in some contexts such as replicated databases. In this paper, we will focus on optimistic protocols for distributed databases [9.2, 9.17], [9.9, 9.4], with special emphasis on those based on reliable multicast [9.16], [9.20, 9.1, 9.14, 9.10].

Typical measures of efficiency in an information system are throughput and response time. The latency introduced by reliable multicast, especially when total order and/or uniformity are provided, can have a severe effect on these efficiency measurements. Thus, in practice, designers opt for protocols that provide weaker guarantees but have a lower latency, especially in WANs.

The ideal would be a protocol exhibiting the strong guarantees provided by reliable multicast (i.e. providing total order and/or uniformity) but without the latency penalty typically associated with the implementation of these guarantees. One way to achieve this goal is by using an optimistic approach. One of such optimistic approaches is taken in [9.16], where the spontaneous total order exhibited by IP multicast in a LAN is exploited to reduce the latency of transaction processing in a replicated database. In [9.16] transactions are multicast to all sites. Multicast messages are totally ordered to ensure one copy serializability, the correctness criteria for replicated databases [9.2]. In this approach, multicast messages are delivered optimistically as soon as they are received [9.23, 9.22]. Thus, the database can start their processing in an optimistic fashion. When the total order of the multicast message is established, the message is definitively delivered to the database system. The time elapsed between the optimistic delivery and the definitive delivery is used to process (at least, partially) the message (transaction).

The caveat of such an optimistic approach is that it can result in a high number of transaction aborts (rollbacks) when the optimistic assumption does not hold. More concretely, when the load is high and messages are retransmitted due to buffer overruns, the order in which the messages arrive at different sites differs. Therefore, conflicting transactions can be executed in different orders at different sites. The corrective action needed when the optimism fails consists in aborting those transactions that have been executed optimistically following an order that do not comply with the total order and reexecute them.

Atomic Commitment. Distributed information systems use atomic commit protocols to ensure the atomicity of their operations. Well-known examples of atomic commit protocols are the two phase commit protocol (2PC) [9.7] or three phase commit protocols (3PC) [9.21, 9.12]. One of the intrinsic limitations of commit protocols is the incurred latency, involving several rounds of messages and forced disk writes. Some optimistic approaches have been proposed to overcome these limitations, such as the optimistic two phase commit protocol [9.17] or the OPT two-phase commit protocol [9.9].

The optimistic two-phase commit protocol [9.17] takes as optimistic assumption that the most likely outcome of the commit protocol is to commit the transaction. The

protocol exploits this optimistic assumption by releasing the locks of the committing transaction at each participant (site) when the participant votes "yes" (that is, once the participant is prepared). In this way, conflicting transactions are able to obtain their locks without waiting for the commit to complete. The tradeoff of the protocol is that when the transaction outcome is abort, the transactions that obtained the locks optimistically should be aborted in order to guarantee full atomicity. This is a serious drawback since it could lead to cascading aborts. For that reason, [9.17] resorts to the application semantics and uses compensating transactions to undo semantically the effects of the transaction that optimistically released the locks. The transactions that have acquired the locks do not abort. In this way, if the optimistic assumption does not hold, there are no cascading aborts. However, unlike the approaches that will be described in the next section this safeguard is introduced by relaxing the problem to be solved. More concretely, this atomic commitment protocol only provides semantic atomicity, instead of guaranteeing full atomicity, since it sacrifices the isolation property of transactions.

Concurrency Control. Database replication protocols are classified as eager or lazy depending on whether they propagate the updates of transactions as part of the original transaction or in a different one [9.8]. The advantage of eager protocols is that consistency is kept among replicas. However, transaction latency increases and scalability diminishes as more replicas are added to the system [9.8]. Traditional eager replication approaches synchronize the acquisition of each lock at all sites, that is, a transaction gets a lock on a data item when the lock is granted at all sites, what undermines scalability. This locking protocol is used to ensure one-copy serializability. Recent advances in this area have shown that is possible to scale up by using an optimistic concurrency control method that serializes transactions according to the total delivery order of multicast [9.14, 9.20, 9.15]. These approaches extend former traditional optimistic concurrency control protocols [9.2] with the use of total ordered multicast. In these new optimistic protocols a transaction is executed optimistically at a single site. The optimistic assumption is that transactions executed optimistically at different sites are unlikely to conflict. As part of the protocol, after executing the transaction optimistically, a site propagates the transaction updates to the rest of the sites by means of totally ordered multicast. This information exchange is used to verify whether that transaction conflicts with the transactions executed concurrently at the rest of the sites, and the total order is used as the serializing order. If there is a conflict, the optimistic assumption does not hold, and conflicting transactions must be aborted.

9.3 Future Direction: Robust Optimistic Approaches

In the previous section we have seen that optimism has been introduced in different distributed protocols in order to enhance the performance of the corresponding conservative protocol. In this section, we propose a new direction to overcome the problems of traditional optimistic approaches. We describe some early steps taken in this direction related to the successful application of robust optimistic distributed protocols. These protocols are enriched with safeguards that prevent a trashing behavior when optimistic assumptions do not hold frequently enough. In the following, a partially synchronous

system with crash failures is assumed. More details about the underlying model are found in the given references.

Replication. [9.18] presents an eager data replication protocol based on total order multicast enriched with safeguards to improve its robustness. This protocol uses the same optimistic approach for multicast messages as the one described in [9.16]. That is, the calculation of the total order is overlapped with the optimistic execution of transactions, taking advantage of the spontaneous total order exhibited by LANs. The novelty in this approach lies in its robustness. The protocol is enhanced with a reordering algorithm. The reordering acts as a safeguard that prevents the abortion of transactions executed optimistically when the optimistic assumptions do not hold (i.e., when the spontaneous order does not match the total order). If the spontaneous and total orders differ at the site where the transaction is executed a reordering technique is used to prevent the transaction abortion. That site informs the rest of the sites about the new order when it propagates the transaction updates (without any extra messages). The reordering can be done as long as the updates of the transaction optimistically executed are a subset of the updates of the preceding transactions in the total order. A transaction aborts only if both, this property and the optimistic assumption about spontaneous total order do not hold. This reordering has the additional advantage that it does not incur in any significant overhead. The protocol has been used successfully in a replication middleware [9.11]. Experimental results showed that in most cases there were no aborts and that under worst case scenarios, aborts never exceeded 0.2%, therefore limiting successfully the cost of scenarios where the optimistic assumption did not hold.

Atomic Commitment. Another recent advance in this direction consists in an optimistic non-blocking atomic commitment protocol [9.10]. Non-blocking atomic commitment is known to have an inherent cost of two rounds of messages in a synchronous system [9.5] and, although not proven, it seems that there is a lower bound of three rounds in a partially synchronous system (at least, only protocols with three rounds have been proposed). The extra round that seems to be needed in a partially synchronous system with respect to a synchronous system is due to the possibility of false suspicions. As suggested by [9.13], it is possible to circumvent some lower bounds of agreement problems, such as non-blocking atomic commitment, by using an optimistic approach. One way to model the non-blocking atomic commitment problem is by using uniform multicast [9.6]. The protocol proposed in [9.10] hides the latency introduced by the uniformity by overlapping the stabilization of uniform multicast messages with an optimistic execution of the atomic commitment protocol. Once all the votes have been received optimistically and are yes, the transaction locks are released and they can be granted optimistically to other transactions. In this way, conflicting transactions do not pay for the full cost of non-blocking atomic commitment, three rounds. Instead they are allowed to progress after the second round of messages. In this protocol the safeguard consists in limiting the optimistic execution of conflicting transactions to one level, therefore preventing cascading aborts in the unlikely case the optimism fails (this safeguard was first proposed in the OPT 2PC protocol [9.9]). The optimistic assumption fails when the message (uniform multicast) with the last vote arrives at a single receiver optimistically and then the sender and the receiver fail before the message reaches any other receiver, and additionally, the receiver committed optimistically the transaction before failing. It

should be noted that garbage collection in this protocol can be performed in the same way is done in traditional commit protocols, without requiring to keep a trace of the whole history.

Some recent work is focused on extending optimistic delivery multicast protocols to WANs. [9.23] proposes a uniform total ordered multicast for WANs in which optimistic delivery takes place after the total order is established and before the message is stable. A different approach has been taken in [9.22] a spontaneous total order in WANs is tentatively attempted by enforcing a homogeneous delivery time at all sites.

We think that the robust optimistic approaches followed in some of the presented protocols might be extended to other contexts and might help to advance the state of the art in this area and at the same time foster their industrial use. It is our opinion that optimistic protocols will become essential for developing the upcoming wide-area distributed systems to deal with the inherent latency of wide-area networks.

References

9.1 Y. Amir and C. Tutu. From Total Order to Database Replication. In *ICDCS*, 2002.

9.2 P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

9.3 K. Birman. *Building Secure and Reliable Network Applications*. Prentice, 1996.

9.4 Y. Breitbart, H. F. Korth, and A. Silberschatz. Optimistic protocols for replicated databases. Technical Report BL0112370-970227-07, Bell Labs, 1997.

9.5 B. Charron-Bost and F. LeFessant. Validity Conditions in Agreement Problems and Time Complexity. Technical Report 4526, INRIA, 2002.

9.6 G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):427–469, December 2001.

9.7 J. Gray. *Notes on Database Operating Systems*. Springer, 1978.

9.8 J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of the SIGMOD*, pages 173–182, Montreal, 1996.

9.9 R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *Proc. of the ACM SIGMOD*, 1997.

9.10 R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arevalo. A Low-Latency Non-Blocking Atomic Commitment. In *Proc. of DISC. LNCS-2180*. Springer, 2001.

9.11 R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *Proc. of 22nd IEEE ICDCS*, Vienna, Austria, July 2002.

9.12 I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proc. of ACM Principles of Database Systems*, 1995.

9.13 I. Keidar and S. Rajsbaum. On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, MIT CS Lab, 2001.

9.14 B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of VLDB*, 2000.

9.15 B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, 2000.

9.16 B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of ICDCS*, 1999.

9.17 E. Levy, H. F. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *ACM SIGMOD Conf.*, 1991.

9.18 M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of DISC. LNCS-1914*. Springer, 2000.

9.19 F. Pedone. Boosting System Performance with Optimistic Distributed Protocols. *IEEE Computer*, pages 80–86, 2001.
9.20 F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Journal of Distributed and Parallel Databases and Technology*. to appear.
9.21 D. Skeen. A Quorum-Based Commit Protocol. In *Proc. of the Works. on Distributed Data Management and Computer Networks*, pages 69–80, 1982.
9.22 A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *Proc. of SRDS*, 2002.
9.23 P. Vicente and L. Rodrigues. An Indulgent Uniform Total Order Algorithm with Optimistic Delivery. In *Proc. of SRDS*, 2002.

10. Towards a Practical Approach to Confidential Byzantine Fault Tolerance

Jian Yin, Jean-Philippe Martin, Arun Venkataramani,
Lorenzo Alvisi, and Mike Dahlin

The University of Texas at Austin

10.1 Introduction

As the world becomes increasingly interconnected, more and more important services such as business transactions are deployed as *access-anywhere services* – services that are accessible by remote devices through the Internet and mobile networks. Such services often must access confidential data to provide service. For example, an online bank service must access a user's checking account to process an online transfer request. In such a scenario, guarantees of availability, integrity, and confidentiality are essential. By availability, we mean that services must provide service 24/7 without interruption. By integrity, we mean that services must process clients' requests correctly. By confidentiality, we mean that services must restrict who sees what data. Given today's economics and technology that make it infeasible to rigorously test and verify complex components, it is more attractive to allow untrustworthy components to be assembled into a trustworthy system. A traditional Byzantine fault-tolerant (BFT) system runs different implementations of the same service on several replicas and ensures that correct computation is performed by enough correct replicas to mask incorrect replicas [10.1, 10.7, 10.8]. Recent research has shown that BFT systems can be practical for several important services as they can be implemented with low overhead compared to the unreplicated services [10.3].

Although traditional BFT systems improve availability and integrity through redundancy, existing BFT architectures make such systems more prone to compromising confidentiality. In a traditional BFT system, replicas send all replies directly to clients. Thus, if a hacker manages to compromise one of the replicas, he can steal confidential data. Moreover, in traditional BFT systems, there is a fundamental tradeoff between increasing availability and integrity on one hand and strengthening confidentiality on the other. BFT systems use several replicas to provide availability, based on the reasoning that the replicas are different and it is therefore unlikely that they all fail simultaneously. However, because it is sufficient for an attacker to compromise a single replica, this approach also increases the chance that at least one replica contains an exploitable bug, allowing the attacker to gain access to the confidential data that the service uses.

This paper discusses how to use redundancy to simultaneously improve availability, integrity, and confidentiality. We call this problem Confidential BFT (CBFT) and propose the Privacy Firewall architecture to solve it. Service replicas connect to the Privacy

This work was supported by DARPA/SPAWAR grant N66001-98-8911, ATP, Tivoli, two NSF CAREER awards (CCR-9734185 and CCR-9733842), and two Alfred P. Sloan Fellowships.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 51–56, 2003.

Firewall and can send messages to the outside world only through it. The firewall runs a majority voting algorithm just like the clients of a traditional BFT system and filters out faulty messages that may contain confidential data. This approach can improve the security of any replicated service. Moreover, the Privacy Firewall can be adapted for different services with little modification, thus amortizing the development cost.

The firewall system has to be correct to provide confidentiality. Even though the firewall is simple, building a formally verified bug-free firewall may not be feasible. However, redundancy can be used to improve the robustness of the firewall. Such a firewall system consists of a group of nodes that are interconnected such that any path from a service replica to the outside world is longer than a threshold f. Thus, as long as there are f or fewer faulty firewall nodes, any communication from any service replica to the outside world must go through at least one correct node. Moreover, a correct node in a firewall chain can independently ensure that a unique sequence of replies results from a sequence of requests just as if this sequence of requests were processed by a single correct server. Thus, faulty machines are prevented from using steganography to leak confidential data.

In summary, this note introduces the problem of confidential Byzantine-fault tolerance and presents a system, the Privacy Firewall, to show that CBFT can be solved. The key challenges ahead are to evaluate this new design and to develop a deeper understanding of the range of solutions to the CBFT problem.

10.2 State of the Art

Previous work in Byzantine fault tolerance falls into two categories: i) using client voting to improve availability and integrity without considering confidentiality and ii) using secret sharing to improve confidentiality for a limited class of services where servers perform few or no operations on the data they store.

Redundancy techniques to improve availability and integrity without considering confidentiality have been extensively studied. The common characteristic of such systems is that all service replicas send replies to clients and voting takes place at clients. There is a significant body of work on BFT quorum systems [10.7] in which a group of service replicas use intersection properties of quorum sets to guarantee that clients retrieve values of variables written previously. Byzantine fault tolerance for arbitrary services captured by state machines have been studied in both theoretical and practical settings [10.1, 10.3, 10.8].

When considering confidentiality, the most direct approach is to have the clients encrypt all data before sending it to the servers for storage. If the servers do not have the key then the data is stored securely, even if some servers are compromised [10.5]. Although this approach provides fault-tolerant confidentiality, the only two operations that the servers can implement are "store" and "retrieve". Threshold cryptography [10.4, 10.9] allows a threshold number of servers to generate signatures cooperatively. Less than the threshold number of faulty nodes can neither generate a valid signature nor reconstruct the shared private key.

Byzantine fault-tolerant confidentiality can theoretically be implemented using secure multi-party computation [10.2]. Secure multi-party computation guarantees that no

group of less than a threshold number of nodes holds enough information to reconstruct the confidential data.

However, in practice these schemes can only be used for simple functions such as small-scale voting and bidding. More complex functions are computationally prohibitive since the schemes make heavy use of oblivious transfers, which in turn employ operations no cheaper than exponentiation.

We aim to build a practical confidential Byzantine fault tolerant system. Unlike multiparty secure computation, we allow server replicas to access and process confidential data. Thus, a faulty server replica can contain confidential data. However, our server replica nodes and firewall nodes are interconnected in such a way that no confidential data can be sent out of our system unless a threshold number of server replica nodes and firewall nodes are comprised. Our system can be implemented efficiently since it uses much fewer exponent operations than secure multiparty computation.

10.3 Future Directions: Confidential Byzantine Fault Tolerance

The traditional state machine approach for implementing Byzantine fault-tolerant services delegates the responsibility of combining the outputs of the ensemble of state machine replicas to the client. To quote Schneider [10.8], *"the voter - a part of the client - is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant"* because a faulty voter is then synonymous with a faulty client. Although such a model is appropriate for services where availability and reliability are the goals, it fails to address confidentiality for access-anywhere services. In particular, if a hacker manages to compromise one replica in such a system, the system has no mechanisms to prevent the compromised replica from sending confidential data back to the hacker.

As shown in Figure 10.1(**b**), traditional approaches to tolerate Byzantine faults in replicas attempt to emulate the abstract notion of a single correct server as in Figure 10.1(**a**). Here R is a request sent by the client, P the response the client receives, and $P^{'}$ the set of responses filtered by the voting component V to eliminate incorrect information I received by V from a faulty replicas in the BFT. I can be used by a malicious replica as a channel to transmit confidential data. The obvious solution is to move the voter, V, away from clients into the service provider's domain as shown in Figure 10.1(**c**). Unfortunately, doing so is difficult since V is a potential point of vulnerability for availability, integrity, and confidentiality. In particular, Schneider's fault-sharing argument above no longer applies – faults in the voter V can hurt availability and integrity of correct clients. Furthermore, a fault in the voter can reintroduce a covert channel that allows the information I to escape to clients.

One solution is to use formal methods to construct a perfect voter that never fails. Another approach is to use redundancy to construct a highly reliable voter system out of imperfect components. The whole system can survive Byzantine failures in some of its components and protect confidentiality without compromising availability. This is a hard problem, especially in the face of sophisticated attacks that can delay, omit or generate messages.

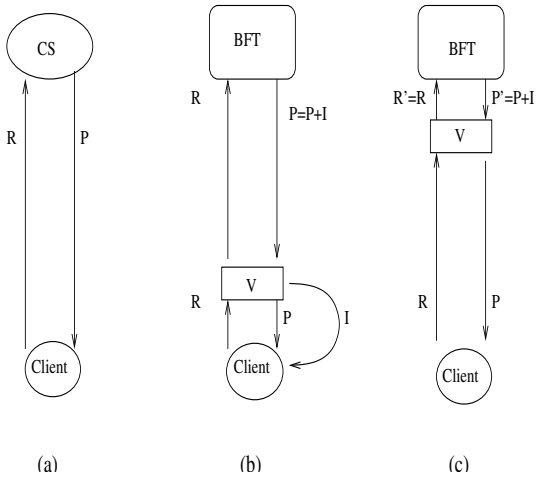

Fig. 10.1. Fault-tolerance models

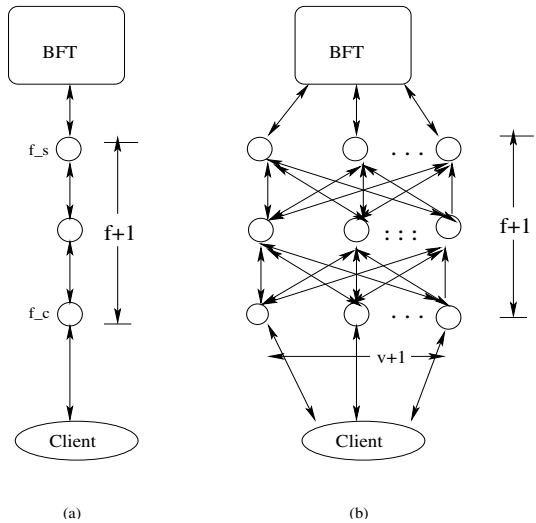

Fig. 10.2. The Privacy Firewall

To provide intuition, we first describe a simplified firewall that filters some incorrect information I, but is vulnerable to poor availability and in-band signaling that can leak confidential data. We then outline how to address these limitations. The system consists of two components as illustrated in Figure 10.2(**a**):

a) A backend BFT consisting of replicated servers U to tolerate s Byzantine faults.

b) A firewall network V, organized as a chain of $f + 1$ machines, at most f of which are faulty.

Voters can communicate directly only with their neighbors. Only f_c at the bottom of the chain can communicate directly with clients, and only the topmost machine f_s can communicate with the server replicas.

Clients send encrypted, signed requests to f_c. Each machine in the chain forwards the request up the chain towards f_s which broadcasts it to all servers in U. Servers in U are expected to respond with encrypted, signed replies to f_s. On receiving $s + 1$ identical replies with verifiable correct signatures for a given request, f_s forwards the reply and the set of $s+1$ signatures to the machine just below it in the chain. Each correct firewall thereafter independently verifies the signatures and forwards the message, until the client. If any firewall other than f_s detects that the replies do not match, it discards the reply and triggers a fault-detection alarm.

This design guarantees that the client only receives correct replies, because every response is verified by $f + 1$ machines in the firewall (at least one of which is correct) before reaching the client. In other words, replies will get through the firewall if and only if they would have been returned by the service when no failures occur. This design guarantees that data considered confidential by the service will remain confidential in the presence of failures.

The system described above, though simple, has the following three drawbacks: (1) Availability: if one of the machines in the firewall is compromised, it could drop requests or replies. (2) A faulty firewall machine might leak confidential information by altering the membership of the set of correct signatures vouching for a reply. (3) A faulty firewall machine might leak confidential information by inserting, omitting, or reordering requests or replies.

Availability of the system can be improved simply by augmenting the firewall with multiple interlinked chains each $f + 1$ in length as shown in Figure 10.2(**b**). This topology guarantees that faulty servers cannot transmit information by opting not to answer queries. If v machines are susceptible to crash failures, it is a straightforward graph-theoretical problem to prove that a minimum of $(f + 1)(v + 1)$ machines are necessary to protect confidentiality and ensure availability in the face of f Byzantine or v crash failures. The second problem can be rectified by using threshold signatures [10.6] with a threshold of $s + 1$. Each of the top firewalls combines the partial signatures, generating a service signature that can be independently verified by each machine along each chain. One solution to (3) is for each firewall machine to ensure that it transmits replies in exactly the order that it receives requests, thus leaving no degree of freedom for erasure or reordering of replies.

With the above additions, it can be shown that the Privacy Firewall emulates the single correct server abstraction depicted in Figure 10.1. We show the equivalence by considering the sequence of requests at the first correct firewall machine and showing that our system produces the same sequence of replies as the correct single server. By removing all flexibility in how information is sent out, we prevent faulty machines from hiding confidential data in the information flow. It is straightforward to extend the argument to multiple chains.

10.4 Conclusion

In this abstract we argue that availability, integrity, and confidentiality are essential for access-anywhere services and that traditional replicated Byzantine fault tolerant systems, although they strengthen integrity and availability, weaken confidentiality. A Confidential BFT system (CBFT) differs from traditional BFT systems in that it increases confidentiality guarantees instead of reducing them, while maintaining the availability and integrity guarantees provided through redundancy.

We claim that it is possible to implement a CBFT service which can perform arbitrary operations on the data it stores and yet leak no information if some of the servers are compromised. The Privacy Firewall shows how such a system may work. We are currently implementing a prototype and evaluating its performance.

This example, we hope, will motivate others to find more efficient solutions to the CBFT problem and to add confidentiality guarantees to other protocols.

References

10.1 G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. In *Journal of the Association for Computing Machinery*, October 1985.

10.2 Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. Ph.d. thesis, Weizmann Institiute of Science, Israel, 1995.

10.3 M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

10.4 Y. G. Desmedt. Threshold cryptography. *European Trans. on Telecommunications, 5(4)*, pages 449–457, July-August 1994.

10.5 Juan Garay, Rosario Gennaro, Charanjit Jutla, and Tal Rabin. Secure distributed storage and retrieval. In *Theoretical Computer Science TCS 243(1-2)*, pages 363–389, 2000.

10.6 R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology*, 13(2):273–300, Spring 2000.

10.7 D. Malkhi and M. Reiter. Byzantine quorum systems. In *Distributed Computing*, 1998.

10.8 F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

10.9 L. Zhou, F. Schneider, and R. Renesse. COCA: A secure distributed on-line certification authority. In *Technical Report TR2000-1828, Computer Science Department, Cornell Univesity*, 2000.

11. Modeling Complexity in Secure Distributed Computing*

Christian Cachin

IBM Research, Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
cca@zurich.ibm.com

11.1 Introduction

Security considerations play an increasingly important role for distributed computing. In the future, dependable distributed systems for open networks can no longer be designed without taking malicious attacks into account. The enabling technology for security is cryptography, which has been placed on sound theoretical foundations during the last twenty years. The formal model of modern cryptography is based on computational complexity theory because of the need for modeling computational difficulty; it differs substantially from the formal models of distributed computing, which do not usually deal with bounds on time complexity or with randomization. We argue that an integration of these two approaches is necessary for reasoning about the security of cryptographic protocols in distributed systems. We discuss the notion of a *uniformly bounded protocol statistic* that allows for composing protocols with computational security; it has recently been proposed for constructing cryptographic randomized atomic broadcast protocols for asynchronous systems.

11.2 Two Formal Approaches

Distributed computing. The prevalent formal models in distributed computing today are based on finite automata and on infinite time. In the *I/O automaton* model [11.11], for instance, liveness and fairness properties of a system are expressed in terms of the system's external behavior (its trace) as observed during a potentially infinite run.

Take the classical problem of *Byzantine agreement* [11.12], where a set of n parties must reach the same decision despite the fact that up to t of them fail in arbitrary, potentially malicious ways. The standard formalization consists of three conditions:

Validity: If all non-faulty parties start with the same value v, every non-faulty party that terminates decides for v.
Agreement: No two non-faulty parties decide on different values.
Termination: All non-faulty parties eventually terminate.

* This work was supported by the European IST Project MAFTIA (IST-1999-11583). However, it represents the view of the author(s). The MAFTIA project is partially funded by the European Commission and the Swiss Department for Education and Science.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 57–61, 2003.

No violation of these properties is allowed even if it would occur with extremely small probability, no bound is placed on the time it takes to reach agreement, and the parties are allowed to perform an unbounded number of computation steps.

Because of its adversarial nature, Byzantine agreement is perhaps the best problem to illustrate the use of cryptographic techniques in distributed computing. The problem as stated is impossible to solve in asynchronous networks [11.6], a result that had considerable influence on subsequent research in distributed computing. If one augments the model with a digital signature scheme such that every party can add an unforgeable authentication tag to any message, which can be verified by all parties, the problem becomes easier to solve. The formal statement of the authenticated model requires an *ideal* digital signature scheme, where it is *impossible* for any faulty party to generate a tag that is recognized as a valid signature by a non-faulty party.

Cryptography. Since the discovery of public-key cryptography [11.5], research in theoretical cryptography has concentrated on appropriate models for cryptographic tasks, such as encryption and digital signatures. The security notions of modern cryptography, originating with [11.8], are based on asymptotic formalizations in the tradition of complexity theory (see [11.7] for an introduction). This is because the efficient cryptographic algorithms available today provide only *computational* security guarantees against adversaries whose resources are bounded.

As an example, consider a *one-way function* — one of the fundamental concepts in modern cryptography. Such a function is easy to evaluate but hard (on average) to invert. It is intuitively clear that a digital signature scheme must involve a one-way function because every party should be able to verify a signature issued by party P with an efficient algorithm, but no feasible computation by a malicious party must be able to come up with P's signature on a message that P has not signed. Unfortunately, real-world cryptographic primitives are not ideal: a simple algorithm may guess an input for the one-way function and hit one that is mapped to a given output with non-zero probability, or, for that matter, forge a signature of P with non-zero probability.

Modern cryptography takes this into account by introducing a security parameter k and formalizing the asymptotic behavior of a primitive in dependence of k. The security parameter may be thought of as indicating the key length of the primitive. The basic assumptions are that polynomial-time algorithms are efficient and that a "negligible" probability of failure cannot be ruled out. A function $\epsilon(k)$ is called *negligible* if it decreases faster than any inverse polynomial, i.e., if for all $c > 0$, there exists a constant k_0 such that $\epsilon(k) < \frac{1}{k^c}$ for $k > k_0$.

Now, a *one-way function* may be defined as a *family of functions* $f_k : \{0,1\}^k \to \{0,1\}^k$ for $k > 0$ such that (1) there exists a polynomial-time algorithm that computes $f_k(x)$ for all $x \in \{0,1\}^k$, and (2) for any probabilistic polynomial-time algorithm A, there exists a negligible function $\epsilon_A(k)$ such that $\Pr\big[f\big(A(f(x))\big) = f(x)\big] \leq \epsilon_A(k)$ where the probability is over the uniform choice of $x \in \{0,1\}^k$ and the random choices of A. Algorithms are modeled as Turing machines that take k as an (implicit) auxiliary input to enforce their dependence on k (for technical reasons k is often given in unary notation as 1^k).

It is clear that these two formal models cannot be combined in a straightforward manner, say, for analyzing randomized Byzantine agreement protocols that use cryp-

tography. At the very least, an appropriate model should allow a protocol to fail with negligible probability because a cryptographic primitive has been broken.

One may also wonder if the impossibility of asynchronous agreement [11.6] would have received so much attention, had one known at the time that it can be solved efficiently [11.4, 11.3] except with negligible probability. This probability is so small that no efficient algorithm can detect it.

11.3 Towards a Unified Model

We propose to explore a new formal system model for distributed computing that bridges this gap. It is a refinement of traditional models in distributed computing, such as the I/O automaton model, which allows for reasoning about the computational complexity of a protocol and for implementing protocols with cryptographic primitives. Because the model must allow to bound the running time of a system, most changes will affect the formal treatment of termination. We introduce the notion of *uniformly bounded statistics* for this purpose. Our model allows also for randomized protocols, whose running time cannot be bounded a priori, through the concept of *probabilistic uniformly bounded statistics*.

A preliminary version of such a model has been developed in connection with asynchronous cryptographic protocols for distributed systems with Byzantine faults [11.3, 11.2, 11.10, 11.1]; two of its components, Turing machines and uniformly bounded protocol statistics, are presented next.

Turing machines. A party executing a particular protocol is modeled by a probabilistic interactive Turing machine [11.9] that runs in polynomial time in the security parameter k. Two interactive Turing machines communicate through a pair of special communication tapes, where each party may only write on one tape and read from the other tape. Input and output actions of the protocol are also represented as messages on communication tapes.

There is a single adversary A, which is a probabilistic interactive Turing machine that runs in polynomial time in k. W.l.o.g. every party communicates only with the adversary, who therefore also implements the network. We sometimes restrict the adversary's behavior such that it implements a reliable network by saying that it *delivers all messages*. The parties are completely reactive and receive service requests (input actions) from the adversary and deliver their payload (output actions) also to the adversary. The notion of compatible protocols and their composition may be defined analogously to the I/O automaton model.

In the Byzantine agreement example, there are n parties ($n \leq k$), of which up to t are *faulty* and controlled by the adversary; for simplicity, faulty parties are absorbed into the adversary.

Uniformly bounded statistics. We say that a message written to a communication tape is *associated* to a given protocol if it was generated by a *non-faulty* party on behalf of the protocol. The *message complexity* of a protocol is defined as the number of associated messages (generated by non-faulty parties). It is a random variable that depends on

the adversary and on k. The *communication complexity* of a protocol may be defined analogously.

For a particular protocol, a *protocol statistic* X is a family of real-valued, non-negative random variables $X_A(k)$, parameterized by adversary A and security parameter k, where each $X_A(k)$ is a random variable induced by running the system with A. We restrict ourselves to *bounded protocol statistics* X such that for all A, there exists a polynomial p_A with $X_A(k) \leq p_A(k)$ for $k > 0$ (this bound may depend on A). Message complexity is an example of such a bounded protocol statistic.

We say that a bounded protocol statistic X is *uniformly bounded* if there exists a fixed polynomial $p(k)$ such that for all adversaries A, there is a negligible function ϵ_A, such that for all $k \geq 0$, $\Pr\big[X_A(k) > p(k)\big] \leq \epsilon_A(k)$. A protocol statistic X is called *probabilistically uniformly bounded* if there exists a fixed polynomial $p(k)$ and a fixed negligible function δ such that for all adversaries A, there is a negligible function ϵ_A, such that for all $l \geq 0$ and $k \geq 0$, $\Pr\big[X_A(k) > lp(k)\big] \leq \delta(l) + \epsilon_A(k)$. In other words, (probabilistically) uniformly bounded protocol statistics are *independent* of the adversary except with negligible probability. Assuming that the adversary delivers all messages, termination of a cryptographic protocol may be defined by requiring that the message complexity is (probabilistically) uniformly bounded.

Example. For illustration, we provide a definition of Byzantine agreement with computational security in the new model. For all polynomial-time adversaries, the following holds except with negligible probability:

Validity and *Agreement* as before.
Liveness: If all non-faulty parties have started and all associated messages have been delivered, then all non-faulty parties have decided.
Efficiency: The message complexity of the protocol is probabilistically uniformly bounded.

Hence, protocols are live only to the extent that the adversary chooses to deliver messages among the non-faulty parties, but they must not violate safety even if the network is unreliable.

Termination follows from the combination of *liveness* and *efficiency*. These properties ensure that the protocol generates some output and that the number of communicated messages is *independent* of the adversary, causing the protocol to terminate by ceasing to produce messages.

If X is (probabilistically) uniformly bounded by p, then for all adversaries A, we have $\mathrm{E}[X_A(k)] = O(p(k))$, with a hidden constant that is independent of A. Additionally, if Y is (probabilistically) uniformly bounded by q, then $X \cdot Y$ is (probabilistically) uniformly bounded by $p \cdot q$, and $X + Y$ is (probabilistically) uniformly bounded by $p+q$. Thus, (probabilistically) uniformly bounded statistics are closed under polynomial composition, which is their main benefit for analyzing the composition of (randomized) cryptographic protocols.

Outlook. The ingredients of a formal model that unites the approaches of distributed systems and cryptography have been sketched. Such a model opens the road for applying advanced cryptographic methods to many distributed computing problems with security demands.

Acknowledgments

This paper is based on joint work with Victor Shoup and Klaus Kursawe.

References

11.1 C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *Proc. 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

11.2 C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001.

11.3 C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000.

11.4 R. Canetti and T. Rabin, "Fast asynchronous Byzantine agreement with optimal resilience," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 42–51, 1993. Updated version available from `http://www.research.ibm.com/security/`.

11.5 W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, pp. 644–654, Nov. 1976.

11.6 M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

11.7 O. Goldreich, *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

11.8 S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, pp. 270–299, 1984.

11.9 S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, pp. 186–208, Feb. 1989.

11.10 K. Kursawe and V. Shoup, "Optimistic asynchronous atomic broadcast." Cryptology ePrint Archive, Report 2001/022, Mar. 2001. `http://eprint.iacr.org/`.

11.11 N. A. Lynch, *Distributed Algorithms*. San Francisco: Morgan Kaufmann, 1996.

11.12 M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, pp. 228–234, Apr. 1980.

12. Communication and Data Sharing for Dynamic Distributed Systems*

Nancy Lynch[1] and Alex Shvartsman[2,1]

[1] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA
[2] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 3155, Storrs, CT 06269, USA

12.1 Introduction

This research direction aims to develop and analyze algorithms to solve problems of communication and data sharing in highly dynamic distributed environments. The term *dynamic* here encompasses many types of changes, including changing network topology, processor mobility, changing sets of participating client processes, a wide range of types of processor and network failures, and timing variations. Constructing distributed applications for such environments is a difficult programming problem. In practice, considerable effort is required to make applications resilient to changes in client requirements and to evolution of the underlying computing medium. We focus our work on distributed services that provide useful guarantees and that make the construction of sophisticated distributed applications easier. The properties we study include ordering and reliability guarantees for communication and coherence guarantees for data sharing. To describe inherent limitations on what problems can be solved, and at what cost, the algorithmic results will be accompanied by lower bound and impossibility results.

One example of our approach is the new dynamic atomic shared-memory service for message-passing systems. We formally specified the service and developed algorithms implementing the service. A system implementation is under development. The service is reconfigurable in the sense that the set of owners of data can be changed dynamically and concurrently with the ongoing read and write operations. We proved the correctness of the implementation for arbitrary patterns of asynchrony and crashes, and we analyzed its performance conditioned on assumptions about timing and failures.

12.2 Recent Directions

In recent years, we worked on distributed algorithms and their analysis for a wide range of problems including maintenance of replicated data, e.g., [12.6, 12.7] and view-oriented group membership and communication, e.g., [12.5, 12.8]. We have also carried out implementation studies, e.g., [12.3, 12.13]. Several of our projects were inspired by middleware used in commercial and academic systems, most notably, by *group communication systems* [12.4]. Additional motivation comes from dynamic voting systems,

* This work is supported by the NSF Grants 0121277, 9988304 and 9984774.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 62–67, 2003.

e.g., [12.15], protocols for atomic data, e.g., [12.1, 12.10], and dealing with unbounded number of processors, e.g., [12.14].

Many of our algorithms are designed to cope with timing anomalies and some forms of processor and communication failure, and some handled explicit requests to reconfigure the system. In carrying out this work, we found it useful to formulate problems and decompose solutions in terms of precisely-defined *global services*. Most of our work, and other work on fault-tolerant distributed computing, makes it clear that algorithms for such a setting can be extremely complex. The use of abstract global services with well-defined interfaces and behavior to decompose the algorithms helps considerably in reducing this complexity. For example, consensus algorithms have been used as building blocks in other work [12.9]. Additionally, such decomposition is useful in analyzing correctness of algorithms and their performance.

12.3 New Directions: Dynamic Distributed Systems

Our new directions target communication and data sharing problems in highly dynamic distributed environments. The environments we consider will be (even) less well-behaved than the ones we considered earlier, including, for example, unknown universe of processors, explicit requests by participants to join and leave the system, and mobility. We aim for a coherent theory rather than isolated algorithmic results, and we look for common services that can be used as parts of many algorithms, and for lower bound results as well as upper bound (algorithmic) results. We are considering network environments in which the set of processors and their connectivity changes over time. Processors and links may be added and removed from a network, and while they are in the network, they may fail and recover. Different processors and communication links may operate at drastically different speeds, and these speeds may vary over time. Processors may be connected wirelessly and be mobile, moving about in space while they communicate. Application processes may also migrate around the network. On such substrates, we will consider running distributed applications involving identified groups of participants. These will include sharing of files in wide-area networks, distributed multi-player games, computer-supported cooperative work, maintaining and disseminating information about real-world, real-time endeavors with strict data-consistency requirements (such as military operations), multimedia transmission, and others.

Approach. We view the communication and data-sharing problems to be solved as high level *global services*, which span network locations. These services generally will provide performance and fault-tolerance guarantees, conditioned on assumptions about the behavior of the environment and of the underlying network substrate.

Traditionally, research on distributed computing primitives and services has emphasized specification and correctness, while research on distributed and parallel algorithms has emphasized efficiency and performance. Our approach will combine and synthesize these two concerns: It will yield algorithms that perform efficiently and degrade gracefully in dynamic distributed systems, and whose correctness, performance, and fault-tolerance guarantees are expressed by precisely-defined global services. We will include the study of trade-offs involving service guarantees and performance. For exam-

ple, achieving atomicity is expensive, and there are reasons to believe that weakening consistency slightly may reduce the cost and still provide useful semantics.

Because the setting is very complex, the algorithms will also be very complex, which means that it is necessary to decompose them into smaller, more manageable pieces. In our research, many of those smaller pieces will be viewed as lower-level, *auxiliary global services*. These services will provide lower-level communication and data-sharing capabilities, plus other capabilities such as failure detection, progress detection, consensus, group membership, leader election, reconfiguration, resource allocation, workload distribution, location determination, and routing. These services must also include conditional performance and fault-tolerance guarantees. This decomposition can be repeated any number of times, at lower levels of abstraction. The work we pursue in attaining our goals includes:

- Defining new global services to support computing in complex distributed environments, with particular emphasis on communication and data-sharing services.
- Developing and analyzing algorithms that implement these services in dynamic systems, and algorithms that use the services to implement higher-level services.
- Obtaining corresponding lower bounds and impossibility results.

This work is carried out in terms of a mathematical framework based on interacting state machines. The state machines will include features to express issues of timing, continuous behavior, and probabilistic behavior. Supporting metatheory, including general models, performance measures, and proof and performance analysis methods, will also be developed. Our theoretical work complements ongoing work on implementation and testing of distributed system services. Parts of this work are guided by examples chosen from prototype applications, including distributed file management, information collection and dissemination, computer-supported cooperative work, and distributed games. When developing specifications motivated by existing systems we also rely on information from the developers about what their services guarantee.

Intended impact. Our research will contribute to developing the theory of communication and data sharing in dynamic distributed systems. In terms of practical implications, our research has the potential to produce qualitative improvements in capabilities for constructing applications for dynamic distributed environments. New global services can be used to decompose the task of constructing complex into manageable subtasks. Integrating conditional performance and fault-tolerance guarantees into service specifications will decompose the task of analyzing the performance and fault-tolerance of complex systems, which in turn will make this analysis more tractable. Lower bound and impossibility results will tell system designers when further effort would be futile.

12.4 Reconfigurable Atomic Memory Service

We now present an example of our new work on algorithms for dynamic systems. We overview our algorithm [12.12] that can be used to implement atomic read/write shared memory in a dynamic network setting, in which participants may join or fail during the course of computation. Examples of such settings are mobile networks and peer-to-peer

networks. One use of this service might be to provide long-lived data in a dynamic and volatile setting such as a military operation.

We developed a formal specification for a reconfigurable atomic shared memory as a global service. We call this service RAMBO, which stands for "Reconfigurable Atomic Memory for Basic Objects" ("Basic" means "Read/Write"). Then we provided a dynamic distributed algorithm that implements this service. In order to achieve availability in the presence of failures, the objects are replicated. In order to maintain memory consistency in the presence of small and transient changes, the algorithm uses *configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. In order to accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the set of members and the sets of quorums are modified. Such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time.

The algorithm carries out three major activities, all concurrently: reading and writing objects, introducing new configurations, and removing ("garbage-collecting") obsolete configurations. The algorithm is composed of a *main algorithm*, which handles reading, writing, and garbage-collection, and a global reconfiguration service, *Recon*, which provides the main algorithm with a consistent sequence of configurations. Reconfiguration is loosely coupled to the main algorithm, in particular, several configurations may be known at one time, and read and write operations can use them all.

The main algorithm performs read and write operations using a two-phase strategy. The first phase gathers information from read-quorums of active configurations and the second phase propagates information to write-quorums of active configurations. This communication is carried out using background gossiping, which allows the algorithm to maintain only a small amount of protocol state information. Each phase is terminated by a *fixed point* condition that involves a quorum from each active configuration. Different read and write operations may execute concurrently: the restricted semantics of reads and writes permit the effects of this concurrency to be sorted out later. A facility is provided for *garbage-collecting* old configurations when their use is no longer necessary for maintaining consistency.

The reconfiguration service is implemented by a distributed algorithm that uses consensus to agree on the successive configurations. Any member of the latest configuration c may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of c. Consensus is, in turn, implemented using a version of the Paxos algorithm [12.11]. Although such consensus executions may be slow—in fact, in some situations, they may not even terminate—they do not delay read and write operations. Garbage-collection uses a two-phase strategy, where the first phase communicates with an old configuration c and the second phase communicates with a new configuration c'. A garbage-collection operation ensures that both a read-quorum and a write-quorum of c learn about c', and that the latest object value from c is conveyed to a write-quorum of c'.

We show atomicity for arbitrary patterns of asynchrony and failure. We analyze performance *conditionally*, based on timing and failure assumptions. For example, assuming that gossip and garbage-collection occur periodically, that reconfiguration is requested infrequently enough for garbage-collection to keep up, and that quorums of

active configurations do not fail, we show that read and write operations complete within time $8d$, where d is the maximum message latency.

A complete distributed system implementation is also underway.

12.5 Closing Remarks

Our approach to middleware (of which RAMBO is an example) differs from common practice: although middleware frameworks such as CORBA, DCE and Java/JINI support construction of distributed systems from components, their specification capability is limited to the formal definition of interfaces and informal descriptions of behavior. These are not enough to support careful reasoning about the behavior of systems that are built using such services. Moreover, current middleware provides only rudimentary support for fault-tolerance. In contrast, our services are precisely defined, with respect to both their interfaces and their behavior. The specified behavior may include performance and fault-tolerance. Our component behavior is specified in a compositional way, so that correctness, performance, and fault-tolerance properties of a system can be inferred from corresponding properties of the system's components.

Our project will, we believe, contribute substantially toward a coherent theory of algorithm design and complexity analysis for dynamic distributed environments, as powerful as the theory that currently exists for static distributed systems. The contributions of this project will be mainly theoretical. However, the resulting services and algorithms will also have the potential for impact on design of real systems for dynamic environments. Note that actually incorporating theoretical services like ours into systems will require additional work of another sort: software engineering work to integrate them with other system components built using object-oriented and component technologies (Birman discusses some of these issues in [12.2]).

References

12.1 H. Attiya, A. Bar-Noy and D. Dolev, "Sharing Memory Robustly in Message Passing Systems", *J. of the ACM*, vol. 42, no. 1, pp. 124-142, 1996.

12.2 Kenneth P. Birman. A review of experiences with reliable multicast. *Software, Practice and Experience*, 29(9):741–774, September 1999.

12.3 O. Cheiner and A. Shvartsman, "Implementing an eventually-serializable data service as a distributed system building block," in *Networks in Distributed Computing*, vol. 45, pp. 43–72, AMS.

12.4 *Communications of the ACM*, special section on group communications, vol. 39, no. 4, 1996.

12.5 R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman, "A dynamic primary configuration group communication service," in *Distributed Computing* Proceedings of DISC'99 - 13th International Symposium on Distributed Computing, 1999, LNCS, vol. 1693, pp. 64–78.

12.6 B. Englert and A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proc. of the 20th IEEE Int-l Conference on Distributed Computing Systems (ICDCS'2000)*, pp. 454-463, 2000.

12.7 A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman, "Eventually-serializable data service," *Theoretical Computer Science*, vol. 220, no. 1, pp. 113–156, June 1999.

12.8 A. Fekete, N. Lynch, and A. Shvartsman. "Specifying and using a partitionable group communication service." *ACM Trans. on Computer Systems*. vol. 19, no. 2, pp. 171-216, 2001.

12.9 R. Guerraoui and A. Schiper, "The Generic Consensus Service," *IEEE Trans. on Software Engineering*, Vol. 27, No. 1, pp. 29-41, January, 2001.

12.10 S. Haldar and P. Vitányi, "Bounded Concurrent Timestamp Systems Using Vector Clocks", *J. of the ACM*,Vol. 249, No. 1, pp. 101-126, January, 2002

12.11 Leslie Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2) 133-169, 1998.

12.12 N. Lynch and A. Shvartsman, "RAMBO: A Reconfigurable Atomic Memory Service", in *Proc. of 16th Int-l Symposium on Distributed Computing, DISC'2002*, pp. 173-190, 2002.

12.13 K. W. Ingols, "Availability study of dynamic voting algorithms," M.S. thesis, Dept. of Electrical Engineering and Computer Science, MIT, May 2000.

12.14 M. Merritt and G. Taubenfeld, "Computing with infinitely many processes (under assumptions on concurrency and participation)," In *Proc.14th International Symposium on DIStributed Computing (DISC)*, October 2000.

12.15 E. Yeger Lotem, I. Keidar, and D. Dolev. "Dynamic voting for consistent primary components." In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, August 1997.

13. Dissecting Distributed Computations

Rachid Guerraoui

Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne

13.1 Context

13.1.1 Historical Perspective

Beginning with the earliest assembly languages, programming even simple centralised and sequential applications was very cumbersome. Major advances in computing came with the adoption of programming languages that promoted the use of machine independent data abstractions like *sets, arrays*, and *records*, as well as control abstractions like *while, if* and *repeat*. With the arrival of parallel machines and multi-threading, the need for additional abstractions that factor out the difficulty of concurrent computing has driven several research activities. These activities resulted in new abstractions like *semaphores, monitors* and *rendez-vous*. Not very long ago, these abstractions used to sit beside programming languages in the form of libraries, e.g., concurrency libraries in C++. They have now been integrated as first class citizens into modern programming languages, e.g., through synchronised methods in Java.

Despite the many decades of research in distributed computing, abstractions for programming distributed applications are still lacking. Currently, the only abstraction that is of a relatively wide use in distributed programming is the *RPC (Remote Procedure Call)* [13.1]. Surprisingly enough, RPC actually aims at hiding distribution: the goal of RPC is to make an invocation of a remote object looks like the invocation of a local object. This illusion is however broken when the remote object is located on a machine that crashes or that becomes inaccessible because of a link failure [13.2]. In fact, *RPC* tackles only the easy part of distributed computing: *communication*. The heart of the difficulty is the handling of *partial failures* and this is in general ignored and left to the programmer. Some academic RPC-based systems indeed provide mechanisms to circumvent partial failure issues, e.g., transactional RPC [13.21]. Nevertheless, these mechanisms are very coarse-grained and rarely effective. In a sense, they do too much: they make the fundamental problem of handling partial failures transparent to the programmer [13.15].

13.1.2 Research Objective

Roughly speaking, programming a distributed application with just RPC, is like programming a concurrent application with just the *fork* abstraction, without means to control this concurrency, e.g., *semaphores*: the programmer would have to use operating system specific IO constructs. At the other extreme, programming a distributed application with transactional RPCs is basically like programming a concurrent application using a high level *actor* model [13.19], where concurrency is transparent and accesses

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 68–72, 2003.

to shared objects are automatically serialised. Experience shows that the programmer usually needs to be aware of concurrency through some finer-grained abstractions. Similarly for distributed computing, we need distribution-aware abstractions to provide the ability for the programmer to tune the way partial failures are handled. Of course, some recurrent coarse-grained solutions could be factored out within dedicated frameworks (e.g., transactional RPC). The programmer should however have the ultimate possibility to tune its solution by directly accessing finer-grained abstractions.

We advocate the search for distribution-aware abstractions as a challenging and promising research objective. We suggest below a pragmatic *dissection* approach to achieve this goal and we illustrate the approach through our experience with the celebrated Paxos replication algorithm [13.20]. Paxos provides an effective way to build reliable distributed services that are resilient to transient failures of machines and network. Basically, we suggest two abstractions that enable us to express Paxos in a modular way, as well as construct powerful variants of it [13.3]. Our abstractions are *overhead-free* in the sense that solutions based on these abstractions, although modular, are neither less efficient nor less resilient than ad-hoc solutions bypassing these abstractions.

Behind the stated goal of arguing for these abstractions, the hidden objective is to discuss some guidelines for the more general and ambitious quest towards identifying the holy grail abstractions for distributed computing.

13.2 Approach

Identifying meaningful and useful abstractions for distributed computing out of thin air requires a lot of imagination. A more accessible approach consists in *dissecting* good practices in distributed computing. A successful dissection would go through *deconstructing* such practices by factoring out their underlying key abstractions, and then would allow, by re-composing these abstractions or re-implementing them differently, the *reconstruction* of powerful variants of these practices. We illustrate below this approach through our experience with Paxos [13.20].

13.2.1 Distributed Computing Practices

Paxos solves a fundamental problem in distributed computing. It coordinates the execution of a set of replica processes of a shared service so that these processes behave in a consistent way. Basically, Paxos builds a highly-available and consistent distributed (deterministic) implementation of a service out of non-reliable replicas of a sequential and centralised (deterministic) implementation of the service. Paxos does so in an efficient and resilient way. It assumes some (weak) synchrony assumptions [13.11] that are known to be necessary to ensure coordination among replica processes, yet tolerates transient failures of the network: it is *indulgent* in the sense of [13.16]. The algorithm does not preclude the possibility that all the processes might crash and recover, as long as a majority of the replica processes eventually recover and do not crash again for sufficiently long time. This assumption is necessary to ensure the coordination among the replicas, given the indulgence characteristic of Paxos.

13.2.2 Abstractions

We sketch below two abstractions that were keys in our faithful deconstruction and reconstructions of Paxos [13.3]: a *weak* form of *leader election*, denoted by $\diamond$*Leader* (*eventual leader*), and a *weak* form of *storage*, denoted by $\diamond$*Register* (*eventual register*).

1. $\diamond$*Leader* eventually elects a unique correct leader among a set of processes. Its specification is weak in that it does not preclude the existence of concurrent leaders for arbitrary periods of time. It corresponds to the failure detector Ω introduced in [13.5] and thus captures the exact amount of synchrony needed to ensure agreement among processes [13.5], e.g., agreement on a common state among the replicas of a service. $\diamond$*Leader* can be viewed as a *time-oriented* abstraction in the sense that it captures synchrony assumptions.
2. $\diamond$*Register* provides an abstraction of a distributed storage primitive. It can be viewed as a *space-oriented* abstraction. Its specification is weak in the sense that, if two processes try to store different values at the same time, they might both fail. The storage only succeeds if a single process keeps on trying. Once a value is stored, it is stored forever. $\diamond$*Register* can be implemented in a message passing distributed system where processes and channels can crash and recover, provided that a majority of the processes eventually recover and, for sufficiently long, do not crash again and communicate in a reliable manner [13.3]. Given this majority assumption, the implementation of $\diamond$*Register* does not need any synchrony assumption.

13.2.3 Deconstructions and Reconstructions

In the following, we summarize how we used our abstractions to build modular and powerful variants of Paxos.

- In [13.3], we give simple implementations of our abstractions in a crash-stop model with reliable channels. By composing our abstractions, we show how to obtain a simple variant of Paxos in this crash-stop model as well as how to easily port the algorithm to a crash-recovery model by mainly modifying the implementation of our $\diamond$*Register* abstraction. The resulting algorithm is indeed a faithful deconstruction of Paxos. We also show how to adapt the algorithm when (a) some of the processes do never crash, (b) some of the processes might keep on crashing and recovering, or (c) the processes and the disks are disconnected. The latter case leads to a faithful deconstruction of *Disk Paxos* [13.14][1].
- In [13.9], we show how, through a different composition of our abstractions, we efficiently implement a reliable, possibly *non-deterministic*, statefull (vs stateless in [13.13]) service. We obtain here what is called a *universal construction* in a message passing model [13.18], i.e., an algorithm that transforms any sequential specification of a given service into a highly-available distributed implementation of the service. Our algorithm is indeed modular, and even more efficient than the most efficient universal construction in a message passing model we knew of. Thanks to the use

[1] Following a similar approach, a recent variant of Paxos was proposed in [13.7] to handle an infinite number of client processes accessing the reliable service.

of our abstractions, the algorithm is very simple and can easily be adapted to trade performance in steady-state with recovery time.

Reliable distributed services can indeed be built with higher level abstractions like *consensus* [13.12, 13.17]. However, with a consensus abstraction, one cannot obtain an efficient scheme in a practical crash-recovery model unless the abstraction is broken [13.3]. In short, $\diamond$*Leader* and $\diamond$*Register* are overhead-free, whereas abstractions like *consensus* are not. In fact, consenus is coarser-grained abstraction can be effectively implemented based on our finer-grained abstractions: $\diamond$*Leader* and $\diamond$*Register*.

13.3 Directions

It is argued here that, despite the many years of research in the area of distributed computing, we know so far of no candidate abstraction to adequately tackle the fundamental issue of the area: partial failures. There is an interesting challenge in devising such abstractions.

How can we know that an abstraction is good? Like many engineering activities, identifying adequate programming abstractions has an inherent artistic nature. Even if abstraction is a subjective exercise, it is sensible however to require that:

- An adequate abstraction be *meaningful* in the sense that it should encapsulate some non-trivial aspect of distributed computing.
- An adequate abstraction be *overhead-free* in the sense that modular solutions based on this abstraction should be as resilient and efficient as ad-hoc solutions that bypass such abstractions[2].

We propose in this position paper a pragmatic approach to come up with such abstractions. Basically, we suggest to dissect effective distributed computing practices, then to factor out of such practices, overhead-free abstractions that enable to build more modular variants of these practices. We illustrated our approach through our experience with the Paxos replication algorithm and two candidate underlying abstractions: a *time-oriented* abstraction, representing a weak form of *leader election*, and a *space-oriented* abstraction, representing a weak form of *storage*. These abstractions handle failures by providing effective solutions to the fundamental problem of making a distributed service look like if it was running on a highly-available centralised machine; pretty much like *semaphores* and *monitors* provide effective solutions to, probably the most fundamental problem in concurrent computing: making a shared service look like if it was accessed by one user at a time.

In general, identifying adequate abstractions for distributed programming is a very challenging task that would require a considerable mixture of theoretical and practical work. Once identified and experimented, a complementary part of the task would lead to integrating these abstractions into modern programming languages.

[2] Of course, no abstraction is really overhead-free because it adds at least the cost of a local procedure call. But this cost is negligible with respect to remote communication and disk accesses.

Acknowledgements

The author is extremely grateful to the anonymous referees who helped improve the presentation of this position paper.

References

13.1 A. Birell and B. Nelson. *Implementing Remote Procedure Call.* ACM Transactions on Computer Systems, February 1984.
13.2 K. Birman and R. van Renesse. *RPC Considered Inadequate.* In Reliable Distributed Computing with the Isis Toolkit. K. Birman and R. van Renesse (eds). IEEE Computer Society Press. 1994.
13.3 R. Boichat, P. Dutta, S. Frolund and R. Guerraoui. *Deconstructing Paxos*. DSC-EPFL Tech-Report 2001-06.
13.4 T. Chandra and S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Journal of the ACM, 43(2), March 1996.
13.5 T. Chandra, V. Hadzilacos and S. Toueg. *The Weakest Failure Detector for Solving Consensus.* Journal of the ACM, 43(4), July 1996.
13.6 T. Chandra, V. Hadzilacos and S. Toueg. *On the Impossibility of Group Membership.* ACM Symposium on Principles of Distributed Computing (PODC), 1996.
13.7 G. Chockler and D. Malkhi. *Active Disk Paxos with Infinitely Many Processes.* ACM Symposium on Principles of Distributed Computing (PODC), 2002.
13.8 R. De Prisco, B. Lampson and N. Lynch. *Revisiting the Paxos Algorithm*. Theoretical Computer Science, 243(1-2), 2000.
13.9 P. Dutta, S. Frolund, R. Guerraoui and B. Pochon. *An Efficient Lightweight Universal Construction in a Message Passing System*. International Symposium on Distributed Computing (DISC), Springer Verlag (LNCS), 2002.
13.10 P. Dutta and R. Guerraoui. *The Inherent Price of Indulgence.* ACM Symposium on Principles of Distributed Computing (PODC), 2002.
13.11 C. Dwork, N. Lynch and L. Stockmeyer. *Consensus in the Presence of Partial Synchrony.* Journal of the ACM, 35 (2), 1988.
13.12 M. Fischer, N. Lynch and M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM, 32(2), 1985.
13.13 S. Frolund and R. Guerraoui. *X-ability: A Theory of Replication.* Distributed Computing, 14:231-249, 2001.
13.14 E. Gafni and L. Lamport. *Disk Paxos*. International Symposium on Distributed Computing (DISC), Springer Verlag (LNCS), 2000.
13.15 R. Guerraoui. *What object-oriented distributed programming does not have to be, and what it may be*. Informatik 2, 1999.
13.16 R. Guerraoui. *Indulgent Algorithms.* ACM Symposium on Principles of Distributed Computing (PODC), 2000.
13.17 R. Guerraoui and A. Schiper. *The Generic Consensus Service.* IEEE Transactions on Software Engineering, 27 (1), 2001.
13.18 M. Herlihy. *Wait-Free Synchronisation*. ACM Transactions on Programming Languages and Systems, 13 (1), 1991.
13.19 C. Hewitt. *Viewing Control Structures as Patterns of Passing Messages*. Artificial Intelligence, (8), 1977.
13.20 L. Lamport. *The Part-Time Parliament*. ACM Transactions on Computer Systems, 16 (2), 1998.
13.21 B. Liskov and R. Scheifler. *Guardians and Actions: Linguistic Support for Robust Distributed Programs.* ACM Symposium on Principles of Programming Languages (POPL), February 1986.

14. Ordering *vs* Timeliness: Two Facets of Consistency?

Mustaque Ahamad[1] and Michel Raynal[2]

[1] College of Computing, Georgia Tech, Atlanta, GA 30332, USA
mustaq@cc.gatech.edu
[2] IRISA, Université de Rennes, Campus de Beaulieu, 35 042 Rennes, France
raynal@irisa.fr

14.1 Introduction

Distributed applications are characterized by the fact that the processes they are made up of execute on possibly geographically dispersed nodes. An important problem the underlying distributed system has to solve lies in maintaining the consistency of the state that is shared by such processes. Unfortunately, the non-instantaneity of message transmissions and failure occurrences make this fundamental task far from being trivial. Of course, this difficulty depends on the type of consistency required by the application. Distributed programming models based on shared variables have been advocated by many researchers. Basically, a consistency criterion states which value has to be returned when a process reads a variable of the shared state. We think that there are two basic axes that help characterize consistency criteria: *ordering* and *timeliness*. The ordering axis defines the possible orders in which operations can be executed while returning values for read operations that are permitted by the consistency criterion. The timeliness axis defines how soon a value written by one process must become visible to others. By exploring these two axes, one can not only define versatile consistency criteria that meet the needs of diverse applications, but consistency levels can also be adapted based on available system resources or changing needs of an application.

We believe that the characterization of consistency criteria using the orthogonal axes of ordering and timeliness helps us understand important issues related to shared objects in distributed systems. For example, scalability of protocols that implement a consistency criteria depends on both ordering as well as timeliness. A more scalable protocol can be derived if the ordering requirement can be relaxed (e.g., the consistency criteria permits more orderings) or a lower degree of timeliness is acceptable. Furthermore, timeliness can be dynamically adapted based on application needs and available system resources. We survey the various consistency criteria that have been proposed and discuss how they can be characterized by the two axes proposed by us. We also list open problems that still need to be addressed and provide future research directions in this field.

14.2 Ordering-Based *vs* Timed Consistency

14.2.1 Ordering-Based Consistency

The set of objects defining the shared state is accessed by operations. Such an operation traditionally spans a single object and reads or modifies it. A more general model considers operations that span several objects.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 73–77, 2003.

When the operations are unary. Atomic consistency (also called linearizability, LIN) [14.5] assumes that, for each execution of a set of processes that share objects, all operations issued to the objects can be totally ordered in such a way that the total order respects real-time order, and each read returns the value of the latest write that precedes it in the total order). Thus, LIN requires a total order on which all processes agree and the timeliness requirement states that a written value is visible as soon as the write operation's execution completes. Sequential consistency (SC) [14.8] is weaker than LIN in the sense that the total order is not required to respect timeliness constraints. Hence, SC is based on a logical time notion, while LIN is based on real-time. The main advantage of LIN is the fact it is a *local* property. A property P is local if the system as a whole satisfies P whenever each of its component object satisfies P. Locality is a first-class notion as it expresses the fact that basic components can be "naturally" composed (i.e., without paying additional price) to get a system enjoying the same property as its basic components. LIN is a local property, while SC is not [14.5] (actually, SC can be seen as a form of lazy LIN [14.11]). Interestingly, it is possible to design a system in which some objects are linearizable, while others are sequentially consistent.

The fact that intrinsically LIN and SC are total order-based criteria make them easy to use but very difficult or costly to implement in systems prone to node failures or in large scale systems. In the former case, as building a total order is as difficult as solving the consensus problem, systems designers are faced with the FLP impossibility result. In the latter case, as (due to its very definition) the total order is unique, when we increase the number of nodes or the number of objects, the coordination required to ensure such total order could result in overheads that limit the scaling of the protocols that implement the consistency criteria.

Causal consistency (CC) [14.1, 14.2] weakens the ordering requirement to causal order as it allows conflicting write operations to be concurrent. Hence concurrent read operations of the same object can get different values (those being produced by concurrent conflicting writes). Not being based on a total order, CC does not suffer the scalability limitations of LIN and SC. But, to date, only very few applications have been identified that require a consistency criterion as weak as CC. There is no timeliness requirement in CC and the value written by one process may not become visible to other processes for an arbitrarily long time. Other orderings that define consistency criteria such as processor consistency (PC) and pipelined RAM (PRAM) have also been proposed.

When the operations span several objects. There are two cases. The first is when the operations are predefined. It is shown in [14.4, 14.9] that it is possible to define a consistency criterion (Normality) which is as strong as LIN when operations are on a single object, but weaker when they span several objects. A second case consists in considering transactions and extending the previous consistency criteria to include operations that span multiple shared objects. This is the approach we sketch next.

From serializable to causal transactions. While serializability considers that all processes must have the same sequential view of the whole execution, causal transaction consistency is weaker. It allows each process to have its own sequential view of the execution as long as each individual view preserves the causality relation. (Let us note

that if every transaction is reduced to a single read or single write operation, then causal transaction consistency becomes identical to CC [14.1, 14.15]).

For some applications, serializability is too strong a consistency criterion while causal transaction consistency is too weak. With causal consistency, when two update transactions that write into the same object are concurrent, they can be ordered differently by two processes in their views of the execution. This could lead to different final states of the system according to different processes. Causal serializability prevents such a possibility by adding the following ordering constraint to causal consistency: all transactions that update the same object must be perceived in the same order by all processes. This constraint ensures that, for each object, there is a unique 'last' value on which all processes agree. It follows from this definition that causal serializability lies between causal consistency and serializability.

Formal definitions of these consistency criteria and protocols implementing them are described in [14.12]. Interestingly these protocols have the following characteristics. Assuming that each process has a vote for every object x, acquiring a read or write access while preserving the ordering constraints amounts to getting a sufficient number of votes. Let r_x (resp. w_x) the number of votes a process has to obtain in order to read (respt. write) x. Table 14.1 depicts the number of votes required by a read (respt. write) operation in protocols that support different levels of consistency (n is the number of processes). As can easily be seen from the table, the ability to execute transactions by contacting fewer other nodes decreases from causal consistency to causal serializability and from the latter to serializability. Interestingly, it is possible to dynamically switch from one consistency criteria to another.

Table 14.1. Protocol Synchronization Cost

number of votes to access object x	protocol ens. causal cons.	protocol ensuring causal serializability	protocol ensuring serializability
read by a query	$r_x = 0$	$r_x = 0$	$r_x = 0$
read by an update	$r_x = 0$	$r_x = 0$	$r_x + w_x > n$
write by an update	$w_x = 0$	$w_x \geq (n+1)/2$	$w_x \geq (n+1)/2$

14.2.2 Timed Consistency

Notions of consistency based on a timeliness parameter (Δ) have been introduced in [14.3] to define time-constrained causal message delivery, and in [14.13] to define a temporal programming model. We have investigated such a notion to define consistency notions in [14.16, 14.17, 14.18]. Timed consistency requires that if the effective time of a write is t, then no value written before t can be seen by a process after time $t + \Delta$, where Δ is is a parameter of the consistency model.

Timed Consistency can be considered with any type of ordering-based consistency criteria. We get the hierarchy of consistency criteria depicted in Figure 14.1 (where TSC and TCC stand for Timed SC and Timed CC, respectively). Interestingly, when we consider "strong" criteria, i.e., the ones based on a total order, $\Delta = 0$ corresponds with LIN and $\Delta = +\infty$ corresponds with SC.

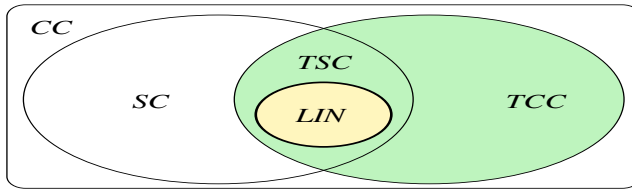

Fig. 14.1. Hierarchy of Consistency Criteria

These timed consistency notions have been developed in [14.7] where, in addition to read and write operations, timed-read and timed-write operations are proposed. A timed-read obtains a value of the object that is no older than the most recent value at time $t - \Delta$ (where t is the execution time of the operation). A timed write defines a new value for the object and also ensures that this value is perceived at all processes no later than Δ time units after the write completion. Timed operations can be used with non-timed operations. A formal definition of a consistency criterion based on such operations, an implementation protocol, and a distributed application (traffic information dissemination) based on this model are described in [14.7]. Recently, to improve scalability of replicated services, Yu and Vahdat have investigated continuous consistency models that explore ordering, timeliness as well as an application defined error measure [14.14]. Although applications can bound the degree of inconsistency, the complexity of programming distributed applications is increased due to the need for determining such bounds.

14.3 Future Directions

While maintaining consistency of the state of a distributed application is a fundamental problem, many consistency-centered problems remain open. We list here a few of them.

- Provide a formal definition of scalability. Only then, it will be possible to prove that protocols have the "scalabilty" property.
- Are scalability and strong consistency criteria antagonistic concepts? How do we characterize the relationship between scalability and consistency in a quantitative manner?
- As mobile and hand held computers become common, is it possible to define a consistency criteria that takes such heterogeneity into account? In particular, resource limited computers may only ask for weaker consistency whereas resourceful nodes can access the shared objects with strong consistency.
- Is it possible to characterize the class of applications that require strong consistency (weak consistency)? More generally, given a consistency criteria, is it possible to characterize the class of problems it allows to solve?
- Can consistency level be offered as a quality-of-service (QoS) contract by the system to an application? If such a level cannot be provided, is there a notion of "gracefully degrading consistency" that would be meaningful. Is such an approach meaningful in the presence of faults?
- Is it possible to define a general consistency criterion based on an ordering parameter plus a timeliness parameter?

– How to trade relaxed consistency for better response time [14.6]? How to take into account the semantics of values to get meaningful consistency criteria that have efficient implementation [14.10]?

References

14.1 Ahamad M., Hutto P.W., Neiger G., Burns J.E. and Kohli P., Causal memory: Definitions, Implementations and Programming. *Dist. Computing*, 9:37-49, 1995.

14.2 Ahamad M., Raynal M. and Thiakime G., An Adaptive Architecture for Causally Consistent Distributed Services. *Distributed Systems Engineering Journal*, 6(2):63-70, 1999.

14.3 Baldoni R., Mostefaoui A. and Raynal M., Causal Delivery of Messages with Real-Time Data in Unreliable Networks. *Real-Time Systems J.*, 10(3):245-262, 1996.

14.4 Garg V.K. and Raynal M., Normality: a Correctness Condition for Concurrent Objects. *Parallel Processing Letters*, 9(1):123-134, 1999.

14.5 Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463-492, 1990.

14.6 Krishnamurthy S., Sanders W.H. and Cukier M., An Adaptive Framework for Tunable Consistency and Timeliness Replication. *Proc. IEEE Int. Conf. on Dependable Systems and Networks (DSN'02)*, Whashington DC, June 2002.

14.7 Krishnaswamy V., Ahamad M., Raynal M. and Bakken D., Shared State Consistency for Time-Sensitive Distributed Applications. *Proc. 21th IEEE Int. Conf. on Dist Comp Systems (ICDCS'02)*, pp. 606-614, Phoenix (AZ), April 2001.

14.8 Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

14.9 Mittal N. and Garg V.K., Consistency Conditions for Multi-Object Distributed Operations. *Proc. 18th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'98)*, pp. 582-589, Amsterdam (Netherland), 1998.
Springer Verlag LNCS #938, pp. 224-241, Dagsthul Castle (Germany), 1994.

14.10 Pereira J., Rodrigues L. and Oliveira R., Reducing the Cost of Group Communcation with Semantic View Synchrony. *Proc. IEEE Conf. on Dependable Systems and Networks (DSN'02)*, pp. 293-302, 2002.

14.11 Raynal M., Sequential Consistency as lazy Linearizability. *Proc. 14th ACM Int. Symp. on Parallel Algorithms and Architectures (SPAA'02)*, Winnipeg (CA), 2002.

14.12 Raynal M., Thiakime G. and Ahamad M., From Causal to Serializable Transactions. *BA, 15th ACM Symposium on Principles of Dist. Comp.*, pp. 310, 1996.

14.13 Singla A., Ramachandran U. and Hodgins J., Temporal Notions of Synchronization and Consistency in Beehive. *Proc. 9th ACM Symposium on Parallel Architectures and Algorithms (SPAA'97)*, pp. 211-220, Newport (RI), 1997.

14.14 Yu, H. and Vahdat, A., Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM TOCS*, 20(3):239-282, 2002.

14.15 Theel O. and Raynal M., Static and Dynamic Adaptation of Transactional Consistency. *Proc. 30th Int. Conference on Systems Sciences (HICSS'97)*, Vol I:533-542, Maui (Hawaii), 1997.

14.16 Torres F., Ahamad M. and Raynal M., Lifetime-Based Consistency for Distributed Objects. *Proc. 12th Int. Symposium on Distributed Computing (DISC'98)*, Springer Verlag #1499, Andros, Greece, pp. 378-392, 1998.

14.17 Torres F., Ahamad M. and Raynal M., Timed Consistency for Shared Distributed Objects. *Proc. 18th ACM Int. Symposium on Principles of Distributed Computing (PODC'99)*, Atlanta, pp. 163-172, 1999.

14.18 Torres F., Ahamad M. and Raynal M., Real-Time Based Consistency Models for Distributed Objects. *Journal of Computer Systems Science and Engineering*, 17(2):133-142, 2002.

Part II
Exploring Next-Generation Communication Infrastructures and Applications

Next-generation network applications are increasing both in scale and complexity. Now more than ever, application writers need infrastructure support to simplify resource management in wide-area networks. Ideally, such an infrastructure would present to the application a network layer with many of the properties of a local area network (low latency, high bandwidth, reliable links). Communication needs to be reliable, fast, and stable. Applications need to find and communicate with objects and resources in the network easily and with minimal delay, similar to functionality offered by local directory services. At the same time, this ideal layer would give applications access to wide-area resources, including storage, bandwidth and computational cycles, and allow applications to manage them in a scalable and stable manner.

Part II outlines some efforts making initial progress towards this infrastructure, identifies potential difficulties of deployment in a real network, and discusses future directions to solving these problems.

We begin with two papers that explore the potential power of the wide-area network. The paper by Johansen, van Renesse, and Schneider (page 81) presents a novel infrastructure for the traditional publish/subscribe model, in which a topic hierarchy plays a crucial intermediary role in connecting producers with interested consumers. Collaborative filters extend this model in a flexible fashion. In the second paper (page 87), van Renesse discusses the benefits of Astrolabe, a scalable aggregation infrastructure based on a gossip protocol. This aggregation model further outlines the number of potential applications made possible by increasing network scale and reach.

Recent work has significantly advanced the state of the art in network infrastructures, particularly in the direction of a decentralized, peer-to-peer model. Recent projects such as CAN, Chord, Pastry, and Tapestry, among others, leverage a decentralized routing mesh to support routing and lookup of objects by location-independent names. The paper by Malkhi (page 93) presents details of Viceroy, a similar system oriented towards surviving attacks of adaptive adversaries who can control the arrival and departure of nodes in the network.

While such peer-to-peer systems are shown to scale in theory and under simulation, we have yet to deploy and evaluate their behavior under stress in a real large-scale network. Several issues remain, including scaling up all aspects of the system, providing application stability under unpredictable conditions, and constraining complexity in the large scale. The next group of papers discuss the issue of scalability. The paper by Birman (page 97) examines the real-world implications of deployment from the perspective of two research communities, those who favor stronger or provable guarantees and those

who favor less stringent properties. It discusses the advantages and disadvantages of systems with strong properties, and makes a compelling argument for the inclusion of strong components in order to ensure reliability and predictability. Finally, the paper considers whether background overhead such as state maintenance and recovery algorithms in these infrastructures can grow super-linearly as the system scales, and contribute to overall instability. The paper by Castro, Druschel, Hu and Rowstron (page 103) attempts to address the scalability issue by examining how these systems deal with locality and presents a specific solution to improving performance and stability.

The next two papers deal with the issues of stability and complexity. Here we have to temper the expectation of our theoretical results with realistic expectations on the types of faults that occur in the real internet. Many existing proposals deal with network failures, but few make the distinction between normal application level traffic and control traffic which has much stronger reliability requirements. Veríssimo (page 108) proposes a novel solution based on the "wormhole" construct. The paper by Friedman (page 114) considers an approach where some of the unpredictable network behavior is allowed to propagate up to the application level. The paper introduces the notion of "fuzziness" into traditional problems such as transactional processing and group membership.

Finally, we shift our attention to the issue of complexity. As applications grow in scale, an increasing number of components and variables begin to affect the performance of the overall system. Management of complexity is a difficult problem, and often cannot be solved by extending solutions derived on a smaller scale. A completely orthogonal approach, then, is to observe the successful management of similar systems in nature. Complex systems such as bird flocks in migration or ant colonies exhibit surprisingly resilient and stable behavior through "emergent behavior," despite the apparent simplicity of each single component. Our last paper by Montresor, Babaoğlu and Meling (page 119) proposes a deeper examination of these successful systems in order to discover principles for fostering emergent behaviors and to reduce complexity in large scale systems.

It is clear we are making significant progress towards the ultimate goal of a stable, manageable global-scale application infrastructure. An increasing number of the current proposals are being deployed in the wide-area network. For example, researchers are deploying and experimenting with several P2P network infrastructures on the global PlanetLab network testbed (http://www.planet-lab.org). While we analyze the merits of current infrastructure proposals, directions presented here provide road maps for future designs, moving us closer to truly scalable systems with provable stability and complexity properties.

Ben Y. Zhao

15. WAIF: Web of Asynchronous Information Filters*

Dag Johansen[1], Robbert van Renesse[2], and Fred B. Schneider[2]

[1] Dept. of Computer Science, University of Tromsø, Tromsø, Norway
dag@cs.uit.no
[2] Dept. of Computer Science, Cornell University, Ithaca, NY 14853
{rvr,fbs}@cs.cornell.edu

Summary. The Internet is seeing a rapid increase in on-line newspapers and advertising for new products and sales. Yet only primitive mechanisms are available to help users discover and obtain that subset of these news items likely to be of interest. Current search engines are really only first step. For locating news providers, word-of-mouth and mass mailings are still used; for retrieval of news items, users are forced to poll web sites regularly or provide e-mail addresses for follow-up mailings.

WAIF is a new framework to facilitate easy user access for Internet users to relevant news items. WAIF supports new kinds of browsers, personalized filters, recommendation systems, and – most importantly – an evolution path intended to enable efficient deployment of new techniques that enhance the user retrieval experience.

15.1 Challenges

Today's World Wide Web has begun to offer convenient mechanisms for locating and retrieving information. But search engines – like Google and AllTheWeb – and other of current technology only work well for information that is relatively static and remains relevant for long intervals. More and more, we see on-line services providing highly dynamic kinds of information. This information has value for only a short period of time and thus might be stale by the time it has been recorded in a search engine's index. We call such information *news* and are driven to provide high precision access; our goal is an easy way of getting news items to exactly those people who have an interest in that news.

In today's WWW *publishers* have few mechanisms to identify the set of *consumers* for news item dissemination. A publisher either must wait for a subscription or generate an ad hoc mailing list. With subscriptions, news items reach only a small set of the interested parties; email blitzes ("spam") reach many people outside the target audience.

Consumers also have few mechanisms to specify what information they are interested in. The consumer must find, often by chance, and subscribe to publishers whose output has high overlap with the consumer's interests. Interesting information from other publishers is not seen, and unsolicited and irrelevant news items are received from publishers employing ad hoc mailing lists.

* This research was funded in part by the Norwegian Research Council (IKT-2010 Program), in part by DARPA/AFRL-IFGA grant F30602-99-1-0532, and in part by the AFRL/Cornell Information Assurance Institute.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 81–86, 2003.

15.2 Future Research

WAIF (Web of Asynchronous Information Filters) is a new project that attempts to address these current inadequacies of WWW by supporting real-time news location, routing, filtering, and analysis. In short, WAIF provides a framework to enable news publishers to reach interested consumers. The architecture offers a standard protocol for users to subscribe to news item streams and for publishers to publish news items. A small set of WAIF mechanisms facilitates the construction of collaborative filtering and recommendation systems. So, subscribers are able to rank publishers and to re-publish news items that interest certain other communities. We are mindful that the success of WAIF thus depends heavily on having a user-friendly browser, so this is a central research concern for us.

15.2.1 An Information Overlay Network

WAIF is essentially an Overlay Network [15.1] where the endpoints are publishers and consumers; the WAIF transport infrastructure contains mechanisms to rank news items for each consumer individually as well as for routing messages to consumers according to this ranking. In WAIF, consumers explicitly subscribe to publishers, and consumers have a convenient way to rate the news stream provided by each publisher. The paradigm can be likened to a sound mixer control panel, with a slide control for each subscription; we are considering the mixer panel as part of our WAIF browser. This is a prototype browser similar in respect to the Curious Browser [15.3], which infer user interests based on a combination of explicit and implicit ratings.

A consumer in WAIF can be a producer as well. If a consumer C receives a news item that C thinks is of interest to other consumers, then C can re-publish this news item. Similarly, if C happens across an interesting web page or receives some interesting e-mail, then C may publish this information as a news item. (We discuss below how re-publication is specified using a simple drag-and-drop interface.) Note how WAIF blurs the distinction between publishers and consumers. Both are called WAIF *principals*.

Each WAIF principal can publish news items to one (or more) *topics*. The "topic hierarchy" can be created as the principal sees fit. For example, the New York Times might create topics like "news/politics/international" and "money/stock". An individual might create "personal/family", "personal/bowling", and "work" topics. For the WAIF browser, we display a tree to represents this topic hierarchy. An individual publishes web pages or re-publishes news items simply by dragging them onto the correct node of this topic hierarchy. This is much like dragging e-mail messages into a folder hierarchy.

If a WAIF consumer does not think the news item is worthy of re-publishing, the consumer can either delete a news item after reading it, or drop the news items into a "garbage can." The latter indicates annoyance with the news item. Such actions enable WAIF algorithms to improve how subsequent news items are ranked. (Other methods to get ranking input include keeping track of reading time, mouse movement, etc.)

A URL is associated with each WAIF principal and with each topic to which the principal posts news items. Examples might include

"waif://nytimes.com/news/politics" or
"waif://aol.com/personal/john/family".

Consumers subscribe to such URLs; a subscription generates a record for the subscriber, including the *mailbox* at which news items will be delivered. The mailbox is similar to a standard SMTP mailbox, and similar (if not the same) protocols may be used to ensure reliable delivery of news items.

Third parties can deploy information *filters* and *fusers*. These news item processors are WAIF principals; they subscribe to WAIF URLs while also publishing information based on their input. We intend to use our TOS system [15.7] so that users can upload new filters into the WAIF infrastructure in a safe manner. Filters might even migrate between TOS servers in order to optimize scalability or other notions of performance.

Not all filters add value by enhancing precision. Some filters might be deployed to improve scalability of WAIF or to support anonymous subscription. Other filters might maintain state and attempt information synthesis. For example, a filter might analyze the news items published by several stock exchanges and publish forecasts.

15.2.2 Personalized Filtering

News-on-demand systems that automatically process news and provide personal presentations are currently being constructed [15.8]. We are developing a personalized filtering system that allows individuals to do content-based filtering. We call such a system of filters a "PONS" (Personal Overlay Network System)[1]. Each user will be able to deploy his or her own PONS using a set of filters, possibly obtained from the web. The PONS infrastructure will automatically place the filters on appropriate TOS servers to minimize network resources, while maximizing sharing between users.

We will use a PONS prototype under construction to illustrate this concept. The goal is that a novice Internet user shall be able to configure and transform the Internet into a highly personalized, asynchronous and autonomous distributed filtering network with high precision and recall. Creating this PONS works as follows.

Initially, the user specifies interest in certain predefined UDDI conformant topics through a *WAIF-browser*. The user does not have to know anything about programming and how to deploy filters, location of remote servers and the like; all that is needed is personalized preferences specified through scroll down menus. The net effect of this dialogue is a file with what we call a *user profile*, a list of topics, some general, some very detailed. This user profile is submitted to a remote *WAIF Deployment server*.

A WAIF Deployment server acts as an advanced match-making server. It keeps track of remote data sources and tries to match a user profile with these. Locating resources is a key problem, and both pull- and push-based techniques are investigated. This includes use of pull-based centralized search solutions [15.9] and peer-to-peer techniques [15.6], to more push-based schemes where data sources update the WAIF Deployment server with new directory information.

Upon successful location of a convenient data source, filters have to be composed for deployment. It is a key requirement that a regular user should not be involved in this specialized task. Therefore, the WAIF Deployment server transforms a user profile into one or a set of filters. We have identified a set of software patterns in this type of computations and have devised a collection of reusable pattern-based *nano-filters*.

[1] A *pons* is also a relay station between the brain and the spinal cord.

Hence, a nano-filter (code) is coupled with specific user data and is deployed close to the data source. This deployment is at one or several *WAIF Filter servers*, which act as advanced mediators [15.10].

A WAIF Filter server either produces data itself or subscribes to data streams from other data sources. This can be traditional topic- or content-based Internet data sources. The nano-filters can now parse the data streams, and specific alerts triggered lead to notifications being sent to the user. This might create a precision problem closely related to how users experience e-mail spam today, and we approach this problem by sticking one or a stack of *spam filters* into the upstream data feed in the PONS. The idea is that the nano-filters do the course grain data filtering, while spam filters do more and more fine grained filtering. Typically, the spam filter is stateful, while the nano-filters are, besides eventual parameters, stateless. Finally, data passed through the spam filters, ends up at a *relay filter*, a highly context sensitive distribution filter much like a MS .Net Passport Alert service.

We have now described how a user can create and deploy his own data fusion and filtering PONS. Other PONS services are also being constructed. For instance, a user profile can be submitted to a *WAIF Profiler*. This is a server that takes a user profile as input and generates an HTML-page of recommendations for that particular profile. In our first naive implementation this is a personal start page for traditional web browsing. This page evolves over time based on user actions recorded by the WAIF browser.

A more elaborate PONS is one using collaborative filtering techniques in a socialware context [15.5] The idea is that captured preferences of multiple users can be used to recommend comprehensive events or items of interest to other users. Multiple PONS from like-minded users are connected together through *WAIF Recommender servers*. A horizontal network of such servers co-operates and exchanges data to predict additional topics or products a user might like.

15.2.3 An Information Market Place

WAIF defines a new web – one in which the links join principals that communicate through subscriptions and thus capture relationships based on how information is being used (rather than how the original author intended it to be used, which is what WWW hypertext links do today). This new web may be crawled and indexed, just like today's WWW. (Each WAIF URL may have a short XML description associated with it to allow for keyword search). And, as with the WWW, information sources may be ranked. Unlike WWW, the links in the WAIF have weights associated with them, hopefully leading to improvements in relevance ranking.

WAIF will go a step beyond passive relevance ranking and notify consumers automatically of news item sources that might interest them. This is similar to what Amazon does when it suggests or recommends other items of interest to a consumer. Such a service can be implemented in WAIF by a filter. We expect many such filters to coexist, just like today there are many search engines to choose from. A consumer can subscribe to one or more of these WAIF filters and relate his or her "profile" (the set of subscriptions of the consumer, along with the corresponding rankings). Based on this information and information obtained by crawling, the filter could then post recommendations as news

items which, just like any other news item source, the user can rank using the mixer control panel, and/or re-publish.

We envision that WAIF will form a market place of information. Consumers negotiate contracts with publishers for information. Such a contract specifies not only what a consumer will pay the publisher for information, but also restrictions on what the consumer is allowed to do with said information. Subsequent re-publishing of received information, for example, may be restricted by copyright protection, or may require extra payment. It is unlikely that mechanisms can be implemented that directly enforce the terms of such contracts, but we are interested in extending WAIF with automated auditing and tracking mechanisms that may help in tracking down violations.

Although there is a similarity between WAIF and newsgroups (and, if you will, the publish/subscribe paradigm), what we are proposing is fundamentally different. In newsgroups, publishers post messages to particular groups, forcing the publisher to anticipate which communities will be most interested in the news item. These communities are explicit collections of users (even though the subscribers are anonymous) joined by simple notions of affinity. In WAIF, publishers do not publish to any explicit group of subscribers. In that sense, WAIF is closer to a content-based [15.2, 15.4] rather than to a topic-based publish/subscribe paradigm. Of course, newsgroups may be tied to WAIF, in that its messages may be published in WAIF.

15.3 Conclusion

We are currently refining the WAIF architecture and have started building some of its components. Ranking strategies will be key to the success of WAIF, so at present we are focusing on that question. It clearly will be important to create prototypes and actively use them, in order to drive this research. Other research issues we are tackling include: the scalability of routing and news items, the privacy of consumers, and a way for publishers to charge consumers for news items.

References

15.1 D.G. Andersen, H Balakrishnan, M.F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. of the Eighteenth ACM Symp. on Operating Systems Principles*, pages 131–145, Banff, Canada, October 2001.

15.2 G. Banavar, T.D. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based subscription systems. In *Proc. of the International Converence on Distributed Computing (ICDCS'99)*, Austin, TX, June 1999.

15.3 M. Claypool, D. Brown, Le P., and M. Waseda. Inferring user interest. *IEEE Internet Computing*, 5(6):32–39, Nov/Dec 2001.

15.4 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, August 2001.

15.5 F. Hattori, T. Ohguro, M Yokoo, Matsubara, and S. Yoshida. Socialware: Multiagent systems for supporting network communities. *CACM*, 42(3):55–61, March 1999.

15.6 H.D. Johansen and D. Johansen. Improving object search using hints, gossip, and supernodes. In *Proc. of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, Osaka, Japan, October 2002.
15.7 K.J. Lauvset, D. Johansen, and K. Marzullo. TOS: Kernel support for distributed systems management. In *Proc. of the Sixteenth ACM Symposium on Applied Computing*, Las Vegas, USA, March 2001.
15.8 M. Maybury. News on Demand. *CACM*, 43(2):33–34, February 2000.
15.9 M. Meng, C. Yu, and K.-L. Liu. Building efficient and effective metasearch engines. *ACM Computing Surveys*, 34(1):48–89, March 2002.
15.10 G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

16. The Importance of Aggregation*

Robbert van Renesse

Cornell University, Ithaca, NY 14853
rvr@cs.cornell.edu

16.1 Introduction

In this paper, we define *aggregation* as the ability to summarize information. In the area of sensor networks [16.2] it is also referred to as *data fusion*. It is the basis for scalability for many, if not all, large networking services. For example, address aggregation allows Internet routing to scale. Without it, routing tables would need a separate entry for each Internet address. Besides a problem of memory size, populating the tables would be all but impossible. DNS also makes extensive use of aggregation, allowing domain name to attribute mappings to be resolved in a small number of steps. Many basic distributed paradigms and consistency mechanisms are based on aggregation. For example, synchronization based on voting requires votes to be counted.

Aggregation is a standard service in databases. Using SQL queries, users can explicitly aggregate data in one or more tables in a variety of ways. With so many examples of aggregation in networked systems, it is surprising that no standard exists there as well. On the contrary, each networked service uses implicitly built-in mechanisms for doing aggregation. This results in a number of problems. First, these mechanisms often require a fair amount of configuration, which is not shared and needs to be done for each service separately. Second, the configuration is often quite static, and does not adapt well to dynamic growth or failures that occur in the network. Finally, the implementations are complex but the code cannot be reused.

There are only few general services available for aggregation in networked systems. "Mr. Fusion" [16.3] is a recent aggregation service intended for use with CORBA. Based on a voting framework [16.1], the Fusion Core collects ballots that are summarized when enough ballots have been collected. The output of the aggregation can be multi-dimensional, and represented as a hierarchical data cube. Cougar [16.4] is a sensor database system that supports SQL aggregation queries over the attributes of distributed sensors. Some other sensors network systems, like Directed Diffusion [16.5] have limited support for data aggregation as well.

We have developed an aggregation facility as well, called Astrolabe [16.6], for use by networked services. It resembles DNS in that it organizes hosts in a domain hierarchy and associates attributes with each domain. Different from DNS, the attributes of a non-leaf domain are generated by SQL aggregation queries over its child domains, and new attributes are easily introduced. Astrolabe can be customized for new applications by specifying additional aggregation queries. The implementation of Astrolabe is peer-to-peer, and does not involve any servers. To date, we have used Astrolabe to

* This research was funded in part by DARPA/AFRL-IFGA grant F30602-99-1-0532 and in part by the AFRL/Cornell Information Assurance Institute.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 87–92, 2003.

implement a scalable publish/subscribe facility and are currently developing a sensor network application with Astrolabe. Such a service may have many uses:

Leader Election. A group of cooperating processes often requires a coordinator to serialize certain decisions. Leader election algorithms are well known. Using an aggregation service, finding a deterministic leader is trivial, say by calculating the minimum (integer-valued) identifier. When this leader fails, the aggregation service should detect this and automatically report the next lowest identifier. Similarly, when a new member joins with a lower identifier, the aggregation service should report the new identifier.

Voting. In case a more decentralized synchronization is desired, voting algorithms are often used. These, too, can be implemented using aggregation. The aggregation service would simply count the total number of yes votes, the total number of no votes, and the total number of processes so the application can determine if there is a quorum. Weighted voting is a trivial generalization, and barrier synchronization can be done in a similar fashion.

Multicast Routing. In order for a message to be routed through a large network, it is necessary to know which parts of the network contain subscribers. An aggregation service should be able to aggregate data not only for the entire network, but also for parts of the network (domains). Now the aggregation service can count the number of subscribers in a domain, allowing a multicast service to route messages efficiently to only the necessary domains. As members join, leave, and/or crash, the aggregation service has to update these counts reliably.

Resource Location. Distributed applications often need to be able to locate the nearest-by resource, such as, for example, an object cache. If an aggregation service reports the number of such resources in each domain, a location-sensitive recursive search through a domain hierarchy will allow a process to find a nearby resource.

Load Balancing/Object Placement. A problem that faces many distributed applications is the placement of tasks or objects among the available hosts. Sometimes simple ad hoc solutions, such as round robin, are applied. Sometimes sophisticated placement and balancing algorithms are used, but they are tightly integrated with the application itself and not easily replaced with a new strategy. An aggregation service, by calculating such aggregates as the minimum, average, maximum, or total load in domains, can greatly simplify the problem of placement and separate the issue from the actual functionality of the application.

Error Recovery. When an error has occurred, say a host crash or a lost update message, it is often necessary to inspect the state of the application in order to detect and repair inconsistencies, or re-establish invariants. For example, when a host recovers, it may require a state snapshot from one or more other hosts. An aggregation service can help check invariants (akin to GPD), and provide the information necessary for the repair (e.g., the location of the most recently applied update).

16.2 Astrolabe

The goal of Astrolabe is to maintain a dynamically updated data structure reflecting the status and other information contributed by hosts in a potentially large system. For

reasons of scalability, the hosts are organized into a domain hierarchy, in which each participating machine is a leaf domain. The leaves may be visualized as tuples in a database: they correspond to a single host and have a set of attributes, which can be base values (integers, floating point numbers, etc) or an XML object. In contrast to these leaf attributes, which are directly updated by hosts, the attributes of an internal domain (those corresponding to "higher levels" in the hierarchy) are generated by summarizing attributes of its child domains.

The implementation of Astrolabe is entirely peer-to-peer: the system has no servers, nor are any of its agents configured to be responsible for any particular domain. Instead, each host runs an agent process that communicates with other agents through an epidemic protocol or *gossip* [16.7]. The data structures and protocols are designed such that the service scales well:

- The memory used by each host grows logarithmically with the membership size;
- The size of gossip messages grows logarithmically with the size of the membership;
- If configured well, the gossip load on network links grows logarithmically with the size of the membership, and is independent of the update rate;
- The *latency* grows logarithmically with the size of the membership. Latency is defined as the time it takes to take a snapshot of the entire membership and aggregate all its attributes;
- Astrolabe is tolerant of severe message loss and host failures, and deals with network partitioning and recovery.

In practice, even if the gossip load is low (Astrolabe agents are typically configured to gossip only once every few seconds), updates propagate very quickly, and are typically within a minute even for very large deployments [16.6]. In the description of the implementation of Astrolabe below, we omit all details except those that are necessary in order to understand the function of Astrolabe.

As there is exactly one Astrolabe agent for each leaf domain, we name Astrolabe agents by their corresponding domain names. Each Astrolabe agent maintains, for each domain that it is a member of, a relational table called the *domain table*. For example, the agent "/nl/amsterdam/vu" has domain tables for "/", "/nl", and "/nl/amsterdam". A domain table of a domain contains a row for each of its child domains, and a column for each attribute name. One of the rows in the table is the agent's *own* row, which corresponds to that child domain that the agent is a member of as well. Using a SQL aggregation function, a domain table may be aggregated by computing some form of summary of the contents to form a single row. The rows from the children of a domain are concatenated to form that domain's table in the agent, and this repeats to the root of the Astrolabe hierarchy.

Since multiple agents may be in the same domain, the corresponding domain table is replicated on all these agents. For example, both the agents "/nl/amsterdam/vu" and "/nl/utrecht/uu" maintain the "/" and "/nl" tables. As hosts can only update the attributes of their own leaf domains, there are no concurrent updates. However, such updates do affect the aggregate values, and so the updates need to be propagated so agents can recalculate the aggregates.

Separately for each domain, Astrolabe runs an epidemic protocol to propagate updates to all the agents contained in the domain. Not all agents in the domain are directly

involved in this protocol. Each agent in a domain calculates, using an aggregation function, a small set of representative agents for its domain (typically three).

The epidemic protocol of a domain is run among the representatives of its child domains. On a regular basis, say once a second, each agent X that is a representative for some child domain chooses another child domain at random, and then a representative agent Y within the chosen child domain, also at random. X sends the domain table to Y. Y merges this table with its own table, and sends the result back to X, so that X and Y now have the latest versions of the attributes known to both of them. X and Y repeat this for all ancestor domain tables up to the root domain table. As this protocol is run for all domains, eventually all updates spread through the entire system.

The rule by which such tables are merged is central to the behavior of the system as a whole. The basic idea is as follows. Y adopts into the merged table rows from X for child domains that were not in Y's original table, as well as rows for child domains that are more current than in Y's original table. To determine currency, agents timestamp rows each time they are updated by writing (in case of leaf domains) or by generation (in case of internal domains). Unfortunately, this requires all clocks to be synchronized which is, at least in today's Internet, far from being the case. Astrolabe therefore uses a mechanism somewhat similar to Lamport timestamps so no clock synchronization is required in practice.

The aggregation functions themselves are gossiped along with other updates, so that new aggregation function can be installed on the fly to customize Astrolabe for new applications.

16.3 Research Agenda

Above we have seen important uses for aggregation. Although much of the required functionality for aggregation is provided by Astrolabe and other aggregation services, all these services have shortcomings that suggest directions for future research. Below we list eight such issues, using Astrolabe an illustration of how such issues might appear:

Weak Consistency. Astrolabe samples the local state and lazily aggregates this information, but does not take consistent snapshots. There is obviously a trade-off between consistency and performance or scalability. Aggregation services should provide multiple levels of consistency so that an application can choose the appropriate trade-off.

Staleness. The aggregated information reflects data that may no longer be current. This is not the case when, say, counting votes (assuming processes vote only once), but it is the case when aggregating continuously changing attributes such as host load. Load balancing schemes that depend on such attributes may make incorrect decisions and be prone to oscillation. Research is necessary to improve accuracy of information.

High Latency. Astrolabe uses an epidemic protocol to disseminate updates. Although this protocol has excellent scaling properties, it takes significant time to evaluate an aggregation (measured in seconds), and thus the solution given above for voting may suffer from high latency. A hybrid solution of an epidemic protocol and a multicast protocol may improve latency while maintaining good scaling properties and robustness.

Inappropriate Hierarchy. The Astrolabe hierarchy is fixed by configuration, and this hierarchy is not always appropriate for an application at hand. Many applications will not even require a hierarchy. Future aggregation services will have to decide what kind of topologies are most appropriate for particular applications.

Limited Expressiveness. Astrolabe puts limitations on what aggregations can be expressed. This is partially due to SQL's limited expressiveness, but also due to the recursive nature in which aggregations are evaluated in the Astrolabe hierarchy. Also, in order to ensure that resource use is limited, the Astrolabe SQL engine does not support join operations. Future research could determine what expressiveness is possible while observing limited resources.

Limited Attributes. Astrolabe puts a limit on the size of the attribute set of a domain in order to make sure that its epidemic protocols scale well. Currently, an attribute set for a domain cannot be larger than about one kilobyte. This places significant limitations on how many aggregations can be executed simultaneously, and thus how many applications can use Astrolabe concurrently. Better compression in gossip exchanges may improve this situation. New protocols may be able to scale better in this dimension.

Weak Security. A scalable system has to be secure. The larger a system, the more likely that it will be subject to malicious attack. These attacks may include confusing the epidemic protocols, introducing bogus attributes, stealing information, etc. Although Astrolabe does use public key cryptography to avoid such problems, there are a number of weaknesses left. For example, in the voting algorithm described above, it does not prevent a host from voting twice. Also, the computational cost of public key cryptography is very high. New protocols may increase security while decreasing computational costs.

Unpredictable Performance. Although we have performed extensive simulation of large Astrolabe networks, Astrolabe still has to prove itself in a large deployment. Concern exists that as Astrolabe is scaled up, variance in its behavior will grow until the system becomes unstable and thus unusable. It may also place high loads on particular links in the network infrastructure. More detailed simulation may identify such problems before deployment, while more proactive protocols may avoid a meltdown.

Thus there is ample opportunity for research in this area, with practical implications for the development of distributed applications, particularly those of large scale.

References

16.1 Bakken, D., Zhan, Z., Jones, C., Karr, D.: Middleware support for voting and data fusion. In: Proc. of the International Conference on Dependable Systems and Networks, Göteborg, Sweden (2001)

16.2 Walz, E., Llinas, J.: Multisensor Data Fusion. Artech House, Boston (1990)

16.3 Franz, A., Mista, R., Bakken, D., Dyreson, C., Medidi, M.: Mr. Fusion: A programmable data fusion middleware subsystem with a tunable statistical profiling service. In: Proc. of the International Conference on Dependable Systems and Networks, Washington, D.C. (2002)

16.4 Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Proc. of the Second Int. Conf. on Mobile Data Management, Hong Kong (2001)

16.5 Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: A scalable and robust communication paradigm for sensor networks. In: Proc. of the Sixth Annual Int. Conference on Mobile Computing and Networks (MobiCom 2000), Boston, MA (2000)
16.6 Van Renesse, R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (2003) Accepted for publication.
16.7 Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proc. of the Sixth ACM Symp. on Principles of Distributed Computing, Vancouver, BC (1987) 1–12

17. Dynamic Lookup Networks*

Dahlia Malkhi

School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel 91904
dalia@cs.huji.ac.il

17.1 Introduction

This paper surveys exisiting work in distributed lookup services using dynamic shared hash tables (DHTs), with the goal of pointing out open issues and key challenges.

Discovering and locating resources in networked systems are quintessential problems of distributed computing. Their goal is to design a distributed lookup service that allows clients to contact the service from anywhere and find any resource in a reasonable amount of work. The solution paradigm of interest to us employs a *distributed hash table* (DHT), as employed, e.g., in [17.6, 17.11, 17.12, 17.14, 17.15, 17.18, 17.20]. The DHT paradigm uses a hash map to look up keys quickly. The hash table is distributed among a dynamic set of servers, each one holding a fraction of the map. An overlay network is employed in order to route each query to the server that holds the relevant information.

A central challenge is to construct an *overlay network* that routes lookup queries from any starting point to the server closest to the target. Much is known on routing networks [17.4, 17.17], including networks of low degree, small dilation and even congestion, such as Shuffle-Exchange, Butterfly, and Cube-Connected-Cycle. Deviating from these works, dynamic lookup networks additionally require the following properties: (i) the network size is unknown and dynamically changes as processes join and leave it, and (ii) there should be no centralized maintenance of the network. Thus, the problem is to dynamically construct a routing network that accommodates rapid changes in size and has no central management. In particular, when a server joins the network, it should be possible for it to become a participant in the routing network using a reasonable amount of communication with existing members.

Fortunately, we can borrow ideas from the way real-life networks and "small worlds" form, as observed by Kleinberg in [17.8] and Barriere et al. in [17.2]. In these works, each node of the network is assigned *long range contacts*, chosen appropriately so that a localized routing strategy produces short paths. The inspiration from [17.8] is that these few long range contacts should *not* be uniformly distributed, but have a bias toward closer points[1]. The manner in which long-range link are chosen precisely affects the resulting routing construction. Some of the existing solutions are: The network construction of [17.8] emulates a Cube-Connected-Cycle (with poly-logarithmic diameter). The Viceroy network of [17.11] offers a dynamic emulation of the Butterfly network, yielding a

* This work was supported in part by the Israeli Ministry of Science grant #1230-3-01.

[1] Note that Kleinberg's paper is descriptive, trying to explain how a social network allowing small hop routing may develop. We are in the process of developing a constructive algorithm providing a design to a network based directly on this approach.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 93–96, 2003.

network whose degree is constant yet its dilation is logarithmic. Chord [17.18] emulates a Hypercube (logarithmic degree, logarithmic dilation), as do Plaxton et al. in [17.13], Tapestry [17.20] and Pastry [17.15]. The CAN algorithm of [17.14] emulates a multi-dimensional torus, whose degree is a constant d determined as a system parameter, and whose dilation is $O(dn^{(1/d)})$.

The applicability of good solutions is vast. In order to share resources and access services over large, dynamic networks, users require means for locating them in an efficient manner. The fundamental service that is required is a lookup directory that maps names to values. The Domain Name System (DNS) is a known example of such a lookup service, but one that is static and furthermore, is tailored to the DNS hierarchical namespace and suffers from increased load and congestion at nodes close to the root of the DNS tree. In contrast, the DHT approach builds an on-the-fly dynamic lookup service, and paves the way to deployment in peer-to-peer networks, where the participating servers are particularly dynamic and no central control or information is maintained. Good solutions may find application in many peer-to-peer systems like music-sharing applications (e.g., Gnutella [17.5]), file sharing and anonymous publishing systems (e.g., Freenet [17.3]) and distributed engines that take advantage of CPU sharing (e.g., Seti@home [17.16]). In addition, several other settings make use of a logical communication infrastructure over an existing communication network. For example, Amir et al. use in [17.1] an overlay network for wide area group communication; and many ad hoc systems use overlay routing for regulating communication (see [17.19] for a good exposition).

17.2 Research Goals

The research and development of dynamic, scalable lookup networks is intensively going on in academic and industrial groups. At the Hebrew University, we have recently launched a project called Viceroy to build a dynamic, scalable lookup service. We list some of our goals here.

Given the decentralization and scalability of the problem at hand, randomization is a natural design choice. However, previous works derive their performance properties from the goodness of random choices made initially in the construction. We envision that in a long lived system, the quality of such initial random choices will necessarily degrade as the system evolves, many processes depart and so on. Therefore, one of our first goals is to re-balance the network dynamically against an adversary that adaptively impacts the randomness in the system. We need to further devise load shifting mechanisms that will not change the basic structure of the lookup network, and will be completely localized so as to preserve the decentralized nature of our approach. Moreover, we look for solutions that maintain these good measures over long range, and are not based on good randomization of initial choices made by processes when joining. Thus, we consider a powerful adversarial situation, where departures and joining are controlled by a malicious, partially adaptive adversary. This problem is further intensified by the need to deal with multiple concurrent changes.

Resilience is another important facet of a dynamic and scalable resource location. We envision a two-layer architecture to provide fault tolerance. At the bottom tier, each process is replicated in a small cluster, using any known clustering technology. Each

cluster then serves as a *super node* in the second tier, the routing network. A super node may gracefully join or leave the network but cannot fail. This is achieved by letting a cluster size vary between a low and a high water mark. When the number of participants in a cluster drops below the low threshold, it seeks to merge with another cluster, thus virtually leaving the network. Likewise, when a cluster size grows above the high mark, it splits and virtually create a new joining node. In order for a full cluster to fail, multiple simultaneous failures must occur, an event that we can rule out with proper tuning. Further issues of atomicity and fault tolerance are exposed in [17.10].

The next topic that concerns deployment of lookup methods in practice is adaptation and caching in order to cope with variable access load. In reality, there may be situations where the load becomes unbalanced due to transient hot spots. For example, in a music-sharing application, many users might access Madonna's recent album shortly after release. To accommodate such situations, the basic efficient design should be accompanied by a good caching strategy. A particular challenge is the lack of inherent hierarchy that would naturally support cache-flushing.

A final issue that deserves our immediate attention is locality of placement. In practice, it might be preferable to take into consideration proximity and network connectivity in the decision of how to place servers and interconnect them. Locality is considered in [17.13, 17.15], leading to encouraging results. These ideas and others need to be explored further for general lookup networks.

References

17.1 Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *International Conference on Dependable Systems and Networks* (FTCS-30, DCCA-8), New York, 2000.

17.2 L. Barriere, P. Fraigniaud, E. Kranakis and D. Krizanc. "Efficient routing in networks with long range contacts". In the *15th International Symposium on Distributed Computing (DISC '01)*, October 2001.

17.3 I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. "Freenet: A distributed anonymous information storage and retrieval system". In *Proceedings the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

17.4 T. H. Cormen, C. E. Leiserson and R. L. Rivest. "Introduction to algorithms". MIT Press, 1990.

17.5 `http://gnutella.wego.com`.

17.6 A. Fiat and J. Saia. "Censorship resistant peer-to-peer content addressable networks". Proceedings of the 13th ACM-SIAM Symp. on Discrete Algorithms, 2002.

17.7 D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web". In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, May 1997.

17.8 J. Kleinberg. "The small world phenomenon: An algorithmic perspective". Cornell Computer Science Technical Report 99-1776, October 1999. (Published, shorter version available as "Navigation in a Small World", *Nature* 406(2000).)

17.9 J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. "OceanStore: An architecture for global-scale persistent storage", Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

17.10 N. Lynch, D. Malkhi and D. Ratajczak. "Atomic data access in Content Addressable Networks: A position paper". Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02).

17.11 D. Malkhi, M. Naor and D. Ratajczak. "Viceroy: A scalable and dynamic emulation of the Butterfly". In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, July 2002. To appear.

17.12 G. Pandurangan, P. Raghavan and E. Upfal. "Building low-diameter p2p networks". *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, 2001.

17.13 C. Plaxton, R. Rajaram, and A. Richa. "Accessing nearby copies of replicated objects in a distributed environment". In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (SPAA 97), pages 311–320, June 1997.

17.14 S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. "A scalable content-addressable network". In *Proceedings of the ACM SIGCOMM 2001 Technical Conference*. August 2001.

17.15 A. Rowstron and P. Druschel. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329-350, November 2001.

17.16 `http://www.setiathome.ssl.berkeley.edu`.

17.17 H. J. Siegel. "Interconnection networks for SIMD machines". *Computer* 12(6):57–65, 1979.

17.18 I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications". In *Proceedings of the SIGCOMM 2001*, August 2001.

17.19 C. K. Toh. Ad Hoc mobile wireless networks: Protocols and systems. Prentice Hall, 2001.

17.20 B. Y. Zhao, J. D. Kubiatowicz and A. D. Joseph. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing". U. C. Berkeley Technical Report UCB/CSD-01-1141, April, 2001.

18. The Surprising Power of Epidemic Communication

Kenneth P. Birman*

Dept. of Computer Science
Cornell University; Ithaca, New York 14583

18.1 Introduction

The focus of this position paper is on the most appropriate form of middleware to offer in support of distributed system management, control, information sharing and multicast communication. Our premise is that technology has been deficient in all of these areas. If recent advances can be transitioned into general practice, this could enable a new generation of better distributed systems, with value in settings ranging from such "critical infrastructure" areas as air traffic control and control of the restructured electric power grid to emerging areas, such as large-scale sensor networks, data mining and data fusion.

The middleware domain of interest to us has witnessed some three decades of debate between distributed computing systems with strong properties (such as virtual synchrony, fault-tolerance, security, or guaranteed consistency) and those with weak properties (typified by web browsers, but extending into the broader area of web services and network applications built from remote procedure call and using timeout for failure detection). It seems fair to say that neither has been completely satisfactory, and commercial platforms have yet to include either kind of technology in a standard, widely available form.

Systems with stronger guarantees would be preferable to systems with weaker guarantees if the two categories were comparable in other dimensions (including performance, ease of use, programming support, configuration and management, runtime control, complexity of runtime environment, etc). However, the two classes differ in most of these respects, hence the question is more subtle. Systems offering stronger guarantees are very often slow, scale poorly, and require complex infrastructure. They have rarely been supported to the same degree as other technologies by commercial vendors. Programming tools are inferior or completely lacking, and integration with commercial platforms is poor. Underlying this dubious picture is a broader phenomenon: the market for strong solutions has been too small to be commercially exciting, thus the sort of funding that has gone into the most commonly available commercial solutions dwarfs that available to the developers of solutions having stronger properties. Several attempts to commercialize distributed computing technologies with strong properties failed.

---

* ken@cs.cornell.edu; 607-255-9199. This work was support in part by DARPA/AFRL grant number RADC F30602-99-1-0532 under ARPA Order J026.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 97–102, 2003.

The strong properties community typically defends its work by pointing to the anomalous behavior often experienced when using systems with weak guarantees: timeouts misinterpreted as evidence for failures, inconsistency that can arise when these occur, and the general shakiness of the resulting edifice. For situations in which lives might be at stake, large sums of money are at risk, or large numbers of users might be inconvenienced by outages, strong properties can give the operator confidence that a system will work as desired, when desired, and be available where needed, and make it clear what guarantees are needed from the hardware and network infrastructure.

As to the commercial failure of systems offering strong properties and the relative success of weak technologies, the collapse of eCommerce is a reminder that the user community can be fickle; indeed, the commercial community may already have abandoned weak solutions. If web services and similar technologies are to succeed, vendors will need to increase the degree of user confidence in the quality and reliability of their products. This, in turn, is likely to require that some system components implement strong properties.

In this debate, there is a tendency for the proponents of each approach to overlook the legitimate criticisms raised by its opponents. In their advocacy of weak properties, many technologists forget that for other purposes, such as operating system software or databases, when we cannot explain precisely why a thing works, it probably doesn't. Individuals who argue for a form of strong properties in other settings seemingly contradict themselves when it comes to distributed systems.

The converse is also true. Systems with strong properties have traditionally scaled poorly, even when the workload is held constant as the system size or network size is increased. As strong systems are scaled up, they become fragile and difficult to manage, and are more and more prone to disruptive outages triggered by relatively minor phenomena, such as transient overloads or bugs that cause small numbers of machines to freeze up while still responding to failure-detection probes. A deeper issue also looms: in most distributed systems technologies offering strong properties, there are forms of background overhead that grow as the system scales up (often, quadratically). The reality, then, is that for a new world to emerge in which strong properties become part of the "usual" approach we need to find new ways of building them, so that very large-scale deployments are easily constructed, easily installed, and work "even better" than small ones. Proponents of systems with strong properties have often ignored these concerns, even to the degree of writing papers that assert that such-and-such a solution is scalable, and yet ignore substantial costs because the events that trigger them these costs are "rare". Each time a belated analysis forces such a system to back away from its scalability claims, credibility is lost.

In effect, one might argue that those of us who promote strong properties have been inattentive to the genuinely high costs, genuinely poor scalability, and genuinely poor manageability of the kinds of systems we are advocating. True, we offer guarantees not otherwise available, but in doing so, we also abandon many kinds of guarantees that the user community now takes for granted.

18.2 The Promise of Peer-to-Peer Gossip

Recent years have witnessed the emergence of a new kind of distributed system in which probabilistic properties are achieved using various forms of randomization.

These solutions find their roots in a long history of work with probabilistic protocols, yet for the first time such methods have been applied successfully on a truly grand scale. Examples include Distributed Hash Tables (DHTs), which can be used to build large peer-to-peer indexing structures, peer-to-peer file systems and archival storage systems, gossip-based distributed database systems, Astrolabe [18.1] and Bimodal Multicast [18.2]. The latter are Cornell-developed systems, and combine elements of peer-to-peer design with gossip-based protocols.

The thesis of this white paper is that these new systems, and especially those using the mixture of approaches employed by the Cornell work, offer a spectrum of properties that overcome the inadequacies of traditional "strong property" solutions such as virtual synchrony or Consensus-based state machine replication, at least as such systems are normally implemented.

Bimodal Multicast and Astrolabe are resilient to disruption, route around transient network overloads, and have predictable normal-case and worst-case latencies. Both are "strongly convergent" and achieve a type of convergent consistency that can be quantified analytically and reasoned about mathematically. Overheads are fixed and low, exhibiting no growth even as the system is scaled up. While the guarantees of these systems are probabilistic and hence not as strong as the traditional kinds of strong properties identified earlier, they are strong enough to build upon directly and can even be used to implement stronger properties (with probability one) if so desired. Most important of all, although the code used to build these kinds of systems is rather simple and hence easily deployed (no special infrastructure is needed), they scale extremely well in all respects that matter and will achieve stronger guarantees as the size of a deployment rises.

Although a number of systems are now using probabilistic techniques of the sort employed by the Cornell work, not all systems within this class have been analyzed and shown to be scalable. Many peer-to-peer systems include infrequently used but costly algorithms for such purposes as rebuilding link tables or migrating copies of files or indexing information (for example, the file copying mechanisms in PAST [18.3] and CFS [18.4], or the link rebuilding protocol in the Tapestry system [18.5]). In earlier generations of systems with strong properties, scalability problems can often be traced to such mechanisms. Indeed, a common pattern can be identified: an "infrequently" used mechanism that is triggered by a rare event, but has cost linear in the size of the system, and in which the frequency of the rare event is also roughly linear in the size of the system, yielding a quadratic form of overhead. Eventually, these overheads rise to swamp the available network, and the system melts down. To argue that DHT or peer-to-peer solutions are scalable, we need to show that they are immune to such problems, yet this aspect has been largely overlooked, much as it was overlooked by many developers of systems with strong properties in the past.

On the other hand, it is certainly not the case that scalability is only possible in the manner of Astrolabe or Bimodal Multicast. It is quite possible that some of the DHT and peer-to-peer work cited does scale well, and that we thus have other solutions in hand and merely lack convincing demonstrations of their properties. Moreover, our own work has revealed that virtual synchrony can be implemented in a manner that would scale far better by borrowing some ideas from this new probabilistic world, but coupling them with other mechanisms that wait for safety to be achieved (with probability 1.0) before reporting events to the application [18.6]. We believe that scalability can be achieved in many ways, provided that the research community begins to attach appropriate importance to doing so, and to offering convincing proofs of

scalability in association with the other types of analysis routinely undertaken for their solutions.

18.3 Recommendations for Future Research and Conclusions

We arrive at a small set of recommendations to the research community, and then conclude with some suggestions for future research in the area.

1. For two decades, our focus has been on fault-tolerance and various forms of consistency that can be achieved in the presence of various forms of failure. The needs of potential users have shifted to emphasize scalability considerations, and we as a community need to follow the trend.
2. The key to achieving scalability in distributed systems revolves around attentiveness to the costs associated not just with normal-mode behavior, but also to the worst-case costs associated with infrequent disruptions.
3. Gossip-based techniques offer surprising scalability and robustness because information spreads through the system along a set of paths that grows exponentially over time. A disruption can delay the spread of data but cannot prevent it from reaching those participants that remain active and connected.
4. This same exponential curve makes gossip-based solutions easy to analyze, because it permits us to make simplifications for the purpose of modeling the system. In practice our simplifications may lead to unrealistically optimistic or pessimistic analysis, but in light of an exponential epidemic, unrealistic optimism or pessimism results in just minor errors – predictions that may be off by a round or two of communication. Thus, in distinction to the case for more classical protocols and distributed systems, mathematics turns out to be an effective and practical tool for reasoning about gossip-based software.
5. Peer-to-peer and DHT structures can be combined with gossip to implement robust systems that adapt rapidly as conditions change. In contrast, traditional peer-to-peer and DHT solutions will not discover that conditions have changed until an attempt is made to communicate with a machine and it is found to no longer be a system member. Gossip techniques allow a system to react and repair itself within seconds, at which point the peer-to-peer or DHT data structure can be trusted to be largely intact and correct. This, in turn, permits a much smarter style of planning within the DHT or peer-to-peer system.
6. When desired, stronger properties can be superimposed on these kinds of scalable primitives, for example using Gupta's methodology [18.6].

We conclude with some thoughts concerning directions for further research:

1. *Build some ambitious real-world systems for large-scale use.* The success we and others have had with gossip-based solutions suggests that it is time to build a "full scale" distributed systems infrastructure capable of supporting commercially interesting applications such as web-services, but offering superior manageability, performance and scalability. To convince the networking community that these techniques really work, are easy to use, and are easy to understand, we need to show potential practitioners examples of real systems which they can play with, evaluate, perhaps extend or imitate. Little of this important practical work has been completed.

2. *Learn more about the properties of large-scale communication environments.* A challenge that systems like Astrolabe must overcome is the need to communicate despite potentially high churn rates, firewalls, and asymmetric connectivity. As a community, we need to better understand the options and fundamental limitations associated with large-scale environments and find ways of encapsulating "best of breed" solutions in standard packages.
3. *Learn to talk about properties of systems "in the large."* We need to begin to think about the properties of really large systems, and to find ways of doing so that let us abstract beyond the behaviors of individual components and to talk about large-scale system behaviors spanning thousands or even tens of thousands of nodes.
4. *Pursue metaphors from emerging fields to which our ideas might be applicable.* It is intuitively appealing to speculate that many kinds of physical systems, notably biological ones, may operate along principles analogous to the ones exploited in these new kinds of scalable systems. For example, communities of insects, or cells in the body, signal with probabilistic mechanisms (proteins of various kinds), and are extremely robust to disruption. Elucidating such a connection would let the distributed systems community demonstrate its relevance to the rapidly growing biology community and might infuse our area with a new kind of very exciting application to investigate.
5. *Seize the high ground by demonstrating success in emerging large-scale networks used for narrow purposes, such as sensor grids.* The thinking here is that we need to look for completely new kinds of applications on which we can demonstrate the value of these techniques, ideally by targeting opportunities outside of the traditional Internet domain. One that seems especially interesting involves sensor networks. Gupta, elsewhere in this volume, speaks to this issue at some length, hence I limit myself to the observation that compelling success in real systems with large numbers of real computers would quickly stimulate a significant wave of research and reveal great numbers of new applications. Moreover, the Internet is a relatively mature world and it may not be realistic to try and change it; this is not true for many emerging areas, such as sensor networks.
6. *Revisit classical distributed systems problems from a scalability perspective.* We need to remove our rosy glasses and revisit many of the classical problems from a scalability perspective. How scalable are the traditional solutions to problems such as Consensus or Byzantine Agreement? Are there intrinsic scalability limitations to some problems, or perhaps even a "complexity hierarchy" for scalability? In particular, the costs of infrequent system mechanisms need to be studied more closely; a great many distributed systems have some mechanism that looks costly but runs rarely. Traditionally, this is cited as a reason to ignore the associated costs. But "rarely", one finds, is a relative term; in a very large setting, rare events may occur with unexpectedly high absolute frequency. The costs of these rare events may thus represent a huge load and even rise to limit scalability. Our community has little chance of winning the debate with those who ignore properties if we, in our own turn, ignore some of the most serious costs associated with the solutions we promote!

The surprising power and scalability of gossip communication could open the door to a new wave of distributed systems combining strong properties with the type of robustness and predictability which has been lacking in classical systems having strong properties. Meanwhile, the commercial community is starting to demand more of the

weaker web-based solutions on which they have come to depend. Our ability to offer strong properties in appealing packages could spur a new generation of distributed systems applications in which for the first time, the community associated with strong solutions would emerge as full-fledged participants.

References

18.1 Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. Robbert van Renesse, and Kenneth Birman. Submitted to ACM TOCS, November 2001

18.2 Bimodal Multicast. Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu and Yaron Minsky. ACM Transactions on Computer Systems, Vol. 17, No. 2, pp 41-88, May, 1999.

18.3 Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. Antony Rowstron (Microsoft Research), Peter Druschel (Rice University)

18.4 Wide-Area Cooperative Storage with CFS. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris (MIT), Ion Stoica (UC Berkeley)

18.5 A Scalable Content-Addressable Network. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. Proc. ACM SIGCOMM, San Diego, CA, August 2001.

18.6 Fighting Fire with Fire: Using Randomized Gossip to Combat Stochastic Reliability Limits. Indranil Gupta, Ken Birman, Robbert van Renesse. *Quality and Reliability Engineering International*, 18: 165-184 (Wiley; 2002)

19. Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks

Miguel Castro[1], Peter Druschel[2], Y. Charlie Hu[3], and Antony Rowstron[1]

[1] Microsoft Research, 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK
[2] Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA
[3] Purdue University, 1285 EE Building, West Lafayette, IN 47907, USA

19.1 Introduction

Structured peer-to-peer (p2p) overlay networks like CAN, Chord, Pastry and Tapestry [19.14, 19.20, 19.17, 19.22] provide a self-organizing substrate for large-scale p2p applications. They can implement a scalable, fault-tolerant distributed hash table (DHT), in which any item can be located within a small number of routing hops using a small per-node routing table. These systems have been used in a variety of distributed applications, including distributed stores [19.7, 19.18, 19.10, 19.6], event notification, and content distribution [19.23, 19.5, 19.9, 19.4].

It is critical for overlay routing to be aware of the network topology. Otherwise, each routing hop takes a message to a node with a random location in the Internet, which results in high lookup delays and unnecessary wide-area network traffic. While there are algorithmic similarities among each of these systems, an important distinction lies in the approach they take to topology-aware routing. We present a brief comparison of the different approaches that have been proposed [19.16], and give an outlook on future research directions.

19.2 State of the Art

In this section, we outline the state of the art in topology-aware routing for structured p2p overlays. We begin with a brief description of four protocols: Pastry, Tapestry, CAN and Chord. In all protocols, nodes and objects are assigned random identifiers (called *nodeIds* and *keys*, respectively) from a large, sparse id space. A *route* primitive forwards a message to the live node that is closest in the id space to the message's key.

In **Pastry**, keys and nodeIds are 128 bits in length and can be thought of as a sequence of digits in base 16. A node's routing table has about $log_{16}N$ rows and 16 columns (N is the number of nodes in the overlay). The entries in row n of the routing table refer to nodes whose nodeIds share the first n digits with the present node's nodeId. The $(n+1)$th nodeId digit of a node in column m of row n equals m. The column in row n corresponding to the value of the $(n+1)$th digit of the local node's nodeId remains empty. At each routing step in Pastry, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 103–107, 2003.

is forwarded to a node whose nodeId shares a prefix with the key as long as the current node but is numerically closer to the key than the present node's id. Each Pastry node maintains a set of neighboring nodes in the nodeId space (leaf set), both to locate the destination in the final routing hop, and to store replicas of data items for fault tolerance. The expected number of routing hops is less than $log_{16}N$.

Tapestry is very similar to Pastry but differs in its approach to mapping keys to nodes in the sparsely populated id space, and in how it manages replication. In Tapestry, there is no leaf set and neighboring nodes in the namespace are not aware of each other. When a node's routing table does not have an entry for a node that matches a key's nth digit, the message is forwarded to the node in the routing table with the next higher value in the nth digit modulo 2^b. This procedure, called *surrogate routing*, maps keys to a unique live node if the node routing tables are consistent. For fault tolerance, Tapestry inserts replicas of data items using different keys. The expected number of routing hops is $log_{16}N$.

Chord uses a circular 160 bit id space. Unlike Pastry, Chord forwards messages only clockwise in the circular id space. Instead of the prefix-based routing table in Pastry, Chord nodes maintain a *finger table*, consisting of nodeIds and IP addresses of up to 160 other live nodes. The ith entry in the finger table of the node with nodeId n refers to the live node with the smallest nodeId clockwise from $n + 2^{i-1}$. The first entry points to n's successor, and subsequent entries refer to nodes at repeatedly doubling distances from n. Each node also maintains pointers to its predecessor and to its k successors in the id space (the successor list). Similar to Pastry's leaf set, this successor list is used to replicate objects for fault tolerance. The expected number of routing hops in Chord is $\frac{1}{2}log_2N$.

CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. The entries in a node's routing table refer to its neighbors in the d-dimensional space. Unlike Pastry, Tapestry and Chord, CAN's routing table does not grow with the network size but the number of routing hops grows faster than $logN$ in this case, namely $O(dN^{1/d})$.

Next, we describe and compare the three approaches to topology-aware routing in structured overlay networks that have been proposed: *proximity routing*, *topology-based nodeId assignment*, and *proximity neighbor selection* [19.16].

Proximity routing: With proximity routing, the overlay is constructed without regard for the physical network topology. But when routing a message, there are potentially several nodes in the routing table closer to the message's key in the id space. The idea is to select, among this set of possible next hops, the one that is closest in the physical network or one that represents a good compromise between progress in the id space and proximity. With k alternative hops in each step, the approach can reduce the expected delay in each hop from the average delay between two nodes to the average delay to the nearest among k nodes with random locations in the network. The benefits are proportional to the value of k. Increasing k requires a larger routing table with correspondingly higher overheads for maintaining the overlay. Moreover, choosing the lowest delay hop greedily may lead to an increase in the total number of hops taken. While proximity routing can yield significant improvements over a system with no topology-aware routing, its cost/benefit

ratio falls short of the other two approaches. The technique has been used in CAN and Chord [19.14, 19.7].

Topology-based nodeId assignment: Topology-based nodeId assignment attempts to map the overlay's logical id space onto the physical network such that neighboring nodes in the id space are close in the physical network. A version of this technique was implemented in CAN [19.14, 19.15]. It achieves a delay stretch (i.e., relative delay to IP) of two or less. However, the approach has several drawbacks. First, it destroys the uniform population of the id space, which causes load balancing problems in the overlay. Second, the approach does not work well in overlays that use a one-dimensional id space (Chord, Tapestry, Pastry) because the mapping is overly constrained. Lastly, neighboring nodes in the id space are more likely to suffer correlated failures, which can have implications for robustness and security in protocols like Chord and Pastry that replicate objects on neighbors in the id space.

Proximity neighbour selection: Like the previous technique, proximity neighbor selection constructs a topology-aware overlay. But instead of biasing the nodeId assignment, the idea is to choose routing table entries to refer to the topologically closest node among all nodes with nodeId in the desired portion of the id space. The success of this technique depends on the degree of freedom an overlay protocol has in choosing routing table entries without affecting the expected number of routing hops. In prefix-based protocols like Tapestry and Pastry, the upper levels of the routing table allow great freedom in this choice, with lower levels having exponentially less choice. As a result, the expected delay of the first hop is very low and it increases exponentially with each hop. Therefore, the delay of the final hop dominates. This leads to low delay stretch, good load balancing, and local route convergence [19.3]. A limitation of this technique is that it does not work for overlay protocols like CAN and Chord, which require that routing table entries refer to specific points in the id space.

Discussion: Proximity routing is the most light-weight technique because it does not construct a topology-aware overlay. But, its performance is limited because it reduces the expected per-hop delay to the expected delay to the nearest among a (usually small) number k of nodes with random locations in the network. Increasing k also increases the overhead of maintaining the overlay. With topology-aware nodeId assignment, the expected per-hop delay can be as low as the average delay among neighboring overlay nodes in the network. However, the technique introduces load imbalance and requires a high-dimensional id space to be effective.

Proximity-neighbor selection can be viewed as a compromise that preserves the load balance and robustness afforded by a random nodeId assignment but still achieves a small constant delay stretch. In a full-length version of this paper [19.3], we show that: proximity neighbor selection can be implemented in Pastry and Tapestry with low overhead; it achieves comparable delay stretch to topology-based nodeId assignment without sacrificing load balancing or robustness; and it has additional route convergence properties that facilitate efficient caching and group communication in the overlay. Moreover, we confirm these results via simulations on two large-scale Internet topology models, and via measurements in a small Internet testbed.

Experience shows that topology-aware routing is critical for application performance. Without it, each routing hop incurs wide-area network traffic and has an average delay equal to the average delay between nodes in the Internet.

19.3 Future Research Directions

Large-scale simulations using Internet topology models and small-scale Internet testbed experiments show the effectiveness of topology-aware routing in p2p overlays. However, given the complexity of the Internet, larger-scale experiments on the live Internet are necessary to confirm these results and refine the algorithms. In the near term, we expect the PlanetLab effort [19.1] to provide a medium-scale testbed for this purpose. Longer term, it will be necessary to deploy an application that is able to attract a very large user community. It is currently an open question whether proximity neighbor selection can be applied to CAN and Chord, or if equally effective techniques exist that work in CAN and Chord.

Topology-aware routing is currently able to achieve an average delay stretch of 1.4 to 2.2, depending on the Internet topology model. A question is whether this figure can be improved further in a cost-effective manner. One possible approach is to directly exploit Internet topology information from IP or BGP routing tables, or via limited manual configuration. The Brocade effort [19.21] adds hierarchy to Tapestry, where manually configured supernodes route messages among intra-AS Tapestry overlays. However, the performance results are not significantly better than those reported with a flat Pastry overlay [19.3], and supernodes make self-organization, load balance, and security more difficult.

Beyond topology-aware routing, many significant research challenges remain in the area of p2p overlays. In many environments, p2p overlays and the applications built upon them must be tolerant of participating nodes that act maliciously. Initial work has been done in securing the routing and lookup functions [19.19, 19.12, 19.2], and securing application data and services [19.10, 19.13]. However, more effective defenses are needed against the Sybil attack [19.8], where an attacker joins the overlay under many different identities in order to control a fraction of the overlay sufficiently large to compromise security. Moreover, effective solutions must be found to ensure that participating nodes contribute resources proportional to the benefit they derive from the system.

It is also an open question whether structured p2p overlays are suitable in a highly dynamic environment where the set of participating nodes or the underlying physical network topology changes very rapidly. In these environment, unstructured p2p overlays that rely on random search to locate objects may have an advantage [19.11]. One could envision hybrid overlays, where short-term or mobile participants join an unstructured overlay that is connected to a structured overlay consisting of more stable, well-connected participants.

Finally, p2p overlays have been shown to support useful applications like large-scale network storage, event notification, and content distribution. The hope is that they will ultimately prove to enable a larger class of novel, yet to be discovered applications.

References

19.1 Planetlab. http://www.planet-lab.org.

19.2 M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. OSDI'02*, Dec. 2002.

19.3 M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks, 2002. Technical report MSR-TR-2002-82.

19.4 M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: A bandwidth-intensive content streaming system, 2002. Submitted for publication.

19.5 M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

19.6 L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. OSDI'02*, Dec. 2002.

19.7 F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, Oct. 2001.

19.8 J. Douceur. The Sybil attack. In *Proc. IPTPS'02*, Mar. 2002.

19.9 S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. PODC'02*, July 2002.

19.10 J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Nov. 2000.

19.11 Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make Gnutella scalable? In *Proc. IPTPS'02*, Mar. 2002.

19.12 N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. IPTPS'02*, Mar. 2002.

19.13 A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI'02*, Dec. 2002.

19.14 S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. SIGCOMM'01*, Aug. 2001.

19.15 S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. INFOCOM'02*, 2002.

19.16 S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *Proc. IPTPS'02*, Mar. 2002.

19.17 A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware'01*, 2001.

19.18 A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, Oct. 2001.

19.19 E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. IPTPS'02*, Mar. 2002.

19.20 I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM'01*, 2001.

19.21 B. Zhao, Y. Duan, L. Huang, A. Joseph, and J. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proc. IPTPS'02*, Mar. 2002.

19.22 B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, Apr. 2001.

19.23 S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proc. NOSSDAV'01*, June 2001.

20. Uncertainty and Predictability: Can They Be Reconciled?*

Paulo Veríssimo

Univ. of Lisboa, Faculty of Sciences
Lisboa - Portugal
pjv@di.fc.ul.pt
http://www.navigators.di.fc.ul.pt

20.1 Introduction

We are faced today with the confluence of antagonistic aims, when designing and deploying distributed systems. On one hand, our applications have to achieve timeliness goals, dictated both by QoS expectations with regard to on-line services (e.g. time-bounded transactions), and by technical issues of real-time nature involved in the deployment of certain services (e.g., multimedia rendering). On the other hand, the open and large-scale environments where applications and users execute and evolve exhibit uncertain timeliness or synchrony. Likewise, services, despite their sometimes critical nature (not only money-critical, but also privacy- or even safety-critical), are more often deployed on-line or through open networks. It is required that they be resilient to intrusions, despite the elusiveness of attacks they are subject to, and the pervasiveness and subtelty of vulnerabilities in the relevant systems. In other words, the environment in which these services have to operate exhibits uncertain behavior: we cannot predict all possible present and future attacks; we cannot diagnose all vulnerabilities.

In the previous paragraph, we essentially talked about *uncertainty*, the grand challenge faced by distributed system researchers and designers. When talking about uncertainty, 'impossibility' and 'probability' are words that come to mind. Literature has relevant examples on being pessimistic and accepting uncertainty, showing impossibility results [20.1], or producing solutions that are uncertain, albeit quantifiably uncertain [20.2, 20.3]. Other works have methodically studied what can be done when the system is incrementally less uncertain [20.4]. Alternatively, other approaches are more optimistic, assuming that the system has periods of determinism, alternating with uncertainty, and try to identify and successfully explore those (sometimes scarce) periods, to perform useful tasks [20.5, 20.6].

Nevertheless, a designer does not make strong assumptions about synchrony, or security, or structure, just for the sake of it. They are made because they provide guarantees (read: combinations of timeliness and reliability) not enjoyed by other alternatives, in essence, a degree of *predictability* about system attributes. Recently, we observe works which make increasingly strong assumptions about the environment, to get correspondingly higher guarantees. For example, arguing about the advantage of having perfect

* Work partially supported by the EC, through proj. IST-1999-11583 (MAFTIA), IST-FET-2000-26031 (CORTEX), and FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE) and proj. POSI/1999/CHS/33996 (DEFEATS).

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 108–113, 2003.

failure detectors [20.7]. Such failure detectors, however, cannot be implemented on environments with uncertain synchrony, which have been the workhorse of all past work on failure detectors. This status quo might be extended to security and survivability: giving hard guarantees on security of services often requires strong properties of the underlying environments.

On the more practical side, designers of protocols and systems like Squid, Akamai, Inktomi, AOL, etc., confronted for example with the uncertainty of the Internet, have been providing ad hoc (but normally extremely effective) mechanisms, such as warm cache hierarchies or prioritary execution of special tasks, for performance, or dedicated channels (e.g., physical or overlay networks), both for performance and security. However, short of a systemic approach to the problem, guarantees are essentially statistical. This would be enough for those applications, because service provision has largely been based on average performance and loose contractual guarantees, but has not solved the predictability problem, for example: the latency of individual transactions, or multimedia frames, or the intrusion tolerance of a TTP server.

In this paper, we discuss a novel design philosophy for distributed systems with uncertain or unknown attributes, such as synchrony, or failure modes. This philosophy is based on the existence of architectural constructs with privileged properties which endow systems with the capability of evading the uncertainty of the environment for certain crucial steps of their operation where predictability is required. It may open new research avenues allowing to reconcile uncertainty with predictability.

20.2 The Wormhole Metaphore

So, what system model and design principles will allow us to meet the grand challenge posed by uncertainty? We propose a few guiding principles: assume that uncertainty is not ubiquitous and is not everlasting – the system has parts more predictable than others and tends to stabilize; be proactive in achieving predictability – make it happen at the right time, right place.

In what follows, we wish to share with the reader one possible research track that follows the above-mentioned guidelines. We introduce it with the help of a metaphor. In the universe, speed of light is the fastest that can be attained, which would make it impractical to travel to or communicate with remote parts of the universe. However, a theory argues that one could take shortcuts, through, say, another dimension, and re-emerge safely at the desired point, apparently much faster than what is allowed by the speed of light. Those shortcuts received the inspiring name of *wormholes*.

Let us move from metaphor to reality. Assume that we can construct a *distributed system with wormholes*. There would be the 'payload system' where applications execute, i.e. the "normal" system with several hosts interconnected by the 'payload network', the usual Internet/Intranet. Then, there would be a small alternative subsystem whose behavior would be predictable: the 'wormhole subsystem'. This subsystem would be accessed from the payload through 'wormhole gateways', devices local to hosts. In practical terms, the wormhole is an artifact to be used only when needed, and its services supposedly implement functionality hard to achieve on the payload system, which in turn should run most of the computing and communications activity.

So, in conclusion, the key characteristic of the architecture of a system with wormholes consists in assuming that, no matter how uncertain the system and its behavior may be, there will be a subsystem which has 'good', well-defined properties. This subsystem is small and simple, so that its construction with trustworthy behavior is feasible. Note that whilst the most fascinating and powerful incarnation of a wormhole would be distributed, we can envisage simpler versions, with local (non-networked) wormholes, which still provide very useful support (e.g., local security or timeliness functions).

20.3 Is It Possible to Travel through Wormholes?

This metaphoric question translates into two practical ones:

Is it feasible to construct systems such as postulated above?

Are systems with wormholes of any real use?

As to the construction, observe Figure 20.1, where we suggest two possible implementations, one for small-area settings, another for wide-area ones. The first, in Figure 20.1(a), shows a mission critical web server replicated inside a facility, such as a Command, Control and Communications Center. The local wormholes can be implemented by some sort of appliance board with a private network adapter. The wormhole interconnection inside a facility as in the example can be implemented by a private LAN interconnecting the wormhole adapters. Figure 20.1(b) shows an authentication service distributed over a wide area. The wormhole interconnection in this case has to be highly secure, deterministic and work in wide-area. A feasible example of the above is depicted in the figure: wide-area Virtual Private Networks (VPN), constructed over ISDN or 3G-UMTS.

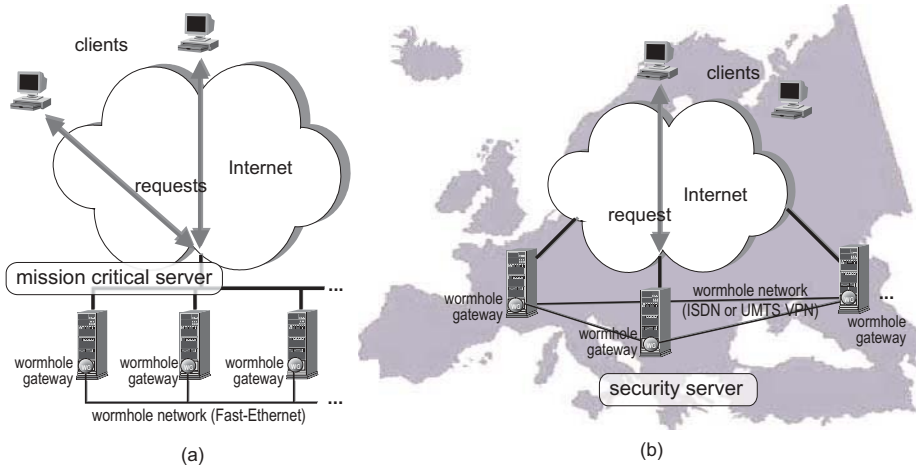

Fig. 20.1. Examples of real systems with wormholes: (a) replicated mission critical web server; (b) distributed security server

Figure 20.2 shows less trivial examples of the utility of wormholes. Figure 20.2(a) suggests the use of wormholes to enhance the control of overlay networks (ON). In fact, wormholes should not be confused with ONs, but the latter could be built on the wormhole concept, to strengthen its predictability, as the figure suggests. The payload channel would be implemented with the normal ON network support, whilst a control channel with differentiated properties would ensure predictable (timely and secure) management and reconfiguration of the ON. Figure 20.2(b) depicts a situation where a team of mobile units is moving and occasionally hooking to wired base stations. Imagine for example a platoon of cars or a team of robots. It is important that cars/robots keep timely and coherent synchronization, despite any glitches in their wireless payload networking support. The wormhole would help achieve that objective.

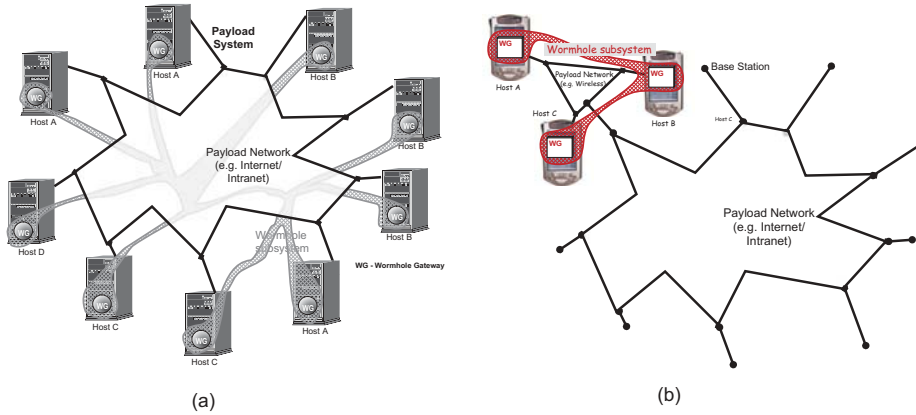

Fig. 20.2. Examples of real systems with wormholes: (a) Enhancing overlay network control; (b) Supporting mobile teams

As to the usefulness of wormholes, consider one instance of the global problem we stated: *performing timely actions in the presence of uncertain timeliness*. In one of our experiments, we have prototyped a specific kind of wormhole subsystem for achieving predictable behavior in systems of uncertain synchrony, anywhere in the spectrum from asynchronous to synchronous [20.8]. We called it the *Timely Computing Base*. In [20.9] we present a formal embodiment of the model. The Timely Computing Base can for example be used to build perfect failure detectors and thus support asynchronous algorithms running on the payload system and relying on the former detectors [20.7].

In the malicious failure domain, the potential for intrusion tolerance can be drastically augmented by using wormholes, for two reasons: they implement some degree of distributed trust for low-level operations, acting as a *distributed security kernel*; they give more room for the uncertainty of malicious behavior in the payload system. In a second experiment, we showed a way of *performing trusted actions in the presence of uncertain attacks and vulnerabilities*. We devised a set of new functions resilient to malicious faults for this new wormhole, calling it Trusted Timely Computing Base [20.10].

20.4 Conclusions and Future Work

We have given a unifying perspective to a recent research effort around a novel approach to distributed systems design. This work was triggered by the intutition of the need to handle uncertainty but still be able to provide predictable behavior. The approach is well-founded, since by introducing the necessary architectural devices – *wormholes* – the desired behavior is enforced, rather than assumed. The example implementations shown address scale from a limited perspective: wormholes can be deployed in large-scale in terms of geographical scope – relatively few but very far apart – and communities of few can serve collections of very many – client-server. However, as a concept, wormholes need not be limited by scale, if the following challenge is solved: how to predictability communicate from many to many, using scarce resources.

Authors are looking for better algorithms, more efficient, faster or even timed. Most of these works require constructs that prefigure the concept of wormhole [20.7, 20.11]. Moreover, a recent paper has shown that "there is no free lunch" [20.12]: if we wish to do really useful things, in the presence of an unbounded number of failures (or uncertain, for the matter), we have to make correspondingly strong assumptions about our environment. In the cited paper, the authors argue about the need for perfect failure detectors (and no weaker). Such failure detectors could be easily implemented on environments as postulated in this paper. More recently [20.13], authors are proposing to study efficient schemes for using wormholes – since they are a scarce resource.

It is our intention that the ideas presented here are used as a foundation to build systems meeting the present and future challenges concerning uncertainty. As a matter of fact, most of our recent work was focused on consolidating this foundation, and producing the prototypes we have shared with the community (**http://www.navigators.di.fc.ul.pt/software/tcb**). Challenging payload protocols can be built using wormholes, and we are just starting to discover that [20.14]. We plan on further exploiting the power of wormholes in areas as different as: intrusion-tolerant consensus and interactive consistency; timed agreement and ordering primitives; event-based communication for cooperative and embedded mobile systems.

References

20.1 Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
20.2 Rabin, M.O.: Randomized Byzantine Generals. In: Procs. of the 24th Annual IEEE Symposium on Foundations of Computer Science. (1983) 403–409
20.3 Cristian, F.: Probabilistic Clock Synchronization. Distributed Computing, Springer Verlag **1989** (1989)
20.4 Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
20.5 Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
20.6 Christian, F., Fetzer, C.: The timed asynchronous system model. In: Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing. (1998) 140–149

20.7 Helary, J., Hurfin, M., Mostefaoui, A., Raynal, M., F., T.: Computing global functions on asynchronous distributed systems with perfect failure detectors. IEEE Transactions on Parallel and Distributed Systems **11** (2000)
20.8 Veríssimo, P., Casimiro, A., Fetzer, C.: The timely computing base: Timely actions in the presence of uncertain timeliness. In: Procs. of the Int'l Conference on Dependable Systems and Networks, New York City, USA (2000) 533–542
20.9 Veríssimo, P., Casimiro, A.: The timely computing base model and architecture. IEEE Transactions on Computers, Special Issue on Asynchronous Real-Time Distributed Systems (2002)
20.10 Correia, M., Veríssimo, P., Neves, N.F.: The design of a COTS real-time distributed security kernel. In: Proc. of the Fourth European Dependable Computing Conference, Toulouse, France (2002)
20.11 M. Aguilera, G.L.L., Toueg, S.: On the impact of fast failure detectors on real-time fault-tolerant systems. In: Proc. of DISC 2002. (2002)
20.12 Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A realistic look at failure detectors. In: Proceedings of the International Conference on Dependable Systems and Networks, Washington, USA (2002) 213–222
20.13 R. Friedman, A. Moustefaoui, S.R., Raynal, M.: Error correcting codes: A future direction to solve distributed agreement problems? In: International Workshop on Future Directions of Distributed Computing, FuDiCo. (2002)
20.14 Correia, M., Lung, L.C., Neves, N.F., Veríssimo, P.: Efficient byzantine-resilient reliable multicast on a hybrid failure model. In: Proc. of the 21st Symposium on Reliable Distributed Systems, Suita, Japan (2002)

21. Fuzzy Group Membership

Roy Friedman

Department of Computer Science, The Technion, Haifa 32000, Israel
roy@cs.technion.ac.il

21.1 Background

Hand-held and palm-held computing devices are becoming increasingly strong. For example, today's high-end devices have the same (theoretical) computing power and memory capacity as high-end desktops of merely five years ago. Judging from the development of laptops, this trend is likely to continue at an even faster paste in the next few years. These powerful computing devices come equipped with commodity operating systems, such as Linux and Windows CE, which will progressively resemble their desktop OS counterparts as the devices become even more powerful. At the same time, hand-held and palm-held computing devices are being equipped with wireless and cellular communication capabilities, whose bandwidth is gradually approaching standard LAN speeds. Of particular interest to us is wireless communication, due to its hardware broadcast nature, as well as its relative high bandwidth, low cost, and low power consumption when compared to cellular communication.

These developments open the way for a whole new set of mobile applications that operate in *ad-hoc networks*, or in other words, networks of devices that are formed in an ad-hoc manner, and utilize them for combined efficiency. Examples of such applications include, e.g., *multi-party ad-hoc transactions*, *taking coordinated actions in ad-hoc networks*, *bandwidth enhancements for cellular communication*, and *collaborative peer-to-peer caching in ad-hoc networks*. More specifically, we can envision multiple users/devices that meet in an ad-hoc manner and may wish to conduct a multi-party transaction. Given the ad-hoc nature of the setting, no fixed infrastructure can be assumed, and in particular, one cannot rely on the existence of servers to drive the interactions and/or record the results of the transactions. Thus, participants must discover each other independently, using their available wireless communication. They must also reach an agreement on what is being exchanged, what are the terms, and which parties must provide what services to which other parties. Such decisions must be binding, and have atomic transactions semantics. For the reasons mentioned above, it is preferable to conduct the communication over wireless communication rather than cellular, to the degree possible. Also, note that such an ad-hoc system may involve multiple transactions, and perhaps some negotiation or auction before each transaction, which implies that the infrastructure should deliver high-throughout. For the sake of brevity, we do not elaborate here on the other three examples, but any expert in the area should be able to fill in the details.

The question we are exploring is how to construct middleware that best support such applications in the described setting. Such middleware should provide a good tradeoff between performance and ease of programming.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 114–118, 2003.

21.2 Fuzzy Membership

Many distributed applications rely on the notion of *membership*. That is, each node maintains an estimation of which other nodes participate in the computation and are alive and connected. Different systems may allow different levels of consistency between the membership estimations held by different nodes, and in some cases, each node only holds partial membership information. However, the accuracy of the membership estimation has a significant impact on the performance of the system.

In particular, if a node is considered dead too soon, the system must go through a costly reconfiguration phase just to repeat it when the mistake is discovered. On the other hand, if a node is faulty or disconnected but it takes a long time to detect it, the system operates in sub-optimal manner until the failure is detected and compensated. For example, a transaction system might be forced to abort all transactions [21.16, 21.23]; a distributed consensus protocol may need to go through a longer recovery phase and take longer to decide [21.8, 21.14, 21.21]; in group communication [21.2, 21.4, 21.6, 21.11, 21.13, 21.18, 21.24], this may block the flow control mechanism and/or create very large in-memory buffers since messages will not stabilize; in other dissemination systems [21.5, 21.19, 21.26], this may result in longer delivery time, and in some cases, a loss of some messages. These issues become more acute in wireless environments in which short-lived disconnections are common.

We propose to address these problems by introducing the notion of *fuzzy membership*. That is, instead of having a binary membership, i.e., either the member is considered alive and connected or is presumed dead, each member is associated with a *fuzziness level*. Moreover, when a member sends heartbeat messages as expected, failure to send other protocol messages in a timely manner can also be a reason to reduce a node's fuzziness level. This way, various aspects of the middleware can act differently for members with different fuzziness.

For example, the flow control mechanism may allow the application to continue sending messages even if its messages were not acknowledged by members with low fuzziness. In the reliable delivery mechanism, we might employ less aggressive retransmission policies for fuzzy nodes to avoid overloading the network. Similarly, the stability mechanism can declare messages that were not acknowledged by fuzzy members as *fuzzy stable*; such messages can be partially deleted, striped with an enhanced RAID-like mechanism, or saved in a concise manner. Note that due to the cost of this manipulation, we do not wish to perform it for every message. The fuzziness threshold is used to select the messages for which we want to apply these mechanisms. We have recently completed prototyping a group communication toolkit that follows these guidelines, by augmenting Ensemble [21.13], and are in the process of benchmarking the system. Additionally, we are currently working on determining good thresholds for each of these concerns using the GloMoSim simulator. Looking at what other aspects of group communication systems can benefit from fuzzy membership information is an interesting open question. Similarly, more sophisticated schemes for capturing and expressing the fuzziness of a member are also desirable. It might be possible to adopt some results from the work on *tailored failure suspectors* [21.9]. However, [21.9] only suggests a framework for collecting relevant information and calculating a weighted health/fuzziness level of nodes. Yet, the important aspect is how to determine the weights in a meaningful way for our

needs, and what other parameters can be useful. Moreover, it is worth exploring other frameworks.

Note that all the issues mentioned above are internal to the infrastructure; they are not exposed to the application, which sees the same traditional virtually synchronous programming model, and thus are in line with Section 21.3 below. Yet, we do not completely rule out the introduction of *fuzzy guarantees* that are exposed to the application (although at this point we do not endorse them either). For example, it is possible to guarantee that a message will be delivered to all members above some fuzziness level, or that the delivery order of messages is guaranteed to be the same among all members above some fuzziness level. Furthermore, one should consider the notion of *fuzzy virtual synchrony*, in which virtual synchrony properties are guaranteed only among members above the threshold fuzziness level; this may obtain an additional reduction in the number of view changes. In fact, it is possible to consider other notions of fuzzy distributed computing, such as *fuzzy publish subscribe* systems, *fuzzy multicast*, *fuzzy consensus* based on *fuzzy failure detectors*, etc. An interesting problem is reasoning about a composable system in which each component makes fuzzy guarantees. It might be possible to use Bayesian networks,or a similar mechanism, for this task.

21.3 A Case for Strong Properties

One of the common techniques for improving the raw performance and scalability of middlewares is to weaken the guarantees they provide, and push the responsibility for obtaining eventual logical consistency to the application. For example, in optimistic approaches to replication, operations propagate through the network (usually) using point-to-point communication, and are applied on each server locally, in a tentative manner. The system guarantees that eventually, all operations that access the same object will be executed by all nodes in the same order, but some operations may need to be canceled and reexecuted on replicas that initially applied them in the wrong order.

Looking at distributed shared memory (DSM), a similar tradeoff has lead to the definition and implementation of various weak consistency models. Here too, each paper that proposes a weak consistency model presents a few sample applications to prove that the condition is indeed useful. However, experience showed that these conditions are very unpopular with programmers, despite their improved performance. The problem is that weak consistency conditions push the burden of providing the right semantics from the middleware to the application. This makes the application code more complex, and therefore more error prone. Additionally, it is much harder to prove the correctness of applications that are designed to operate with weak consistency conditions. We speculate that the same is likely to be true in distributed replicated systems as well. Thus, if we strive to come up with middlewares that facilitate the design of dependable systems, we should try to provide strong guarantees whenever feasible. As in DSM, it may be possible to provide strong guarantees either by incorporating these guarantees in the implementation, or by combining a smart compiler with a weak implementation. For example, in DSM, a compiler might be able to insert enough synchronization operations to make a program execute on a release consistent memory [21.12] as if it was sequentially consistent [21.1, 21.17]. Fuzzy membership is one direction that might prove useful in

providing strong guarantees efficiently in mobile ad-hoc environments *without* the aid of a compiler.

21.4 Other Related Issues

There were several attempts to increase the scalability of group communication systems, e.g., by using overlay networks [21.10], by creating hierarchies [21.3], and by resorting to probabilistic guarantees [21.5]. However, the issues we are trying to address are not necessarily caused by scale.

There have also been several works on diffusing information in ad-hoc sensor networks [21.7, 21.15]. Alas, these works do not guarantee reliable and/or ordered delivery. Another interesting problem in ad-hoc networks is routing, e.g., [21.20, 21.22, 21.25]. Yet, this problem is orthogonal to fuzzy membership.

Acknowledgements

I would like to thank Özalp Babaoğlu, Ken Birman, Keith Marzullo and the annonymous referees for their insightful comments. Thanks also to Erez Hadad, Claudia Lerner, and Galina Tcharny.

References

21.1 S. Adve and M. Hill. Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model. Technical Report 1107, Computer Science Department, University of Wisconsin, Wisconsin-Madison, September 1992.

21.2 Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

21.3 Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-2, The Center for Networking and Distributed Systems, Computer Science Department, John Hopkins University, 1998.

21.4 Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS–94–15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.

21.5 K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, , and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

21.6 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.

21.7 D. Braginsky and D. Estrin. Rumor Routing Algorithm For Sensor Networks. http://lecs.cs.ucla.edu/ daveey/work/lecs/rumorroute.pdf.

21.8 T. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM*, 43(4):685–722, July 1996.

21.9 F. Cosquer, L. Rodrigues, and P. Verissimo. Using Tailord Failure Suspectors to Support Distributed Cooperative Applications. In *Proc. of the 7th International Conference on Parallel and Distributed Computing and Systems*, October 1995.

21.10 R. Friedman, S. Manor, and K. Guo. Scalable Hypercube Based Stability Detection. In *Proc. of the 18th Symposium on Reliable Distributed Systems*, October 1999.
21.11 R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 140–149, October 1996.
21.12 K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.
21.13 M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
21.14 M. Hurfin and M. Raynal. A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209–223, 1999.
21.15 C. Intanagonwiwat, R. Govindan, and D. Estrin. Direct Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networks*, August 2000.
21.16 I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. In *Proc. of ACM Symposium on Principles of Database Systems*, pages 245–254, May 1995.
21.17 L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.
21.18 C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Toolkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.
21.19 A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large Scale Event Notification Infrastructure. In *Proceedings of 3rd International Workshop on Networked Group Communication*, November 2001.
21.20 E.M. Royer and C.E. Perkins. Multicast Operation of the Ad-Hoc On-Demand Distance Vector Routing Protocol. In *Proceedings of the 5th Annual International Conference on Mobile Computing and Networks*, pages 207–218, August 1999.
21.21 A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:194–157, 1997.
21.22 P. Sinha, R. Sivakumar, and V. Bhargavan. MCEDAR: Multicast Core-Extraction Distributed Ad-Hoc Routing. In *Proceedings of the IEEE Wireless Communication and Networking Conference*, September 1999.
21.23 D. Skeen. A Quorum-Based Commit Protocol. Technical Report TR82-483, Department of Computer Science, Cornell University, February 1982.
21.24 R. van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.
21.25 C. Wu and Y.C. Tay. AMRIS: A Multicast Protocol for Ad-Hoc Wireless Networks. In *Proceedings of the MILCOMM '99*, November 1999.
21.26 B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD-01-1141, Computer Science Department, U.C. Berkeley, April 2001.

22. Toward Self-Organizing, Self-Repairing and Resilient Distributed Systems

Alberto Montresor[1], Hein Meling[2], and Özalp Babaoğlu[1]

[1] Department of Computer Science, University of Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy
{montresor,babaoglu}@CS.UniBO.IT
[2] Department of Electrical and Computer Engineering, Stavanger University College,
PO Box 8002, Ullandhaug, N-4068 Stavanger, Norway
meling@acm.org

22.1 Introduction

As access to networked applications become omnipresent through PC's, hand-held and wireless devices, an increasing number of interactions in our day-to-day life with significant economical, social and cultural consequences depend on the reliability, availability and security of distributed systems. Not surprisingly, the increased demand that is being placed by users on networked services can only be met by distributed infrastructures with greater complexity and larger scale. And the extreme dynamism that is often present in different forms in modern systems further aggravates our ability to reason formally about their behavior.

Recent examples of these trends can be found in the *peer-to-peer* (P2P) and *ad-hoc networks* (AHN) application areas. P2P systems are distributed systems based on the concept of resource sharing by direct exchange between nodes that are considered *peers* in the sense that they all have equal role and responsibility [22.7]. Exchanged resources may include content, as in popular P2P document sharing applications, and CPU cycles or storage capacity, as in computational and storage grid systems. P2P systems exclude any form of centralized structure, requiring control to be completely decentralized. The P2P architecture enables true distributed computing, creating networks of resources that can potentially exhibit very high availability and fault-tolerance. In AHN, heterogeneous populations of mobile, wireless devices cooperate on specific tasks, exchanging information or simply interacting informally to relay information between themselves and the fixed network [22.10]. Communication in AHN is based on multihop routing among mobile nodes. Multihop routing offers numerous benefits: it extends the range of a base station; it allows power saving; and it allows wireless communication, without the use of base stations, between users located within a limited distance of one another.

Both P2P and AHN may be seen as instances of *dynamic networks* that are driven by large numbers of input sources (users/nodes) and that exhibit extreme variability in their structure and load. The topology of these systems typically changes rapidly due to nodes voluntarily joining or leaving the network, due to involuntary events such as crashes and network partitions, or due to frequent changes in interaction patterns. The load in the system may also shift rapidly from one region to another; for example, when certain documents become "hot" in a document sharing system.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 119–123, 2003.

Traditional techniques for building distributed applications are proving to be inadequate in dealing with the aforementioned complexity. For example, centralized and top-down techniques for distributed applications based on client-server architectures may simplify system management, but often result in systems that are inflexible and unable to adapt to changing environmental conditions. In these systems, it is not uncommon for some minor perturbation (e.g., a software update or a failure) in some remote corner of the system to have unforeseen, and at times catastrophic, global repercussions. In addition to being fragile, many situations (e.g., adding/removing components, topology changes) arising from the extreme dynamism of these systems require manual intervention to keep distributed applications functioning correctly.

In our opinion, we are at a threshold moment in the evolution of distributed computing: the complexity of distributed applications has reached a level that puts them beyond our ability to design, deploy, manage and keep functioning correctly through traditional techniques. What is required is a paradigm shift, capable of confronting this complexity explosion and enabling the construction of resilient, scalable, self-organizing and self-repairing distributed systems.

22.2 Drawing Inspiration from Nature

The centralized, top-down paradigms that are the basis for current distributed systems are in strong contrast with the organization of many phenomena in the natural world where decentralization rules. While the behavior of natural systems may appear unpredictable and imprecise at a local level, many organisms and the ecosystems in which they live exhibit remarkable degrees of resilience and coordination at a global level. Social insect colonies, mammalian nervous and immune systems are a few examples of highly-resilient natural systems.

We argue that ideas and techniques from *complex adaptive systems* (CAS), which have been applied successfully to explain certain aspects of biological, social and economical phenomena, can also be the basis of a programming paradigm for distributed applications. The natural systems cited in the previous paragraph all happen to be instances of CAS which are characterized by complete lack of centralized coordination. In the CAS framework, a system consists of a large number of autonomous entities called *agents*, that individually have very simple behavior and that interact with each other in simple ways. Despite this simplicity, a system composed of large numbers of such agents often exhibits what is called *emergent behavior* [22.3] that is surprisingly complex and hard to predict. Furthermore, the emergent behavior of CAS is highly adaptive to changing environmental conditions and unforeseen scenarios, is resilient to deviant behavior (failures) and is self-organizing toward desirable global configurations.

Parallels between CAS and modern distributed systems are immediate. In this paper, we suggest using ideas and techniques derived from CAS to enable the construction of resilient, scalable, self-organizing and self-repairing distributed systems as ensembles of autonomous agents that mimic the behavior of some natural or biological process. It is our belief that this approach offers a chance for application developers to meet the increasing challenges arising in dynamic network settings and obtain desirable global properties

such as resilience, scalability and adaptability, without explicitly programming them into the individual agents.

As an instance of CAS drawn from nature, consider an ant colony. Several species of ants are known to group objects (e.g., dead corpses) into piles so as to clean up their nests. Observing this global behavior, one could be mislead into thinking that the cleanup operation is somehow being coordinated by a "leader" ant. Resnick [22.8] shows that this is not the case by describing an artificial ant colony exhibiting this very same behavior in a simulated environment. Resnick's artificial ants follows three simple rules: (i) wander around randomly, until they encounter an object; (ii) if they were carrying an object, they drop it and continue to wander randomly; (iii) if they were not carrying an object, they pick it up and continue to wander. Despite their simplicity, a colony of these "unintelligent" ants is able to group objects into large clusters, independent of their initial distribution in the environment. This simple algorithm could be used to design distributed data analysis and search algorithms, by enabling artificial ants to travel through a network and cluster "similar" information items. It is also possible to consider a simple variant (the inverse) of the above artificial ant that drops an object that it may be carrying only after having wandered about randomly "for a while" without encountering other objects. Colonies of such ants try to disperse objects uniformly over their environment rather than clustering them into piles. As such, they could form the basis for a distributed load balancing algorithm.

What renders CAS particularly attractive from a distributed systems perspective is the fact that global properties such as adaptation, self-organization and resilience are exactly those that are desirable to distributed applications, and we may obtain these properties without having to explicitly program them into the individual agents. In the above example, there are no rules for ant behavior specific to initial conditions, unforeseen scenarios, variations in the environment or presence of deviant ants (those that are "faulty" and do not follow the rules). Yet, given large enough colonies, the global behavior is surprisingly adaptive and resilient.

Instances of CAS drawn from nature have already been applied to numerous problems with notable success [22.2, 22.5]. In particular, artificial ant colonies have been used for solving complex optimization problems, including those arising in communication networks. For instance, numerous simulation studies have shown that packets in networks can be routed by artificial ants that mimic real ants that are able to locate the shortest path to a food source using only trails of chemical substances called *pheromones* deposited by other ants [22.2].

22.3 Current Work and Future Directions

In order to pursue these ideas further, we have initiated the *Anthill* project [22.1], whose aim is to design a novel framework for the development of peer-to-peer applications based on ideas borrowed from CAS and multi-agent systems. The goals of Anthill are to provide an environment that simplifies the design and deployment of novel P2P applications based on swarms of agents, and provide a "testbed" for studying and experimenting with CAS-based P2P systems in order to understand their properties and evaluate their performance.

Anthill uses terminology derived from the ant colony metaphor. An Anthill distributed system is composed of a self-organizing overlay network of interconnected *nests*. Each nest is a peer entity sharing its computational and storage resources. The network is characterized by the absence of a fixed structure, as nests come and go and discover each other on top of a communication substrate. Nests handle requests originated by local users, by generating one or more *ants* – autonomous agents that travel across the nest network trying to satisfy the request. Ants communicate indirectly by reading or modifying their environment, through information stored in the visited nodes. For example, an ant-based distributed lookup service could leave routing information to guide subsequent ants toward a region of the network where the searched key is more likely to be found.

The aim of Anthill is to simplify P2P application development and deployment by freeing the programmer of all low-level details including communication, security and ant scheduling. Developers wishing to experiment with new protocols need to focus on designing appropriate ant algorithms using the Anthill API and defining the structure of the P2P system. When writing their protocols, developers may exploit a set of library components and services provided by nests. Examples of such services include failure detection, document downloading and ant scheduling for distributed computing applications.

A Java prototype of the Anthill runtime environment has been developed. The runtime environment is based on JXTA [22.4]. The benefits of basing our implementation on JXTA are several, including the reuse of various transport layers for communication, and dealing with issues related to firewalls and NAT. In addition to the runtime environment, Anthill includes a simulation environment to help developers analyze and evaluate the behavior of P2P systems. All simulation parameters, such as the structure of the network, the ant algorithms to be deployed, characteristics of the workload presented to the system, and properties to be measured, are easily specified using XML.

After having developed a prototype of Anthill, we are now in the process of testing the viability of our ideas by developing CAS-based P2P applications, including a load-balancing algorithm for grid computing based on the "dispersing" ant described in the previous section [22.6] and a document-sharing application based on keyword pheromone trails left by exploring ants [22.1]. Preliminary simulation results appear to confirm our intuition that applying CAS-based techniques to solving challenging problems in distributed systems is a promising approach. The performance of our preliminary results are comparable to those obtained through traditional techniques. Yet, the approach followed to obtain our algorithms is completely different from previous approaches, as we mimick the behavior of natural systems, thus inheriting their strong resilience and self-organizational properties.

Current efforts in applying CAS techniques to information systems can be characterized as *harvesting* – combing through nature, looking for a biological process having some interesting properties, and applying it to a technological problem by adapting it through an enlightened trial-and-error process. The result is a CAS that has been empirically obtained and that appears to solve a technological problem, but without any scientific explanation of why. In the future, we seek to develop a rigorous understanding of why a given CAS does or does not perform well for a given problem. A systematic

study of the rules governing fitness of CAS offers a bottom-up opportunity to build more general understanding of the rules for CAS behavior. This study should not be limited to biological systems; other areas such as economics and game theory [22.9] can be rich sources of inspiration. The ultimate goal is the ability to synthesize a CAS that will perform well in solving a given task based on the accumulated understanding of its regularities when applied to different tasks. The achievement of this goal would enable the systematic exploitation of the potential of CAS, freeing technologists from having to comb through nature to find the desired behavior.

We conclude with a brief discussion of the inherent limitations of the CAS-based approach to building distributed systems. While natural systems exhibit desirable properties at a global level, their local behavior can be unpredictable and imprecise. Thus, we expect CAS-based techniques to be appropriate for distributed systems where probabilistic guarantees on desired global system behavior are sufficient. The technique is probably not appropriate for guaranteeing strong properties such as consistency, synchronization, and QoS.

References

22.1 Ö. Babaoğlu, H. Meling, and A. Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proc. of the 22th Int. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.
22.2 E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.
22.3 J. Holland. *Emergence: from Chaos to Order*. Oxford University Press, 1998.
22.4 Project JXTA. http://www.jxta.org.
22.5 E. Klarreich. Inspired by Immunity. *Nature*, 415:468–470, Jan. 2002.
22.6 A. Montresor, H. Meling, and O. Babaoğlu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In *Proc. of the 1st Workshop on Agent and Peer-to-Peer Systems*, Bologna, Italy, July 2002.
22.7 A. Oram, editor. *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly, Mar. 2001.
22.8 M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1994.
22.9 K. Ritzberger. *Foundations of Non-Cooperative Game Theory*. Oxford University Press, Jan. 2002.
22.10 C. Toh. *Ad Hoc Mobile Wireless Networks*. Prentice Hall, 2002.

Part III
Challenges in Distributed Information and Data Management

Management of information and data in distributed settings requires replication and redundancy. The large size and exponential growth of networked components and resources increase the instability in distributed and replicated data management systems. Such systems offset this instability by increasing the redundancy of their states and services. In turn, data availability in such distributed systems is a function of this redundancy.

Additional redundancy increases the difficulty of providing a consistent system. Papers in Part III explore the dual challenges of (i) increasing redundancy to ensure performance, fault-tolerance, and availability while (ii) simultaneously providing (different levels of) consistency in the distributed system. Included are topics dealing with trading off availability for consistency and with the implementation consequences of such trade-off decisions. In particular, one topic considers utilities that make dynamic resource allocation decisions that simultaneously incorporate availability and consistency into the decision process. Another topic explores using distributed system techniques to build more reliable replicated databases. Other topics deal with disseminating information in distributed systems through publish/subscribe communication mechanisms. The final topic considers redundancy as a base mechanism for highly available systems.

Researchers have long studied resource allocation problems in operating systems. However, resource allocation in networked systems is more challenging due to the additional requirements of availability and quality of service, and due to the ambiguity in logical ordering of events. The first paper by Vahdat (page 127) discusses dynamic provisioning in distributed systems to meet target levels of performance, availability, and data quality. This paper addresses the twin challenges of decentralization and dynamic resource allocation. It describes a utility that dynamically allocates available resources among thousands of competing services. Further, the paper discusses the challenges in building an infrastructure for decentralized, replicated services.

Successful use of a database system depends in part on how well the system deals with the inherent replication of the managed information and data. Gray observed that distributed databases have not been successful due the dangers of replication; specifically, replication can lead to degradation in performance, consistency, and/or availability. Recently, the distributed systems community has attempted to apply distributed systems theory and mechanisms to database replication. A paper by Kemme (page 132), describes the challenges in dealing with replication using multicast primitives of group communication. The paper also notes that the transfer of research results from the distributed research community to industry is slow. The author argues that the research community has to build small "real" systems to validate designs and to convince industry to use the mechanisms.

Next we look at the use of publish/subscribe mechanisms for information dissemination. A paper by Baldoni *et al.* (page 137) describes how publish/subscribe com-

munication systems have evolved from classical message passing to the many-to-many communication paradigm initially based on topic and then also on content. The considerations of scale and efficient communication serve as reasons for each evolutionary step. Point-to-point message passing does not scale as well as topic-based publish/subscribe communication using IP-multicast. Further IP-multicast is limited to only hundreds or thousands of groups or topics, far short of Internet scale communication, leading to the next step of content-based communication using application-level multicast.

The rest of the papers discuss mechanisms for dealing with redundancy in distributed systems. We break the papers into two groups: papers that discuss systems built using redundant codes (e.g., erasure codes), and papers that discuss systems built using pure replicated data and services.

The next three papers discuss systems built using redundant codes. The first paper by Weatherspoon *et al.* (page 142), discusses some problems with building systems using erasure codes, and providing naming and integrity, and proposes a solution that allows all erasure-coded fragments and the document itself to be verified locally using one name. Such a design is well-suited for solutions to the problem of storing data in untrusted distributed systems. The second paper, by Hand and Roscoe (page 148), also discusses building a system using erasure codes additionally focusing on anonymity. The authors designed a steganographic storage system; that is, a system in which users who do not have the required key not only are unable to read the contents of documents stored under that key (as with a conventional encrypted file system), but furthermore are unable to determine the existence of files stored under that key. The third paper, by Bhagwan *et al.* (page 153), explores the dual issues of what availability guarantees can be made using existing systems; conversely, how to best achieve a desired level of availability using mechanisms available. The paper describes availability using different forms of replication, replica placement policies, and application characteristics.

Part III concludes with two papers that discuss systems built using replicated data and services. The common thread in all existing replicated systems is the need to keep replicas consistent. The first paper by Chockler *et al.* (page 159) discusses a data-centric replication paradigm that separates the replication of control and the replication of state. Data-centric replication differs from a process-centric approach in that replica state is not necessarily kept in persistent storage, but in the process. As a result, data-centric approach is more scalable since the use of persistent storage allows many unknown clients to read from the persistent storage and the membership does not need to include all clients. The second paper by Shapiro and Saito (page 164) compares pessimistic and optimistic replication. That is, pessimistic replication assumes conflicts are the common case in write-sharing systems and therefore checks for stale reads and lost writes on every access, incurring a large latency penalty; whereas, optimistic replication assumes conflicts are rare and handles them off-line allowing the common case to be fast.

To summarize, as distributed systems grow in scale, providing availability, replication, fault-tolerance and consistency becomes correspondingly more challenging. The papers in Part III formulate directions for meeting the challenges of managing replicated data and services, and explore the consequences of pursuing these technical directions.

Hakim Weatherspoon

23. Dynamically Provisioning Distributed Systems to Meet Target Levels of Performance, Availability, and Data Quality

Amin Vahdat

Department of Computer Science
Duke University
http://issg.cs.duke.edu

23.1 Introduction

Increasingly, critical compute services are being delivered across wide-area networks. Today, a primary challenge to computer systems is building highly reliable services. However, the same decentralized nature that contributed to its sustained, exponential growth makes it difficult to deliver highly reliable services over the Internet. Failures and performance fluctuations in the middle of the network mean that no centralized service can continuously deliver high levels of performance and availability to all of its clients. Further, highly bursty and unpredictable access patterns force developers to over-provision their services for some expected worst case. Of course, it is only economically feasible to over-provision by some maximum amount.

To address the twin challenges of decentralization and dynamic resource allocation, we envision a distributed compute *utility* that dynamically allocates available resources among thousands of competing services. In this model, the utility may be composed of thousands of cooperating sites ("points of presence") each with thousands of individual machines. The utility replicates services at strategic points in the network based on dynamically changing network characteristics, failure patterns, client access patterns, etc. The utility makes resource allocation decisions based on Service Level Agreements (SLAs) negotiated with individual services. These SLAs enable to utility to intelligently allocate and deallocate resources among competing services during times of resource constraint. For example, during a breaking news story or an emergency situation, the utility may allocate most available resources to news services.

A primary challenge to achieving this vision is achieving target levels of service "utility" with the minimal global "cost". Utility may be measured in any number of application-specific ways, including performance, availability, and data quality. Performance typically lends itself to a straightforward application-specific definition, such as throughput. Availability is more difficult to quantify [23.10]. Traditional measures of availability for a network service typically focus on service uptime. However, a service may be functioning correctly, but under such heavy load that it is unable to deliver satisfactory performance for a portion of the request stream. An appropriate definition of availability must account for all such considerations. Thus, we view appropriate availability metrics, benchmarks, and evaluation methodologies as an important research challenge in isolation. Data quality may be measured in a number of ways, including the

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 127–131, 2003.

percentage of the database represented by a given query [23.3, 23.5] or the consistency level of the returned data [23.9].

Overall system utility may take the form of more complicated convolutions of individual underlying metrics, for instance, maximizing performance while maintaining some minimal level of data quality. In general, delivering higher utility requires additional system resources, for instance, more CPUs or network bandwidth to deliver higher throughput or additional service replicas to improve availability. Thus, an important research goal is to determine techniques for maximizing utility for minimal cost.

We are pursuing this vision of a wide-area compute utility, through the design, implementation, and evaluation of Opus, and *overlay peer utility service* [23.1]. We next provide a brief overview of our approach in Opus, provide a summary of our initial results, and summarize some of the remaining challenges to achieving our goals.

23.2 An Overlay Peer Utility Service

We are pursuing our agenda of dynamically placing functionality at appropriate points in the network in the context of Opus [23.1], an overlay peer utility service. As discussed earlier, individual applications specify their performance and availability requirements to Opus. Based on this information, Opus maps applications to individual nodes across the wide area. The resulting optimization problems are practical and efficient given reasonable constraints on the form of the utility functions [23.6].

The initial mapping of applications to available resources is only a starting point. Based on observed access patterns to individual applications, Opus dynamically reallocates global resources to match application requirements. For example, if many accesses are observed for an application in a given network region, Opus may reallocate additional resources close to that location. One key challenge to achieving this model is determining the relative utility of a given candidate configuration. That is, for each available unit of resource, we must be able to *predict* how much any given application would benefit from that resource. Existing work in resource allocation in clusters [23.2] and replica placement for availability [23.11] indicate that this can be done efficiently in a variety of cases.

Many individual components of Opus require information on dynamically changing system characteristics. We use network overlays to establish application-level peering relationships among Opus sites. An overlay network enables arbitrary logical topologies on top of the underlying IP network, for instance, to support efficient data propagation trees, application-layer multicast, etc. Opus employs a global *service overlay* to interconnect all available Opus sites and to maintain soft state about the current mapping of utility nodes to hosted applications. The service overlay is key to many individual system components, such as routing requests from individual clients to appropriate replicas and performing resource allocation among competing applications. Individual services running on Opus employ *per-application overlays* to disseminate their own service data and metadata among individual replica sites.

Clearly, a primary concern is ensuring the scalability and reliability of the service overlay. In an overlay with n nodes, maintaining global knowledge requires $O(n^2)$ network probing overhead and $O(n^2)$ global storage requirements. Such overhead quickly

becomes intractable beyond a few dozen nodes. Peer-to-peer systems can reduce this overhead to approximately $O(n \lg n)$ [23.8] but are unable to provide any information about global system state, even if approximate. Opus addresses scalability issues through the aggressive use of hierarchy, aggregation, and approximation in creating and maintaining scalable overlay structures. Opus then determines the proper level of hierarchy and aggregation (along with the corresponding degradation of resolution of global system state) necessary to achieve the target network overhead.

A primary use of the service overlay is to aid in the construction of the per-application overlays. Once Opus makes resource allocation and replica placement decisions, it must create an interconnection graph for propagating data (events, stock quotes, replica updates) among application nodes. It is important to match an overlay topology to the characteristics of individual applications, for instance, whether an application is bandwidth, latency, or loss sensitive. In general, adding more edges to an application overlay results in improved "performance", with a corresponding increase in consumed network resources. Our economic model for resource allocation helps determine the proper balance between performance and cost on a per-application basis.

Security is an important consideration for any general-purpose utility. Opus allocates resources to applications at the granularity of individual nodes, eliminating a subset of the security and protection issues associated with simultaneously hosting multiple applications in a utility model. We believe that a cost model for consumed node and network resources will motivate application developers to deploy efficient software for a given demand level. Initially, we consider applications that are self-contained within a single node. Longer term, however, a utility service must support multi-tiered applications (e.g., a web server that must access a database server or a storage tier). In such a context, one interesting issue is determining the proper ratio of allocation among application tiers, e.g., the number of storage nodes needed to support a particular application server.

23.3 Initial Results

To date, we have made initial progress on a number of fronts in support of Opus, as discussed below.

- *Continuous consistency:* We have defined a continuous consistency model that captures the semantic space between optimistic and strong consistency [23.9]. Applications can dynamically trade relaxed consistency for increased performance and availability, defining another dimension along which Opus can allocate resources while balancing performance, availability, and consistency. In this space, we have also shown how to optimally place (both in terms of number and location) replicas to achieve maximum availability for a given workload and faultload (network failure characteristics) [23.11].
- *Cluster-based resource allocation policies:* In Muse [23.2], we have shown how a centralized load balancing switch can measure per-application throughput levels and service level agreements to determine how to allocate resources to maximize throughput. One key challenge to extending this work to the wide area is to make similar

resource allocation decisions in a decentralized manner based on incomplete and potentially stale information and to incorporate other metrics beyond throughput, such as availability or consistency, into our calculations.

– *Adaptive Cost, Delay Constrained Overlays:* As discussed above, application overlays form a key component of the service architecture. We are exploring the space between small-scale overlays that use global knowledge (making them unscalable) to construct and maintain appropriate topologies and large-scale peer-to-peer systems that randomly distribute functionality across the network to achieve scalability (while sacrificing control over exact performance and reliability of the resulting topology). We have designed and built a scalable overlay construction technique that eliminates any centralized control or global knowledge [23.7]. Our performance evaluation indicates that we are able to export a flexible knob that allows application developers to quickly converge to structures that deliver the performance of shortest path trees (with associated high cost) at one extreme and the low resource utilization of a minimum cost spanning tree at the other extreme (with associated low performance). Applications are able to specify where on this spectrum they wish to reside based on current network and application characteristics. The structure is also adaptive to changing network conditions, quickly adapting to achieve target levels of performance or cost with low per-node state and network overhead.

23.4 Future Challenges

There are many open questions associated with realizing the vision described in this paper. We briefly describe a subset in this section.

Infrastructures for decentralized, replicated services. A fundamental premise of the architecture is that application structure allows a dynamic mapping of functions and data onto physical resources, including dynamic replication. One key question involves whether it is possible for a shared infrastructure to simultaneously support the requirements of a broad range of network services, including replicated web services, application-layer multicast, and storage management.

Overlay construction and service quality. The number and placement of server sites affects multiple dimensions of service quality in a complex way. While allocation of additional server sites typically yields better performance and availability, in some cases additional resources can actually reduce overall performance and availability [23.10]. For a replicated service, the need to propagate updates between replicas imposes new network demands and may compromise availability for applications with strong consistency requirements. Further, the availability and performance of a candidate replica configuration depends in part on the overlay topology connecting the replicas. To configure overlays automatically, the system must efficiently generate valid candidate topologies that meet service quality goals while balancing network performance and cost. Finally, algorithms for building target overlay topologies must be scalable and adaptive to changing network characteristics.

SLAs for automated provisioning and configuration. One key issue is expressing application targets for service quality and using these targets to balance competing demands

on the system. In Opus, utility functions form the basis of Service Level Agreements incorporating multiple dimensions of service quality, including availability and consistency quality as well as performance. Our recent work [23.2, 23.4] shows that utility functions can capture dynamic tradeoffs of service quality and cost; a key challenge of future work is to extend this approach to balance competing dimensions of service quality in an application-directed way.

Scalable resource mapping. The core of the Opus architecture is a set of resource allocation algorithms to map services and their workloads onto the physical server resource pool. Based on a prioritization of hosted applications, individual utility nodes must make local decisions, based on potentially outdated information, to approximate the global good. In the wide area, highly variable network conditions make replica placement as important a decision as resource allocation. In fact, fewer well-placed replicas are likely to outperform more poorly-placed replicas, especially when considering consistency requirements and per-application scalability issues.

References

23.1 Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

23.2 Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, October 2001.

23.3 Armando Fox and Eric Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *Proceedings of HotOS-VII*, March 1999.

23.4 Yun Fu and Amin Vahdat. Service Level Agreement Based Distributed Resource Allocation for Streaming Hosting Systems. In *Proceedings of the Seventh International Workshop on Web Caching and Content Distribution (WCW)*, August 2002.

23.5 Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1997.

23.6 Toshihide Ibaraki and Naoki Katoh, editors. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, MA, 1988.

23.7 Dejan Kostić, Adolfo Rodriguez, and Amin Vahdat. The Best of Both Worlds: Adaptivity in Two-Metric Overlays. Technical Report CS-2002-10, Duke University, May 2002. `http://www.cs.duke.edu/~vahdat/ps/acdc-full.pdf`.

23.8 Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

23.9 Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, October 2000.

23.10 Haifeng Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

23.11 Haifeng Yu and Amin Vahdat. Minimal Replication Cost for Availability. In *Proceedings of the ACM Principles of Distributed Computing*, July 2002.

24. Database Replication Based on Group Communication: Implementation Issues

Bettina Kemme

School of Computer Science
McGill University
Montreal, Quebec, Canada
kemme@cs.mcgill.ca

24.1 Introduction

For many years, developers of group communication systems have claimed that database replication is an interesting application for their multicast primitives. Over the last years, several research groups have had a closer look into this claim and started to analyze how group communication primitives can support replica control in database systems, for instance [24.10, 24.20, 24.19, 24.16, 24.15, 24.14, 24.22, 24.13, 24.4, 24.2]. The main ideas of these proposals are to use the ordering guarantees of multicast primitives to serialize conflicting transactions, and to simplify atomic commit protocols by using reliable or uniform reliable multicast. Most of the results are published in distributed systems conferences, and so far, they have received little attention in database conferences. There, the focus has been on quorum protocols [24.23] (for fault-tolerance), and adaptive lazy replication schemes (for performance and scalability) [24.9, 24.6, 24.5]. However, out of this wide range of research directions, only few ideas have found their way into commercial database systems. Except for highly specialized backup systems, commercial solutions usually choose performance over consistency. Practitioners recommend to not use these replication strategies for systems with high update rates: "...In my experience most of these (replication) schemes end up being more trouble than the benefit they bring" [24.7].

Looking at this low transfer rate between research results and commercial systems, I believe that the use of group communication systems for database replication will only then become reality, if the theoretical results proposed so far will be validated through the development and evaluation of prototypes and small "real" systems (e.g., in the form of open-source software modules). While simulation results of the proposed approaches have been very promising (e.g., [24.20, 24.15, 24.13]), only prototypes will be able to convince industry of the usefulness of the approach. Such research is especially challenging:

- It requires in-depth knowledge in both database systems and distributed systems, and solid background in both theory and systems.
- Two different systems are combined (database and group communication systems). Hence, there exist many alternatives for the overall architecture showing different degrees of interleaving and interaction.
- A realistic solution has to consider many different issues: concurrent execution of transactions, failure-handling, recovery, full support of the SQL standard, and user

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 132–136, 2003.

transparency. Each of these issues by itself is already challenging, combining them makes a "real" implementation very hard.

24.2 State of the Art and Future Challenges

There exist two alternatives of approaching the problem: an independent middleware based replication tool, or the integration of the replication semantics into the database system. In the first case, the replication tool could be an extension to, e.g. CORBA or J2EE. In the second case, the approach would follow database internal replication schemes as implemented in Oracle, Sybase etc. I am aware of three prototype implementations. Postgres-R [24.16] integrates replication into the kernel of the database system PostgreSQL [24.21] and can use both Ensemble [24.11] and Spread [24.3] as group communication system. Amir et al. [24.2] describe a middleware based replication tool using Spread and PostgreSQL. The PostgreSQL development team is interested in incooperating the ideas of these two prototypes into their distribution [24.8]. Jiménez-Peris et al. [24.14] also developed a middleware system, this time based on Ensemble and PostgreSQL.

These prototypes already handle a considerable amount of issues but still lack functionality that is crucial in "real life" systems. In order to add this functionality, both more theoretical work and smart strategies for translating theoretical solutions into practical implementations are needed:

– *Concurrency Control*: Although following the total order in case of conflicts, it should be possible to execute transactions concurrently whenever they do not conflict. If replication is implemented as middleware, the middleware must provide such a concurrency control mechanism since current database systems do not allow the application to specify an order in which concurrent transactions should be serialized. However, the middleware receives SQL or other high-level statements from the client, and hence, it is hard to determine which operations actually conflict. As a consequence, Amir et al. [24.2] execute update transactions serially, Jiménez-Peris et al. [24.14] lock entire relations or partitions, which only allows for restricted parallelism. Concurrency control in middleware systems has been a research topic for a long time but it is remains unclear whether existing solutions are satisfying.
  If replication is performed within the database system, things seem to be easier. Many of the more theoretical research papers have proposed various adjustments to existing concurrency control protocols in order to handle replication [24.13, 24.20, 24.18]. This goes so far as to exploit weaker message ordering mechanisms (like FIFO or causal) in order to provide more concurrency, e.g. [24.22]. However, it is very hard to transform these proposals into practical solutions since the concurrency control module usually heavily depends and interacts with the other components of the database system (e.g., storage manager, access paths, buffer manager, etc). Hence, concurrency control cannot be simply added as an independent component to the database kernel but will probably be a specialized solution for a specific database system. Postgres-R [24.16] provides a restricted form of fine-granularity concurrency control, and discussions with the PostgreSQL development team have shown the willingness to include a

replication based concurrency control into the PostgreSQL engine as a long-term goal. I believe this will be an important step for this research to become generally accepted in industry.

- *Transaction Model:* Most of the transaction models proposed so far have some restrictions. Amir et al. [24.2] and Kemme et al. [24.18] assume that a transaction consists of a single SQL statement or is a single stored procedure executed within the database system. Hence, whenever a transaction arrives, it can be sent to all sites. However, transactions often consist of more than one operation and these operations must be executed one by one since operations might depend on each other. Jiménez-Peris et al. [24.14] and Postgres-R [24.16] overcome this problem by executing a transaction first locally and multicasting all changes in a single message at the end of the transaction. Keeping track of all changes performed by a transaction might lead to a considerable overhead if the database system does not offer adequate support. Further approaches have been proposed, e.g., delaying write operations, including read operations into the messages, or sending messages for each operation [24.20, 24.1, 24.13] – each of these mechanisms having its own advantages and disadvantages. However, none of these approaches has been implemented in a prototype, and it is not clear how they can be applied in practice.

  In my opinion, middleware solutions might have to live with compromises. In contrast, implementations within the database system have more flexibility in solving these issues, and Postgres-R, for instance, works fine with various (but not yet all) types of transactions. Much more work is needed in regard to evaluating the feasibility of different execution models and comparing them for different execution environments.
- *Failure and Recovery*: Talking to people from industry [24.8], it has become clear that failure handling (failure detection, exclusion of the failed sites and load migration to surviving sites) and recovery (rejoining previously failed sites and including new sites into the productive system) are very crucial issues. Most importantly, the reconfiguration process should be as automatic as possible, and should not interrupt ongoing transaction processing significantly. Furthermore, no replica may miss any transactions.

  Discussions in [24.1, 24.16, 24.2] show that in regard to failures, group communication systems are able to do most of the work while the database system only has to perform minor clean-ups. In regard to recovery and integration of new nodes, there are many interesting alternatives depending on where the main recovery support takes place (in the middleware, within the database system, or within the group communication system). Postgres-R currently copies and transfers the entire database to the joining node (still a suboptimal solution). Amir et al. [24.2] log all messages in the middleware layer. If a site is down for a short time, it receives all missed messages at restart, otherwise a database transfer might take place. This is very similar to stable message queues used for lazy replication in commercial systems. [24.12, 24.17] have suggested more flexible schemes providing an optimized data transfer that does not interrupt transaction processing in the system. I believe that reconfiguration in general, and recovery in particular, are issues that will need much further analysis, both in theory and practice. Both the database system and the group communication system provide plenty of functionality for failure handling and recovery. We are currently working

at a clear interface between these two systems such that the functionality of both components can be exploited: the work should be distributed and not duplicated. For instance, the database system performs extensive logging and one should take advantage of it in order to avoid logging of messages in the middleware layer.

- *Transparency*: Transparency is a crucial issue. The main question is of how to embed middleware based replication tools. They might be part of an existing middleware infrastructure (e.g., CORBA or J2EE) and applications are developed using this middleware. Alternatively, the replication tool is put between legacy application and database system without the application being aware of it [24.2].
- *Further Issues:* I only want to mention two further research directions:
  When looking at wide area replication priorities might shift: On the one hand, weaker and flexible transaction models might be acceptable and concurrency within a single site might not be crucial. On the other hand, adaptive failure-handling and recovery are even more important. Message latency will become a significant factor, and optimistic strategies will have to be considered.
  When looking at completely heterogeneous environments or systems in which access is restricted (e.g., access is only available through web-based interfaces), reasonable models have to be developed that are able to handle the given restrictions and still provide acceptable guarantees.

24.3 Conclusion

Recent research has shown that group communication can be used to provide better database replication mechanisms than currently provided in commercial systems. However, I believe that the research part has not yet been completed. One of the main tasks left to do is to prove that the solutions can be transferred into real systems. Furthermore, the theoretical framework must be extended and adjusted in order to address the requirements of a real environment.

References

24.1 D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting Atomic Broadcast in Replicated Databases. In *Proc. of Euro-Par Conf.*, 1997.
24.2 Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical Wide Area Database Replication. Technical Report CNDS-2002-1, CS Dep., John Hopkins University, 2002.
24.3 Y. Amir and J. Stanton, The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, CS Dep., John Hopkins University, 1998.
24.4 Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, 2002.
24.5 Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update Propagation Protocols For Replicated Databases. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, 1999.
24.6 P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 1996.
24.7 S. Allamaraju et. al. *Professional Java Server Programming, J2EE Edition*. Wrox Press, 2000.

24.8 GBorg. *PostgreSQL related Projects: The pgreplication Project; ÊPostgreSQL Replication.* http://gborg.postgresql.org/project/pgreplication/projdisplay.php, 2002.
24.9 J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, 1996.
24.10 R. Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *Proc. of IEEE Int. Workshop on Distributed Algorithms (WDAG'95)*, 1995.
24.11 M. Hayden. The Ensemble System. Technical Report TR-98-1662, CS Dept. Cornell Univ., 1998.
24.12 J. Holliday. Replicated Database Recovery Using Multicast Communications. In *Proc. of IEEE Symp. on Network Comp. and Applications (NCA2001)*, 2001.
24.13 J. Holliday, D. Agrawal, and A. El Abbadi. The Performance of Database Replication with Group Communication. In *Proc. of Int. Symp. on Fault-Tolerant Computing (FTCS)*, 1999.
24.14 R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, 2002.
24.15 B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Trans. on Database Systems,* 25(3), 2000.
24.16 B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of Int. Conf. of Very Large Databases (VLDB)*, 2000.
24.17 B. Kemme, A. Bartoli, and Ö. Babaoğlu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.
24.18 B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, 1999.
24.19 F. Pedone and S. Frolund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proc. of Int. Symp. on Reliable Distr. Systems (SRDS)*, 2000.
24.20 F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *Proc. of Euro-Par Conf.*, 1998.
24.21 PostgreSQL v6.4.2. http://www.postgresql.com, 1998.
24.22 I. Stanoi, D. Agrawal, and A. El Abbadi. Using Broadcast Primitives in Replicated Databases. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, 1998.
24.23 A. Wool. Quorum Systems in Replicated Databases: Science or Fiction? *Bulletin of the Techn. Committee on Data Engineering*, 21(4), 1998.

25. The Evolution of Publish/Subscribe Communication Systems*

Roberto Baldoni, Mariangela Contenti, and Antonino Virgillito

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia.
{baldoni,contenti,virgi}@dis.uniroma1.it

25.1 Introduction

In the last years, a growing attention has been paid to the publish/subscribe (pub/sub) communication paradigm as a mean for disseminating information (also called events) through distributed systems on wide-area networks. Participants to the communication can act as *publishers*, that submit information to the system, and as *subscribers*, that express their interest in specific types of information. Main characteristics of such *many-to-many communication paradigm* are: the interacting parties do not need to know each other (anonymity), partners do not need to be up at the same time (decoupling in time), and the sending/receipt does not block participants (decoupling in flow). So, the publish/subscribe paradigm has been largely recognized as the most promising application-level communication paradigm for integration of information systems.

Pub/sub systems can be classified in two main classes: the *topic-based* class (e.g. TIB/Rendezvous [25.11]) and the *content-based* class (e.g. SIENA [25.9] and Gryphon [25.3]). In a topic-based system, processes exchange information through a set of predefined subjects (topics) which represent many-to-many distinct (and fixed) logical channels. Content-based systems are more flexible as subscriptions are related to specific information content and, therefore, each combination of information items can actually be seen as a single dynamic logical channel. This exponential enlargement of potential logical channels has changed the way to implement a pub/sub system. As an example, a content-based pub/sub system cannot indeed rely on (i) *centralized* architectures based on (ii) *network level* solutions (such as IP multicast) for subscriptions and information diffusion as a single server could not keep up with a high number of subscribers and the limited number of IP multicast addresses could not fit the potential large number of logical channels. Therefore, very recently it is becoming clear that the best way to implement such systems is at *application-level* through a set of *event brokers* which exchange information on a point-to-point basis through a TCP/IP platform. The cooperation among the event brokers has to take into account the following basic problems: *subscription and information routing*. Subscription routing refers to the process of disseminating subscriptions among the brokers in order to create a mapping between subscriptions and subscribers. Information routing includes the *matching* and the *forwarding* operations. Matching refers to the operation of identifying the set of receivers (i.e., subscribers) for

* This work is partially supported by a grant from MURST in the context of project "DAQUINCIS" and by grants of the EU in the context of the IST project "EU-Publi.com".

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 137–141, 2003.

each information item arrived at a broker. This is done by using the mapping between subscriptions and subscribers. Forwarding refers to the operation of routing through a sequence of TCP connections to get the final destinations. The overall aim of the brokers cooperation is therefore (i) to optimize some application-related performance metrics such as the overall information throughput or the number of TCP hops per information diffusion, and (ii) to ensure the specification of the pub/sub service such as reliable and/or timely information diffusion etc.

25.2 State of the Art

First pub/sub prototypes have been developed and deployed over LANs e.g. TIB/Rendezvous [25.5] and Elvin [25.8]. To reduce the network traffic the native LAN broadcast property was exploited. Publishers multicast information within the broadcast domain of the LAN and then each subscriber locally filters out information that does not match its interest. The next step was to bring a pub/sub system on a WAN in order to enlarge the potentiality of these systems in terms, for example, of number of subscribers. An enhanced version of TIB/Rendezvous based on rendezvous router daemons [25.11], SIENA [25.9] and Gryphon [25.3] are examples of such wide-area pub/sub systems. To get scalability, these prototypes are based on an architecture of networks of event brokers that exchange messages through a TCP/IP platform which provides basic point-to-point connectivity.

They differ each other from the way they implement the basic pub/sub mechanisms (subscription and information routing). More specifically, in TIB/Rendezvous information routing is done without taking into account the content of the information. Each information sent by an event broker (called *router daemon* in TIB terminology) is diffused to the other router daemons through a multicast tree based on the topology of TCP connections between the router daemons. Each router daemon stores information on subscribers of its own LAN and subscription routing is then performed only across LAN boundaries. Gryphon employs an efficient information matching algorithm [25.1], that assumes static subscription tables in which each node maintains all the subscriptions in the system. Therefore subscription routing reduces actually to flood the network of event brokers with each subscription. Siena embeds subscription routing algorithms [25.2] that specify how the subscription table at each broker is maintained, preventing subscriptions flooding. This is done by exploiting containment relationships among subscriptions. All previous systems share a common factor. The forwarding operation is done through a fixed topology based on TCP connections. However, to improve the overall application performance, a dynamic and self configuring underlying TCP topology would help to reduce for example the number of TCP hops to be traversed by an information to reach the set of intended subscribers.

Such dynamic TCP topologies are similar to *overlay network infrastructures* (e.g. Overcast [25.4], Tapestry [25.13], Pastry [25.6], i3 [25.10]) studied in the context of scalable information diffusion applications (e.g. multimedia streaming applications). Overlay networks main target is to set up on-the-fly an application-level distribution tree where the root routes content messages (e.g., a “live" video) to a dynamic group of clients. They aim at achieving high performance (as trees are built to optimize network

service parameters), robustness (as trees can rearrange to route around failed nodes), and generality (because any host connected to the Internet can join the network). The notion of overlay network matches very well the structure of a topic-based pub/sub system when considering each topic is associated with a multicast tree (e.g., Bayeux [25.12] and Scribe [25.7]) and information routing is done exploiting a basic lookup service of the overlay network. This approach cannot be employed, as it is, by content-based systems since there can be a different multicast tree for each published information.

25.3 Research Directions

Pub/sub systems represents a nice meeting point of many research communities such as databases, software engineering and distributed systems. As an example, in the context of databases, the definition of "ad-hoc" languages for subscription declaration and publishing advertising is an open problem. Using SQL as such language would imply an a-priori knowledge of the schema of the database which, in a content-based pub/sub system, corresponds to the a-priori knowledge of the structure of the information space. This could be not available or it could change along the time. In the sequel we point out a few research directions related to distributed computing[1]. We consider at first the problem of defining smart *subscription routing* algorithms that, on one hand, try to limit the diffusion of all subscriptions to every broker maintaining, on the other, a certain degree of availability of the mapping between the subscription and the subscribers. For example, SIENA is not fault tolerant in this sense. If a broker fails, its subscriptions are lost. A solution to this problem could be to consider the information space partitioned among overlapping subsets where each broker is responsible of the mapping between subscribers and subscriptions of a given set. In this way a subscription could be stored into more than one broker and, therefore, the failure of a broker does not lead to the loss of that subscription. Criteria and techniques for defining the sets (or other data structures), for assigning sets to brokers and to master the failover phase have to be studied.

Another issue is the designing and the implementation of *information routing based on dynamic TCP topologies* or, more specifically, how to effectively exploit overlay networks techniques in the information routing problem with particular emphasis on content-based pub/sub systems. TCP connections between brokers can be dynamically rearranged to adapt to the logical structure of subscriptions. Such a dynamic mechanism aims at reducing the number of TCP hops per information diffusion, avoiding that a broker acts as pure forwarder of information (i.e., that broker does not store any subscription for that information). At the same time, the delay per information diffusion can be reduced by associating a weight function with each potential TCP connection. This function gives a measure of the available bandwidth for that TCP link in order to decide which pair of brokers has to be connected (a transcontinental TCP link is expected to have much less bandwidth than a TCP link between two brokers in the same campus). Finally, a deep study is needed on the relationship between the subscription routing

[1] We do not consider important research topic such as security, impact of mobility and performances of a pub/sub system. These aspects are actually orthogonal to any pub/sub solution and they have to be taken into account by any solution in order to be a valid one.

and information routing, that could for example consider the use of different dynamic underlying topologies optimized for each of the two functions.

Another future direction of research is on the definition of *formal specifications of the services*, in terms of liveness and safety, provided by a pub/sub systems (like the formal specifications of services provided by group toolkits). To our knowledge there is no agreement on such formal specification and this makes difficult, firstly, to give a formal proof of any algorithm implementing a given service, and, secondly, to compare a given algorithm with another one. This formal specification has to take into account pub/sub communication paradigm peculiarities. For example, keeping the anonymity of the end-users implies a pub/sub system cannot provide end-to-end service semantics between a publisher and a subscriber (like in a client/server interaction). It can rather provide a "one-side" service semantics: one for a subscriber and another for a publisher. In the latter case, the pub/sub system guarantees the requirements imposed by the publisher on that information (such as lifetime of the information, information priority etc.). Therefore in the next two paragraph we consider separately the notification semantics and the publishing semantics and some of their implications.

Notification semantics. The notification semantics is often left non-specified. While it is clear which will be the information delivered to subscribers by the pub/sub system, it is not clear the condition which defines *if*, *when* and *how many times* an information will be delivered at a subscriber. This is amenable to the decoupling in time of a pub/sub system and to the intrinsic asynchrony and concurrency of publish and subscribe operations. The specification of clear notification semantics that defines during a period of subscription which information will be delivered (e.g. all the information seen by the event brokering system that matches the subscription, only the information that matches the subscription since the arrival of the subscription to the even broker etc.) and how many times (e.g at-most once, exactly once and at-least once) this information will be delivered within the subscription interval is mandatory if one wants to use pub/sub systems as communication paradigm for mission-critical or dependable applications.

Publishing semantics. Indirect communication using queuing of messages between a producer and a consumer leaves to the consumer the management of the removal of the messages from the queue (for example when the consumer fetches a message). In a pub/sub system a subscriber has no right to delete an information as the interaction follows a "pull" style. On the other hand some actor has to define a lifetime of an information, otherwise the size of information stored into a pub/sub system could rapidly overflow. Lifetime of an information is clearly publisher dependent. For example if the publisher issues information on a stock quote, the lifetime of this information could be either a fixed amount of time or till the arrival of the next quote. The pub/sub system is then responsible for guaranteeing such policy.

References

25.1 G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Proceedings of International Conference on Distributed Computing Systems*, 1999.

25.2 A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Notification Service. *ACM Transactions on Computer Systems*, 3(19):332–383, Aug 2001.
25.3 Gryphon Web Site. http://www.research.ibm.com/gryphon/.
25.4 J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and Jr. J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, 2000.
25.5 B. Oki, M. Pfluegel, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensive Distributed Systems. In *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, December 1993.
25.6 A. Rowston and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, 2001.
25.7 A. Rowston, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale notification infrastructure. In *3rd International Workshop on Networked Group Communication (NGC2001)*, 2001.
25.8 B. Segall and D. Arnold. Elvin Has Left the Building: A Publish /Subscribe Notification Service with Quenching. In *Proc. of the 1997 Australian UNIX and Open Systems Users Group Conference*, 1997.
25.9 SIENA Web Site. http://www.cs.colorado.edu/users/carzanig/siena/.
25.10 I. Stoica, D. Adkins, S. Ratnasamy, S. Shenker, S. Surana, and S. Zhuang. Internet indirection infrastructure. In *First International Workshop on Peer-to-Peer Systems*, 2002.
25.11 TIB/Rendezvous Web Site. http://www.rv.tibco.com.
25.12 S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.
25.13 S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.

26. Naming and Integrity: Self-Verifying Data in Peer-to-Peer Systems

Hakim Weatherspoon, Chris Wells, and John D. Kubiatowicz

Computer Science Division, University of California, Berkeley, CA 94720, USA

26.1 Introduction

Today's exponential growth in network bandwidth, storage capacity, and computational resources has inspired a whole new class of distributed, peer-to-peer storage infrastructures . Systems such as Farsite [26.4], Freenet [26.6], Intermemory [26.5], OceanStore [26.10], CFS [26.7], and PAST [26.8] seek to capitalize on the rapid growth of resources to provide inexpensive, highly-available storage without centralized servers. The designers of these systems propose to achieve high availability and long-term durability, in the face of individual component failures, through replication and coding techniques.

Although wide-scale replication has the potential to increase availability and durability, it introduces two important challenges to system architects. First, system architects must increase the number of replicas to achieve high durability for large systems. Second, the increase in the number of replicas increases the bandwidth and storage requirements of the system. Erasure Coding vs. Replication [26.18] showed that systems that employ *erasure-resilient codes* [26.3] have mean time to failures many orders of magnitude higher than replicated systems with similar storage and bandwidth requirements. More importantly, erasure-resilient systems use an order of magnitude less bandwidth and storage to provide similar system durability as replicated systems.

There are consequences to using erasure codes. In particular, erasure codes are more processor intensive to compute than replication. As a result, it is desirable to use complete replication to increase latency performance and erasure codes to increase durability. The challenge is finding synergy between complete replication and erasure coding. Also, maintaining systems built using erasure codes is difficult because erasure coded fragments cannot be verified locally and in isolation, but instead have to either be verified in a group or through higher level objects.

This paper makes the following contributions: First, we discuss some problems with data integrity associated with erasure codes . Second, we contribute a naming technique to allow an erasure encoded document to be self-verified by client and servers.

26.2 Background

Two common methods used to achieve high data durability are replication [26.4, 26.8] and parity schemes such as RAID [26.14]. The former imposes high bandwidth and storage overhead, while the latter fails to provide sufficient robustness for the high rate of failures expected in the wide area.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 142–147, 2003.

An *erasure code* provides redundancy without the overhead of strict replication. Erasure codes divide an object into m *fragments* and recode them into n fragments, where $n > m$. We call $r = \frac{m}{n} < 1$ the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from *any* m fragments. For example, using a $r = \frac{1}{4}$ encoding on a block divides the block into $m = 16$ fragments and encodes the original m fragments into $n = 64$ fragments; increasing the storage cost by a factor of *four*. Some valid erasure codes are Reed-Solomon codes, Tornado codes, and secret-sharing codes . Erasure codes are a superset of replicated and RAID systems. For example, a system that creates four replicas for each block can be described by an ($m = 1, n = 4$) erasure code. RAID level 5 can be described by ($m = 4$, $n = 5$).

Identifying Erasures. When reconstructing information from fragments, we must discard failed or corrupted fragments (called *erasures*). In traditional applications, such as RAID storage servers, failed fragments are identified by failed devices or uncorrectable read errors. In a malicious environment, however, we must be able to prevent adversaries from presenting corrupted blocks as valid. This suggests cryptographic techniques to permit the verification of data fragments; assuming that such a scheme exists, we can utilize any m *correctly verified* fragments to reconstruct a block of data. In the best case, we could start by requesting m fragments, then incrementally requesting more as necessary. Without the ability to identify corrupted fragments directly, we could still request fragments incrementally, but might be forced to try a factorial combination of all returned fragments to find a set of m that reconstructs our data; that is, $\binom{n}{m}$ combinations.

Naming and Verification. A dual issue is the *naming* of data and fragments. Within a trusted LAN storage system, local identifiers consisting of tuples of server, track, and block ID can be used to uniquely identify data to an underlying system. In fact, the inode structure of a typical UNIX file system relies on such data identification techniques. In a distributed and untrusted system, however, some other technique must be used to identify blocks for retrieval and verify that the correct blocks have been returned. In this paper, we will argue that a secure hashing scheme can serve the dual purpose of identifying and verifying both data and fragments. We will illustrate how data in both its fragmented and reconstructed forms can be identified with the *same secure hash value*.

26.3 The Integrity Scheme

In this section, we show how a cryptographically-secure hash, such SHA-1 [26.13][1], can be used to generate a *single, verifiable name* for a piece of data and all of its encoded fragments. We may utilize this name as a query to location services or remote servers, then verify that we have received the proper information simply by recomputing the name from the returned information.

Verification Tree. For each encoded block, we create a binary verification tree [26.12] over its fragments and data as shown in Figure 26.1.a. The scheme works as follows: We produce a hash over each fragment, concatenate the corresponding hash with a sibling

[1] Other cryptographically-secure hashing algorithms will work as well.

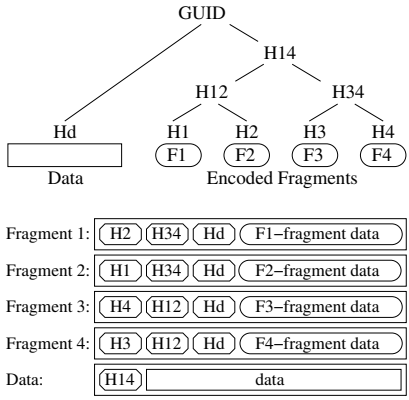

Fig. 26.1. A *Verification Tree:* is a hierarchical hash over the fragments and data of a block. The top-most hash is the block's *GUID*. (b) *Verification Fragments:* hashes required to verify the integrity of a fragment.

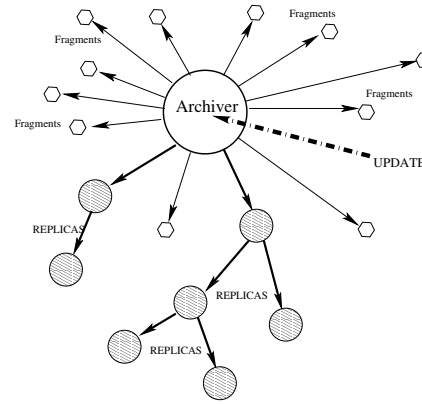

Fig. 26.2. *Hybrid Update Architecture:* Updates are sent to a central "Archiver", which produces archival fragments at the same time that it updates live replicas. Clients can achieve low-latency read access by utilizing replicas directly.

hash and hashing again to produce a higher level hash, *etc.*. We continue until we reach a topmost hash (H14 in the figure). This hash is concatenated with a hash of the data, then hashed one final time to produce a *globally-unique identifier (GUID)*. The GUID is a permanent pointer that serves the dual purpose of identifying and verifying a block. Figure 26.1.b shows the contents of each *verification fragment*. We store with each fragment all of the sibling hashes to the topmost hash, a total of $(\log n) + 1$ hashes, where n is the number of fragments.

Verification. On receiving a fragment for re-coalescing (i.e. reconstructing a block), a client verifies the fragment by hashing over the data of the fragment, concatenating that hash with the sibling hash stored in the fragment, hashing over the concatenation, and continuing this algorithm to compute a topmost hash. If the final hash matches the GUID for the block, then the fragment has been verified; otherwise, the fragment is corrupt and should be discarded. Should the infrastructure return a complete data block instead of fragments (say, from a *cache*), we can verify this by concatenating the hash of the data with the top hash of the fragment hash tree (hash H14 in Figure 26.1) to obtain the GUID. Data supplemented with hashes may be considered *self-verifying*.

More Complex Objects. We can create more complex objects by constructing hierarchies, i.e. placing GUIDs into blocks and encoding them. The resulting structure is a tree of blocks, with original data at the leaves. The GUID of the topmost block serves much the same purpose as an inode in a file system, and is a verifiable name for the whole complex object. We verify an individual data block (a leaf) by verifying all of the blocks on the path from the root to the leaf. Although composed of many blocks, such a complex object is immune to substitution attacks because the integrity and position of each block can be checked by verifying hashes. Complex objects can serve in a variety of rolls, such as documents, directories, logs, etc.

26.4 Discussion

Self-verifying data is important for untrusted wide area storage systems. Specifically, self-verifying data enhances location-independent routing infrastructures (such as, CAN [26.15], Chord [26.16], Pastry [26.8], and Tapestry [26.19]) by cryptographically binding the *name* of objects to their *content*. Consequently, any node that has a *cached* copy of an object or piece of an object can return it on a query; the receiving node does not have to trust the responder, merely check that the returned data is correct. In the following, we explore additional issues with erasure-coding data.

Human-Readable Name Resolution. Up to this point in the document, we have used the word "name" and "GUID" interchangeably. Of course, what people typically call a "name" is somewhat different: a human-readable ASCII string. ASCII names can be accommodated easily by constructing objects that serve as *directories* or *indexes*. These objects can map human readable names into GUIDs. By constructing a hierarchy of directory objects, we easily recover a hierarchical name-space such as present in most file systems. The GUID of a top-level directory becomes the *root inode*, with all self-verifying objects available along a path from this root.

Mutable Data. One obvious issue with the scheme presented in this paper is that data is fundamentally *read-only*, since we compute the name of an object from its contents; if the contents change, then the name will change, or alternatively, a new object is formed. This latter viewpoint is essentially *versioning* [26.17], namely the idea that every change creates a new and independent version[2]. As a result, any use of this verification scheme for writable data must be supplemented with some technique to associate a fixed name with a changing set of version GUIDs. Unfortunately, this binding can no longer be verified via hashing, but must instead involve other cryptographic techniques such as signatures. This is the approach taken in OceanStore [26.10].

Encoding Overhead. Each client in an erasure-resilient system sends messages to a larger number of *distinct* servers than in a replicated system. Further, the erasure-resilient system sends smaller "logical" blocks to servers than the replicated system. Both of these issues could be considered enough of a liability to outweigh the results of the last section. However, we do not view it this way. First, we assume that the storage servers are utilized by a number of clients; this means that the additional servers are simply spread over a larger client base. Second, we assume intelligent buffering and message aggregation; that is, temporarily storying fragments in memory and writing them to disk in clusters. Although the outgoing fragments are "smaller", we simply aggregate them together into larger messages and larger disk blocks, thereby nullifying the consequences of fragment size. These assumptions are implicit in the exploration via metrics of bandwidth and disk blocks in [26.18].

Retrieval Latency. Another concern about erasure-resilient systems is that the time and server overhead to perform a read has increased, since multiple servers must be contacted to read a single block. The simplest answer to such a concern is that mechanisms for *durability* should be separated from mechanisms for *latency reduction* . Consequently,

[2] Whether all past versions are kept around is an orthogonal issue.

we assume that erasure-resilient coding will be utilized for durability, while replicas (i.e. caching) will be utilized for latency reduction. Note that replicas utilized for caching are *soft-state* and can be constructed and destroyed as necessary to meet the needs of temporal locality. Further, prefetching can be used to reconstruct replicas from fragments in advance of their use. Such a hybrid architecture is illustrated in Figure 26.2. This is similar to what is provided by OceanStore [26.10].

26.5 Related and Future Directions

In this paper, we described the availability and durability gains provided by an erasure-resilient system. More importantly, we contributed a naming technique to allow an erasure encoded document to be self-verified.

Other systems have used Merkle trees [26.12] to create chains of signatures [26.11] that prove a document has been signed in the past. Also, other systems have created verifiable documents that use erasure resilient codes. [26.9] would first hash the document then erasure encode the hash itself. [26.1] erasure encoded the document and then hashed each fragment. The difference is that the scheme provided in this paper allows all documents and fragments to be verified locally without requiring extra components.

The idea of a global-scale, distributed, persistent storage infrastructure was first motivated by Ross Anderson in his proposal for the Eternity Service [26.2]. A discussion of the durability gained from building a system from erasure codes first appeared in Intermemory [26.5]. The authors describe how their technique increases an object's resilience to node failure, but the system is targeted for a trusted environment so does not include a checksum.

Some open research issues deal with data structures for documents that take advantage of the peer-to-peer networks and self-verifying data. That is, replicating the document as whole, sequence of bytes, higher level structure, such as a B-tree, etc. Another issue with Peer-to-peer storage infrastructures is that they will fail if no repair mechanisms are in place. Self-verifying documents lend themselves well to distributed repair techniques. That is, the integrity of a replica or fragment can be checked locally.

References

26.1 N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern. Scalable secure storage when half the system is faulty. In *International Colloquium on Automata, Languages and Programming*, pages 576–587, 2000.

26.2 R. Anderson. The eternity service. In *Proc. of Pragocrypt*, 1996.

26.3 J. Bloemer at el. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, ICSI, Berkeley, CA, 1995.

26.4 W. Bolosky et al. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of Sigmetrics*, June 2000.

26.5 Y. Chen at el. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE*, pages 485–495, Feb. 1996.

26.6 I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.

26.7 F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, October 2001.
26.8 P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, 2001.
26.9 H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc. of PODC Symp.*, pages 207–218, 1993.
26.10 J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.
26.11 P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proc. of USENIX File and Storage Technologies FAST*, 2002.
26.12 R. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378. Springer-Verlag, 1988.
26.13 NIST. FIPS 186 digital signature standard. May 1994.
26.14 D. Patterson, G. Gibson, and R. Katz. The case for raid: Redundant arrays of inexpensive disks, May 1988.
26.15 S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of SIGCOMM*. ACM, August 2001.
26.16 I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*. ACM, August 2001.
26.17 M. Stonebraker. The design of the Postgres storage system. In *Proc. of Intl. Conf. on VLDB*, Sept. 1987.
26.18 H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, Mar. 2002.
26.19 B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.

27. Spread Spectrum Storage with Mnemosyne

Steven Hand[1] and Timothy Roscoe[2]

[1] University of Cambridge Computer Laboratory, Cambridge CB3 0FD, UK
steven.hand@cl.cam.ac.uk
[2] Intel Research, 2150 Shattuck Avenue, Berkeley, CA 94704, USA
troscoe@intel-research.net

27.1 Introduction

Redundancy in distributed systems is used to provide increased performance and reliability by techniques such as striping and mirroring [27.1], fast fail-over [27.2], and byzantine fault-tolerance [27.3]. These schemes are oriented toward collections of machines and devices which are fairly small (a few hundred at most) relative to modern wide-area distributed systems – particularly 'peer-to-peer' systems – which may have particpant nodes numbering in the hundreds of thousands.

Indeed, several research efforts are building Internet-scale storage systems [27.4, 27.5, 27.6, 27.7]. In most cases, replication and/or an erasure code is used over a distributed hash table scheme such as [27.8, 27.9, 27.10, 27.11]. This results in self-organising distributed file storage that provides high availability in the face of node failure or network partition. Other systems have used distribution and self-organisation to provide anonymity of access and prevent censorship [27.12, 27.13, 27.14, 27.15, 27.16].

In the following we introduce *spread spectrum computing* as a new term for an old idea. In spread-spectrum computing a subset of a large number of distributed resources are selected according to some keyed pseudo-random process, using redundancy to remove the need to explicitly arbitrate usage between independent users. Although the selection is decentralized, if the candidate set is large enough and the pseudo-random procedure fairly uniform, we can expect relatively good load balancing. If the keys are good enough, the set of resources used by any particular client should be unpredictable and hence difficult to attack.

Mnemosyne [27.17] is a spread spectrum storage system which takes advantage of the widespread availability and low cost of network bandwidth and disk space. The system comprises servers that provide unreliable block storage, and clients which write and read blocks to and from the servers. Although our original motivation for the system was to provide practical steganographic storage, the design has a number of other benefits.

Firstly, it places all responsibility for higher-level functionality (such as a filing system) in the client. Hence each individual user can choose their own trade-offs in terms of cost, reliability availability and privacy. Secondly, tolerance to collisions effectively provides "soft capacity" – the total amount of data which may be stored in the system is shared automatically between the number of users. Thirdly, the extremely simple block storage model lends itself to a commercial deployment in which charging is done for write transactions rather than for storage space [27.18]. In addition to operational simplicity, this approach mitigates against certain denial of service attacks.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 148–152, 2003.

In the remainder of this short paper we first provide an overview of how Mnemosyne operates, and then discuss how the principles used in Mnemosyne relate to the wider notion of spread spectrum computing.

27.2 Mnemosyne: Operational Overview

Mnemosyne was designed as a steganographic storage system; that is, a system in which users who do not have the required key not only are unable to read the contents of files stored under that key (as with a conventional encrypting file system), but furthermore are unable to determine the existence of files stored under that key.

This leads to an interesting property: since users cannot know anything about the location of file blocks stored by other users, it is always possible for them to unwittingly overwrite them; the existence of a file allocation table or list of in-use blocks defeats the steganographic properties of the system. Instead, Mnemosyne uses redundancy in the form of erasure codes to prevent file data being lost due to the write activity of other users.

The process by which a user of Mnemosyne stores a vector of bytes in the system can be broken down into four phases: dispersal, encryption, location, and distribution, which we describe in turn.

Dispersal: In the dispersal phase, the data is encoded to make it robust in the face of block losses. We use Rabin's Information Dispersal Algorithm [27.19] in the field $GF(2^{16})$ to transform n blocks of data into $m > n$ blocks, any n of which suffice to recover the original data. In our prototype, for example, file data is by default treated as chunks of $n = 32$ blocks and each chunk transformed into $m = 3n = 96$ blocks. There are clearly trade-offs involved in the choice of m and n: we investigate these in detail in [27.18].

Encryption: The dispersed blocks from the previous step are now encrypted under the user's key K. The purpose of this is twofold: firstly for security and privacy, but secondly for authenticity. This is necessary since blocks may be overwritten without notice by other users at any time. Hence we need a mechanism by which a user may determine whether a block subsequently retrieved from the network is really the one originally written. We use the AES algorithm in OCB mode [27.20] to provide security and a 16-byte MAC in one step.

Location: Mnemosyne achieves its security properties by storing encrypted data blocks in pseudo-random (and, to an adversary, unpredictable) locations in a large virtual network store, which is then mapped onto distributed physical storage devices. The locations of the encrypted blocks making up a data set are determined by a sequence of 256-bit values obtained by successively hashing an initial value which depends ultimately on the filename and the user's key K.

Distribution: The sequence of 256-bit location identifiers from the previous step is finally mapped onto physical storage using a peer-to-peer network of storage nodes, each of which holds a fixed-size physical block store.

For each block to be stored, both the node identifier and the block offset within the block store are derived from the corresponding location identifier. The top 160 bits of this identifier are used to as a node identifier in a Tapestry [27.11] network and a randomly selected Tapestry node is asked to route the block to the "surrogate" node for the 160-bit node identifier. The next 20 bits of the location id are then used as a block number in a 1GB block store.

The block store at each node supports only the following two operations:

– **putBlock**(*blockid*, *data*)
– **getBlock**(*blockid*) $\rightarrow$ *data*

Note that the block storage nodes themselves need perform no authentication, encryption, access checking, or block allocation to ensure correct functioning of the system, though they might for billing purposes. Indeed, a block store may ignore the above operations entirely: as long as sufficiently many block stores implement the operations faithfully, users' data can be recovered.

27.2.1 Implementation

A working implementation of Mnemosyne for Linux exists. The client is written in C and C++, using freely available reference implementations of SHA-256 and AES-OCB. It provides a command-line interface with operations for key management, creating and listing directories, and copying files between Mnemosyne and the Unix file system. A simple block protocol over UDP is used for communication with block servers. These are implemented in Java and run on Tapestry [27.11] nodes. Performance is plausible: we can copy files into Mnemosyne at 80 kilobytes per second, and read them at 160 kilobytes per second. A principal limiting factor in both cases is our (unoptimised) implementation of matrices over $GF(2^{16})$.

In [27.17] we describe one implementation of a per-user filing system over the data storage and retrieval procedures described above. The filing system uses directories and inodes to simplify the management of keys and initial hash values, and also handles versioning of files, necessary since data is never actually deleted from Mnemosyne, but rather decays over time. As noted earlier, the choice of filing system and parameters is completely under the control of the user.

27.3 Future Directions

Mnemosyne is an illustrative example of spread spectrum computing in the area of storage. A typical file server implements a multi-user file system, explicitly allocating storage blocks to files and metadata. The server ensures the consistency of the file system metadata (including which blocks belong to which files), keeps storage quotas, and enforces security by preventing unauthorised users gaining access to files. Storage-Area Networks (SANs) and their recent extensions over IP networks present a simpler abstraction of storage: authorised users are now allocated virtual block devices which they manage themselves, an approach which moves filing system metadata to the client

side of the interface. The SAN provides storage allocation between users and ensures that unauthorised users cannot write to a given block device.

Mnemosyne stands in stark contrast to these two schemes: block servers enforce no boundaries between users at all. Indeed, they do not need to know the identity of a user writing a block to ensure correct operation of the system. The block servers perform no arbitration whatsoever between users, clients or requests, other than to serialize read and write operations to the disks. All mechanisms for effective sharing of the global storage medium are dispersed among the clients.

This is what we mean by *spread spectrum storage*, by analogy with the use of bandwidth in many wireless communcation systems, including 802.11. Rendundancy, error correction, and encryption distributed among clients are used in place of explicit resource arbitration and allocation centralised in servers.

We feel spread spectrum techniques like Mnemosyne can be appropriate for very large, global-scale applications where central abitration is undesirable for a variety of reasons: scalability, reliability, and the need for a federation of service providers with no controlling authority. For example, we can imagine tackling the co-allocation problem in meta-clusters by essentially dispatching work to m candidate nodes in parallel in the expectation that at least some n of them will succeed. Any node may reject an incoming request for any reason (e.g. overload, security policy). Providing at least n accept the request, the overall execution will complete correctly.

Achieving a practical solution here involves future work addressing the design of redundantly encoded parallel algorithms, investigating the efficiency of coding/partition functions, developing fuzzy distribution protocols and providing programming language support.

References

27.1 Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless Network File Systems. In *Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

27.2 Fernando Pedone and Svend Frolund. Pronto: A Fast Failover Mechanism for Off-the-Shelf Commercial Databases. Technical Report HPL-2000-96, HP Laboratories, July 2000.

27.3 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, Usenix Association*, New Orleans, LA, USA, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

27.4 F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Banff, Canada.*, October 2001.

27.5 Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII). Schloss Elmau, Germany*, May 2001.

27.6 John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

27.7 Tim D. Moreton, Ian A. Pratt, and Timothy L. Harris. Storage, Mutability and Naming in *Pasta*. In *Proceedings of the International Workshop on Peer-to-Peer Computing at Networking 2002, Pisa, Italy.*, May 2002.

27.8 S Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM 2001, San Diego, California, USA.*, August 2001.

27.9 I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001, San Diego, California, USA.*, August 2001.

27.10 Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Internation Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany*, November 2001.

27.11 Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2000.

27.12 D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.

27.13 Ross Anderson. The Eternity Service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology (PRAGOCRYPT'96)*. CTU Publishing House, Prague, 1996.

27.14 Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

27.15 Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, July 2000.

27.16 Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceeding of the 9th USENIX Security Symposium*, pages 59–72, August 2000.

27.17 Steven Hand and Timothy Roscoe. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems, Boston, MA*, March 2002.

27.18 Timothy Roscoe and Steven Hand. Transaction-based Charging in Mnemosyne: a Peer-to-Peer Steganographic Storage System. In *Proceedings of the International Workshop on Peer-to-Peer Computing at Networking 2002, Pisa, Italy.*, May 2002.

27.19 M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Communications of the ACM*, 36(2):335–348, April 1989.

27.20 Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *Eighth ACM Conference on Computer and Communications Security (CCS-8)*. ACM Press, August 2001.

28. Replication Strategies for Highly Available Peer-to-Peer Storage

Ranjita Bhagwan[1], David Moore[2], Stefan Savage[2], and Geoffrey M. Voelker[2]

[1] Department of Electrical and Computer Engineering, University of California, San Diego
[2] Department of Computer Science and Engineering, University of California, San Diego

28.1 Introduction

In the past few years, peer-to-peer networks have become an extremely popular mechanism for large-scale content sharing. Unlike traditional client-server applications, which centralize the management of data in a few highly reliable servers, peer-to-peer systems distribute the burden of data storage, computation, communications and administration among thousands of individual client workstations. While the popularity of this approach, exemplified by systems such as Gnutella [28.3], was driven by the popularity of unrestricted music distribution, newer work has expanded the potential application base to generalized distributed file systems [28.1, 28.4], persistent anonymous publishing [28.5], as well as support for high-quality video distribution [28.2]. The wide-spread attraction of the peer-to-peer model arises primarily from its potential for both low-cost scalability and enhanced availability. Ideally a peer-to-peer system could efficiently multiplex the resources and connectivity of its workstations across all of its users while at the same time protecting its users from transient or persistent failures in a subset of its components.

However, these goals are not trivially engineered. First-generation peer-to-peer systems, such as Gnutella, scaled poorly due to the overhead in locating content within the network. Consequently, developing efficient lookup algorithms has consumed most of the recent academic work in this area [28.9, 28.11]. The challenges in providing high availability to such systems is more poorly understood and only now being studied. In particular, unlike traditional distributed systems, the individual components of a peer-to-peer system experience an order of magnitude worse availability – individually administered workstations may be turned on and off, join and leave the system, have intermittent connectivity, and are constructed from low-cost low-reliability components. One recent study of a popular peer-to-peer file sharing system found that the majority of peers had application-level availability rates of under 20 percent [28.8].

As a result, all peer-to-peer systems must employ some form of replication to provide acceptable service to their users. In systems such as Gnutella, this replication occurs implicitly as each file downloaded by a user is implicitly replicated at the user's workstation. However, since these systems do not explicitly manage replication or mask failures, the availability of an object is fundamentally linked to its popularity and users have to repeatedly access different replicas until they find one on an available host. Next-generation peer-to-peer storage systems, such as the Cooperative File System (CFS) [28.1], recog-

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 153–158, 2003.

nize the need to mask failures from the user and implement a basic replication strategy that is independent of the user workload.

While most peer-to-peer systems employ some form of data redundancy to cope with failure, these solutions are not well-matched to the underlying host failure distribution or the level of availability desired by users. Consequently, it remains unclear what availability guarantees can be made using existing systems, or conversely how to best achieve a desired level of availability using the mechanisms available.

In our work we are exploring replication strategy design trade-offs along several interdependent axes: Replication granularity, replica placement, and application characteristics, each of which we address in subsequent sections. The closest analog to our work is that of Weatherspoon and Kubiatowicz [28.10] who compare the availability provided by erasure coding and whole file replication under particular failure assumptions. The most critical differences between this work and our own revolve around the failure model. In particular, Weatherspoon and Kubiatowicz focus on disk failure as the dominant factor in data availability and consequently miss the distinction between short and long time scales that is critical to deployed peer-to-peer systems. Consequently, their model is likely to overestimate true file availability in this environment.

28.2 Replica Granularity

Systems like Gnutella employ whole file replication: files are replicated among many hosts in the system based upon which nodes download those files. Whole file replication is simple to implement and has a low state cost – it must only maintain state proportional to the number of replicas. However, the cost of replicating entire files in one operation can be cumbersome in both space and time, particularly for systems that support applications with large objects (e.g., audio, video, software distribution).

Block-level replication divides each file object into an ordered sequence of fixed-size blocks. This allows large files to be spread across many peers even if the whole file is larger than what any single peer is able to store. However, downloading an object requires that enough hosts storing block replicas are available to reconstruct the entire object at the time the object is requested. If any one replicated block is unavailable, the object is unavailable. For example, measurements of the CFS system using six block-level replicas show that when 50 percent of hosts fail the probability of a block being unavailable is less than two percent [28.1]. However, if an object consists of 8 blocks then the expected availability for the *entire object* will be less than 15 percent. This dependency is one of the motivating factors for the use of erasure codes with blocking replication.

Erasure codes (EC), such as Reed-Solomon [28.7], provide the property that a set of k original blocks can be reconstructed from any l coded blocks taken from a set of ek coded blocks (where l is typically close to k, and e is typically a small constant). The addition of EC to block-level replication provides two advantages. First, it can dramatically improve overall availability since the increased intra-object redundancy can tolerate the loss of many individual blocks without compromising the availability of the whole file. Second, the ability to reconstruct an object from many distinct subsets of EC blocks, permits a low-overhead randomized lookup implementation that is competitive in state

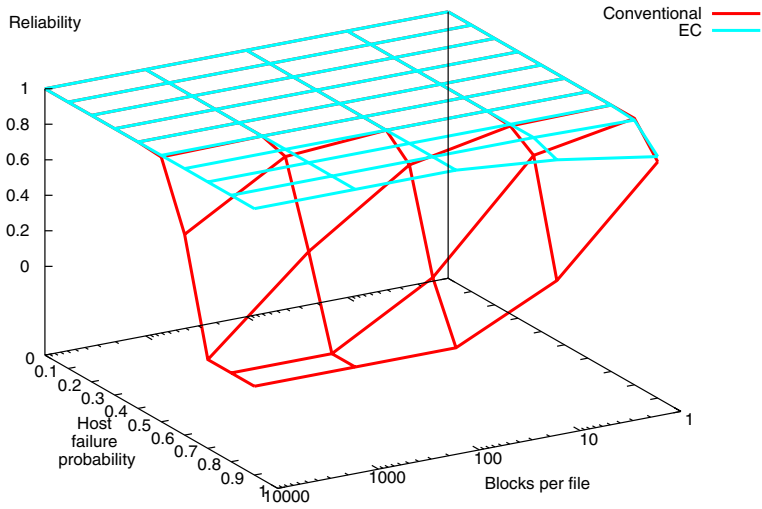

Fig. 28.1. Reliability as a function of the number of blocks per file and host failure probability.

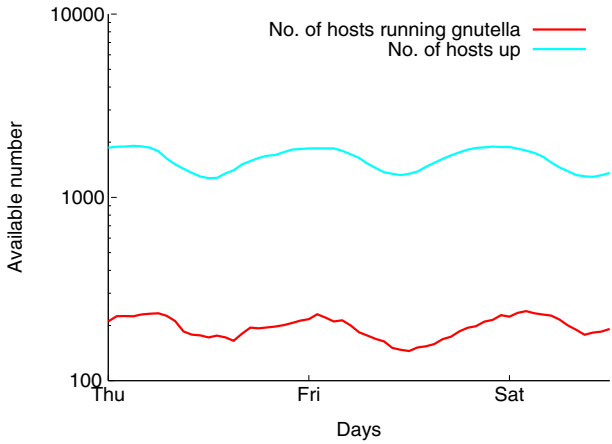

Fig. 28.2. Diurnal patterns in the number of probed hosts available on the network, as well as in the number of these hosts running Gnutella.

with whole-file replication. Rather than maintain the location of every replica for each block, a system using EC blocks can simply track the location of each peer holding *any* block belonging to the object.

28.2.1 Replication and Host Failure

As an initial experiment to explore the trade-off between conventional block replication and block replication with erasure codes, we simulate the replication and distribution of a single file in a idealized system of N hosts. The file is divided into fixed-size blocks and replicated. Replicated blocks are randomly assigned to hosts, each of which has the same

uniform failure probability. We then simulate random host failure and determine whether the file is still recoverable from the system assuming an ideal lookup system and perfect network conditions. We repeat this experiment 100 times and measure the fraction of times that the file is completely recoverable. For the purposes of this experiment, we define this fraction as the reliability of the system.

We use the term storage redundancy to refer to the amount of storage a replication technique uses. In the conventional case, storage redundancy is simply the number of replicas of the file. For the erasure coded case, redundancy is introduced not only by replication, but also by the encoding process. In this case, storage redundancy is the number of replicas times the encoding redundancy. A file consisting of k blocks is encoded into ek blocks, where $e > 1$ is what we call the encoding redundancy (or *stretch factor*). In our simulations, $e = 2$. Figure 28.1 shows simulation results of the idealized system as a function of blocks per file k and host failure probability, for a storage redundancy of 20. From the figure, we see that for host failure probabilities less than 0.5 both replication schemes achieve high reliability. For higher host failure probabilities, the reliabilities of the two techniques diverge. Also, as number of blocks per file increases, the two techniques quickly diverge in reliability. Erasure coded blocks actually increases reliability with larger number of blocks, while conventional block replication reliability decreases drastically. So using conventional blocking and scattering those blocks across a large number of relatively unreliable hosts makes the system less reliable. However, erasure coded replication is able to achieve excellent reliability even when the underlying hosts are quite unreliable.

28.3 Replica Placement

For the purposes of file availability, peer-to-peer systems should not ignore the availability characteristics of the underlying workstations and networks on which they are implemented. In particular, systems should recognize that there is wide variability in the availability of hosts in the system. Saroiu and Gribble found that fewer than 20 percent of Gnutella's peer systems had network-level availability in excess of 95 percent [28.8], while over half of the remainder had availability under 20 percent. Given such a wide variability, the system should not place replicas blindly: more replicas are required when placing on hosts with low availability, and fewer on highly available hosts. Moreover, under many predictable circumstances, peer failures may be correlated. We performed a study of the Gnutella network similar to Saroiu and Gribble's work, and found that the number of hosts running Gnutella is well correlated with time of day, and shows a diurnal pattern, as shown in Figure 28.2. Due to this dependence between host failures based on time-of-day, one should be careful not to place multiple replicas of a file on hosts that are likely to fail at the same time. This motivates a study of the degree of dependence between host failures in such a dynamic environment. Also, independent investigations of client workstation availability has shown strong time-zone specific diurnal patterns associated with work patterns [28.6]. As a consequence, placing replicas in out-of-phase time zones may be a sound replication strategy.

28.4 Application Characteristics

Peer-to-peer systems are being used for a wide range of applications, including music and video sharing and wide-area file systems. Larger objects such as video files take longer to replicate and are more cumbersome to manage as a whole, and naturally motivate the use of block-level replication. On the other hand, when conventional blocking is used for large objects, the reliability of the system depends upon a large number of hosts being available. Consequently, an object's availability is inversely related to its size. So a trade-off requires to be made when designing replication strategies for applications dealing with large file sizes.

The relationship between when data are requested and the time at which they must be delivered creates several opportunities for optimization based on application characteristics. For example, traditional file system applications usually require an entire file object to be delivered to the application buffer cache before the application can make forward progress. However, the order in which this data is delivered and variations in overall delay rarely have an impact. In contrast, streaming media workloads require that only the data surrounding the current playout point be available, but this particular data must be delivered on time for the application to operate correctly.

28.5 Summary

We are investigating strategies for using replication to design and implement highly available storage systems on highly unavailable peer-to-peer hosts. In addition, we are investigating how application properties such as object size, timeliness of delivery, and workload properties such as object popularity should influence replication strategies. We are also investigating how replica placement policies can be tuned to compensate for diurnal patterns in host availability, and to take advantage of out-of-phase time zones. Eventually, we plan to implement our results in a prototype system for practical evaluation.

References

28.1 F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)* , 2001.

28.2 edonkey homepage, http://edonkey2000.com.

28.3 Gnutella homepage, http://www.gnutella.com.

28.4 J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, 2000.

28.5 A. D. R. Marc Waldman and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

28.6 D. Moore. Caida analysis of code-red, http://www.caida.org/analysis/ security/code-red/, 2001.

28.7 V. Pless. *Introduction to the theory of error-correcting codes*. John Wiley and Sons, 3rd edition, 1998.
28.8 S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, 2002.
28.9 I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, 2001.
28.10 H. Weatherspoon and J. Kubiatowicz. Erasure coding v/s replication: a quantitative approach. In *Proceedings of the First International Workshop on Peer-to-peer Systems*, 2002.
28.11 B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB-CSD-01-1141, U. C. Berkeley, April 2000.

29. A Data-Centric Approach for Scalable State Machine Replication*

Gregory Chockler[1,2], Dahlia Malkhi[1], and Danny Dolev[1]

[1] School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel 91904
{grishac,dalia,dolev}@cs.huji.ac.il
[2] IBM Haifa Research Labs (Tel-Aviv Annex)

29.1 Introduction

Data replication is a key design principle for achieving reliability, high-availability, survivability and load balancing in distributed computing systems. The common denominator of all existing replication systems is the need to keep replicas consistent. The main paradigm for supporting replicated data is *active replication*, in which replicas execute the same sequence of methods on the object in order to remain consistent. This paradigm led to the definition of *State Machine Replication* (SMR) [29.8, 29.13]. The necessary building block of SMR is an engine that delivers operations at each site in the same total order without gaps, thus keeping the replica states consistent.

Traditionally, existing SMR implementations follow a *process-centric* approach in which processes actively participate in active replication protocols. These implementations are typically structured as a peer group of server processes that employ group communication services for reliable totally ordered multicast and group membership maintenance. The main advantage of this approach is that during stability periods, work within a group is highly efficient. However, when failures occur and are detected the system needs to reconfigure. This requires solving agreement on group membership changes. Moreover, membership maintenance implies that participants need to constantly monitor each other. Consequently, group communication based systems scale poorly as the group size and/or its geographical span increases. Additionally, due to the high cost of configuration changes, these solutions are not suitable for highly dynamic environments.

In contrast, we advocate the use of a *data-centric* replication paradigm in order to alleviate the scalability problems of the process-centric approach. The main idea underlying the data-centric paradigm is the separation of the replication control and the replica's state. This separation is enforced through a two-tier architecture consisting of a *storage tier* whose responsibility is to provide persistent storage services for the object replicas, and a *client tier* whose responsibility is to carry out the replication support protocols. The storage tier is comprised of logical storage elements which in practice can range from network-attached disks to full-scale servers. The client tier utilizes the storage tier for communication and data sharing thus effectively emulating a shared memory environment.

The data-centric approach promotes fundamentally different replication solutions: First, it perfectly fits today's state-of-the-art Storage Area Network (SAN) environments,

* This work was supported in part by the Israeli Ministry of Science grant #1230-3-01.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 159–163, 2003.

where disks are directly attached to high speed networks, usually Fibre Channel, that are accessible to clients. Moreover, due to the lack of broadcast support by Fibre Channel networks, direct multicast communication among processes, as mandated by the process-centric paradigm, is not easily supportable in SAN environments. Second, it simplifies the system deployment and management as the only deployment pre-requisite is the existence of an infrastructure of storage elements which can be completely dynamic. Also, there is no need to deploy any non-standard communication layers and/or tools (such as group communication). In fact, all the communication can be carried out over a standard RPC-based middleware such as Java RMI or CORBA. Third, the storage elements do not need to communicate with one another nor they need to monitor each other and reconfigure upon failures. This reduces the cost of fault-management and increases the system scalability. The replication groups can be created on-the-fly by clients simply writing the initial object state and code to some pre-defined collection of the storage elements. Fault-tolerance can be achieved by means of quorum replication as it is done in the Fleet survivable object repository [29.11].

Finally, the data-centric approach has a potential for supporting replication in highly dynamic environments where the replicas are accessed by an unlimited number of possibly faulty clients whose identities are not known in advance. In this paper, we first mention the results introduced in [29.3], which extend the Paxos approach of Lamport [29.9] to such environments. These results provide universal object emulation in a very general shared memory model, in which both processes and memory objects can be faulty, the number of clients that can access the memory is unlimited, and the client identities are not known. We then outline future directions for research within the data-centric framework.

29.2 SMR in Data Centric Environments

The Paxos algorithm of Lamport [29.9] is one of the techniques used to implement operation ordering for SMR. Numerous flavors of Paxos that adapt it for various settings and environments have been described in the literature. At the core of Paxos is a Consensus algorithm called *Synod*. Since Consensus is unsolvable in asynchronous systems with failures [29.5], the Synod protocol, while guaranteeing always to be safe, ensures progress when the system is stable so that an accurate leader election is possible. In order to guarantee safety even during instability periods, the Synod algorithm employs a 3-phase commit like protocol, where unique *ballots* are used to prevent multiple leaders from committing possibly inconsistent values, and to safely choose a possible decision value during the recovery phase.

Most Paxos implementations were designed for process-centric environments, where the replicas in addition to being data holders, also actively participate in the ordering protocol. Recently, Gafni and Lamport proposed a protocol for supporting SMR in the shared memory model [29.6] emulated by the SAN environment. Their protocol is run by clients that use network-attached commodity disks as read/write shared memory. The protocol assumes that up to a minority of the disks can fail by crashing. In Disk Paxos, each disk stores an array with an entry for each participating client. Each client can read the entries of all the clients but can write only its own entry. Each of the two Paxos

phases is simulated by writing a ballot to the process entry at a majority of disks, and then reading other process entries from a majority of disks to determine whether the ballot has succeeded.

A fundamental limitation of Disk Paxos, which is inherited from all known variants of the Paxos protocol, is its inherent dependence on a priori knowledge of the number and the identities of all potential clients. The consequences of this limitation are twofold: First, it makes the protocol inappropriate for deployment in dynamic environments, where network disks are accessed from both static desktop computers and mobile devices (e.g., PDAs and notebooks computers). Second, even in stationary clusters, it poses scalability and management problems, since in order for new clients to gain access to the disks, they should either forward their requests to a dedicated server machine, or first undergo a costly join protocol that involves real-time locking [29.6].

In [29.3], we initiated a study of the Paxos algorithm in a very general shared memory model, in which both processes and memory registers can be faulty, the number of clients that can access the memory is unlimited, and the client identities are not known. Since wait-free Consensus is impossible even for two processes in this model [29.7], we augment the system with an unreliable leader oracle, which guarantees that eventually and for a sufficiently long time some process becomes an exclusive leader. Equipped with a leader oracle, Consensus is possible for finitely many processes using read/write registers. However, even with a leader oracle, an infinite number of read/write registers is necessary to implement Consensus among infinitely many clients (see [29.3] for the proof).

Our solution first breaks the Paxos protocol using an abstraction of a shared object called a *ranked register*, which follows a recent deconstruction of Paxos by Boichat et al. in [29.1][1]. The remarkable feature of the ranked register is that while being strong enough to guarantee Consensus safety regardless of the number of participating clients, it is nevertheless weak enough to allow implementations satisfying wait-free termination in runs where any number of clients might fail. Armed with this abstract shared object, we provide a simple implementation of Paxos-like agreement using one reliable shared ranked register that supports infinitely many clients.

Due to its simplicity, a single ranked register can be easily implemented in hardware with the current Application Specific Integrated Circuit (ASIC) technology. Thus, the immediate application of the ranked register would be an improved version of Disk Paxos that supports unmediated concurrent data access by an unlimited number of processes. This would only require augmenting the disk hardware with the ranked register, which would be readily available in Active Disks and in Object Storage Device controllers.

29.3 Future Directions

In contrast to the process-centric approach whose power and limitations are now well understood, the data-centric paradigm poses several unresolved theoretical questions requiring a further study. First, let us take a closer look on the oracle abstraction required

[1] In [29.1], an abstraction called *round-based register* is introduced, which we use but modify its specification.

to guarantee the liveness of the Consensus protocol. Most of the oracle abstractions found in the literature are targeted for message passing systems. An exception is found in [29.10] where the notion of an unreliable failure detector was adapted to the shared memory model (without memory faults). Still, this approach is not easily implementable with infinitely many processes, as it assumes that processes are able to monitor each other which is problematic if the process universe is a priori unknown.

On the other hand, weak synchronization paradigms match the shared memory environment more closely and as demonstrated in [29.4], can be made oblivious to the overall number of clients and their identities. In light of this, an interesting future direction is to identify a class of weak synchronization primitives sufficient for solving Consensus in an asynchronous system, and study their implementability in presence of infinitely many processes.

The next future direction is concerned with tolerating malicious memory failures. Interestingly, this appears to have several non-obvious consequences with respect to the overall number of shared memory objects needed to achieve the desired resilience level. In particular, in contrast to the well-known $3f + 1$ lower bound on the number of processes needed to tolerate up to f Byzantine failures in the message passing model, all the existing wait-free algorithms for asynchronous shared memory model with faults require at least $4f+1$ memory objects to tolerate malicious failures of at most f memory objects (see e.g., [29.11]). Martin et al. show in [29.12] an implementation of an atomic register with $3f + 1$ servers which utilizes enhanced server functionality and increased communication. In particular, in their method clients *subscribe* to servers so as to repeatedly receive updates about the register's state. Further exploration is required to investigate this seeming tradeoff between the resilience on one hand, and the communication cost and memory consumption on the other hand.

Another interesting direction would be to use the ranked register abstraction as a machinery for unifying numerous Consensus implementations found in the literature. Of particular interest is the class of so called *indulgent* Consensus algorithms: i.e., the algorithms designed for asynchronous environments augmented with an unreliable failure-detector. The Synod algorithm of Paxos is an example of such indulgent algorithm. Another example is the revolving-coordinator protocol (e.g., see [29.2]), which is based on a similar principle as Paxos but has the leader election being explicitly coded into the algorithm. One of the benefits of establishing a uniform framework for asynchronous Consensus algorithms will be in a better understanding of how the lower bounds for Consensus in asynchronous message passing model can be matched in its shared memory counterpart.

Finally, on a more practical note, let us take a look on today's SAN based systems. These systems usually employ sophisticated software layers which emulate higher level abstractions such as virtual disks, object stores and file systems over the SAN. These software layers typically manage large volumes of metadata such as directory structure, access permissions, etc., whose availability and consistency are crucial. This mandates using replication to ensure continuous metadata availability despite faults. In this context, our data-centric SMR represents a scalable solution for maintaining metadata replicas in a consistent state. Consequently, integrating data-centric SMR into the SAN software is a challenging future direction of practical importance.

References

29.1 R. Boichat, P. Dutta, S. Frolund and R. Guerraoui. Deconstructing Paxos. *Technical Report DSC ID:200106*, Communication Systems Department (DSC), École Polytechnic Fédérale de Lausanne (EPFL), January 2001.
Available at `http://dscwww.epfl.ch/EN/publications/documents/tr01_006.pdf`.

29.2 T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43(2):225–267, March 1996.

29.3 G. Chockler and D. Malkhi. Active disk Paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, July 2002. To appear.

29.4 G. Chockler, D. Malkhi and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In Proceedings of the *21st International Conference on Distributed Computing Systems*, pages 11-20, April 2001.

29.5 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2):374–382, April 1985.

29.6 E. Gafni and L. Lamport. Disk Paxos. In Proceedings of *14th International Symposium on Distributed Computing (DISC'2000)*, pages 330–344, October 2000.

29.7 P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM* 45(3):451-500, May 1998.

29.8 L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM* 21(7):558–565, July 1978.

29.9 L. Lamport. The Part-time parliament. *ACM Transactions on Computer Systems* 16(2):133–169, May 1998.

29.10 W. K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In Proceedings of the *8th International Workshop on Distributed Algorithms (WDAG)*, Springer-Verlag LNCS 857:280-295, Berlin, 1994.

29.11 D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

29.12 J. P. Martin, L. Alvisi and M. Dahlin. Minimal Byzantine Storage. In Proceedings of *the 16th International Conference on DIStribued Computing (DISC'02)*, pages 311–325, October 2002,

29.13 F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

30. Scaling Optimistic Replication

Marc Shapiro[1] and Yasushi Saito[2]

[1] Microsoft Research, Cambridge, UK
marc.shapiro@acm.org
http://www-sor.inria.fr/~shapiro/
[2] Hewlett Packard Labs, Palo Alto, USA
yasushi_saito@hp.com
http://www.hpl.hp.com/personal/Yasushi_Saito/

30.1 Introduction

Replication improves the performance and availability of sharing information in a large-scale network. Classical, pessimistic replication incurs network access before any access, in order to avoid conflicts and resulting stale reads and lost writes. Pessimistic protocols assume some central locking site or necessitate distributed consensus. The protocols are fragile in the presence of network failures, partitioning, or denial-of-service attacks. They are safe (i.e., stale reads and lost writes do not occur) but at the expense of performance and availability, and they do not scale well.

An optimistic replication (OR) protocol assumes that conflicts are rare. Instead of going to the network, and waiting, to find out whether an access conflicts with a remote site, an OR system lets each site read and write autonomously. The OR system lazily finds whether a conflict has occured and *reconciles*, i.e., recovers from the conflict after the fact. Since it avoids inline network access, it scales better and makes progress even when communication fails. Being offline, reconciliation can use semantic information to avoid false conflicts. OR demonstrates superior fault tolerance and performance. A major drawback of OR is that it is not transparent: the user may observe updates to be aborted or rolled back in order to remove conflicts.

OR applications such as network news [30.5], file or calendar synchronisers [30.1, 30.9] for mobile devices [30.10] and the Concurrent Versioning System (CVS) [30.2] are successful and in daily use.

Beyond a small niche, however, pessimistic systems remain prevalent. In this paper, we try to understand why, and we argue that the success of the above systems is not generalisable. The root of the problem is a fundamental tension between scalability and safety. A scalable system such as Network News is not safe, and safe systems such as CVS rely on pessimistic techniques to ensure consistency. In some specific cases, the pessimistic elements can be made to be static or offline, thus minimising the impact on performance, at the cost of specially-tailored protocols.

30.2 Optimistic Protocols

The most popular use of OR is single-master replication with delayed update propagation. Web proxy caching is an example. Each datum has a central authoritative source

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 164–168, 2003.

where updates are applied; replicas receive updates from the centre after some delay. In the meantime users will have experienced stale reads. Such systems typically do not guarantee consistency between different data items.

A recent survey [30.12] distinguishes two broad categories of multi-master OR systems. A state-based system propagates updates by sending the new value of the datum to all replicas. Conflicts are resolved, without explicit detection, by a syntactic rule. For instance with the Thomas Write Rule (TWR) [30.13] the last writer wins. State-based replication is unsafe in that it allows essentially arbitrary stale reads and lost writes.

In an operation-based system, when a master updates its replica, it records the update operation in a log. To reconcile, a site throws away the current replica state, collects all logs, sorts and merges them, and replays the resulting schedule from the (common) initial state. The sorting and merging function is often time-based [30.7] but semantics-based functions are used as well [30.2, 30.8]. When an unresolvable conflict occurs, one of the operations involved is dropped from the schedule. Stale reads do not occur, and lost updates are replaced by explicit conflict detection and undo/redo execution. In the general case an agreement protocol is needed to uniformly commit an equivalent schedule.

Most OR systems use epidemic propagation [30.3] to ensure that every update eventually reaches every site (liveness). An OR system is safe if: (i) replicas' committed schedule converge, and (ii) the set of actions that are dropped, and the order of execution of the others, is safe according to application semantics. Furthermore systems should scale, i.e., performance, safety and liveness not decrease when the number of objects, the number of masters, or the distance between masters increases.

30.3 Example OR Systems

Network news [30.5] was one of the first OR applications and is probably still the largest-scale in existence. The main operation is creating a news item (posting). Postings can be received in any order, and missing postings are not a problem: the creation operations commute, and the user interface arranges the postings in some sensible order. Thus the network news application has no safety constraints, and has no commitment protocol. Other operations, such creating and deleting newsgroups, are rare and are handled in a best-effort manner.

In CVS, shared files reside on a central repository. A user can edit his local replica of the files. When the user decides to reconcile, CVS computes a "diff" of editing operations between his initial version and his current one, and receives diffs of concurrent updates. CVS merges non-overlapping diffs into the corresponding files; the user must manually resolve updates that do overlap. Software development activities, including editing, compiling, and regression testing, constitute further conflict detection and resolution mechanisms. Finally the user commits a set of diffs that he sends back to the repository. The repository serialises commitments, and ensures that a commit has viewed all previous committed diffs, guaranteeing safety. The commitment protocol is centralised and lock-based, i.e., pessimistic.

In Bayou [30.7], applications log arbitrary high-level operations, and control semantically when an operation is dropped. The shared objects are partitioned into independent

databases; a transaction is restricted to accessing a single database. A primary server for each database centralises the commitment protocol, committing operations in the order they accepted at the home server. We argue that Bayou will not scale because of the centralised commitment and the difficulty of statically partitioning a large number of objects into independent subsets. Despite this, Bayou demonstrates the superiority of OR over pessimistic techniques: Bayou supports a larger set of semantics, and allows a client to make progress even when it cannot communicate with the home server.

Some systems such as TACT and replicated databases partition data dynamically with escrow [30.6, 30.15]. Escrow provides a single process a write-access lease on a subset of the database. If an update transaction commits with all its leasors within the time of all the leases it uses, and only those, it is guaranteed to succeed. Granting an escrow is pessimistic: it improves safety but is sensitive to failures. However escrow is more flexible than locking, as updates might succeed even in the absence of a lease.

30.4 Inherent Limitations of Optimistic Replication

Any OR system must overcome two issues: resolving conflicting updates and commitment. Across successful OR systems, we observe only small variations across the design space, and note that solutions are determined largely by the nature of the application, not by the replication algorithm.

- Manual conflict resolution does not scale. For instance, PDA synchronisers are designed for small numbers of users, of replicas and of shared objects; yet even so prove hard to use.
- Usenet operations are simple and commutative. Network News offers no consistency guarantee. Conflicts and commitment are not issues here.
- File synchronisation systems [30.1, 30.9] work when used at a small scale; when scaling up, people voluntarily partition data to minimize the likelihood of conflicts.
- CVS works because it centralises commits, and because software development has built-in verification mechanisms.
- Systems such as Web proxies are optimistic, but they are single-master systems that preclude write conflicts in the first place.

If an application fits none of the above patterns, we argue that scaling it will be hard.

One issue is frequency of conflicts. Shasha, Gray, and Helland [30.4] show that when an application's data doesn't naturally partition, OR will experience $O(N^2 + 2^L)$ conflicts, where N is the number of nodes and L the number of operations submitted to the system. Another issue is the difficulty of resolving conflicts. The IceCube reconciliation system [30.8] proposes a general-purpose framework for expressing specialised application semantics. It exports clean and simple abstractions, making it easier for application developpers to understand and avoid conflicts, but does not resolve them.

If existing systems are any indication, the only sufficiently general technique for scalability is partitioning data into completely independent subsets. Partitioning can either be static (e.g., ownership) or dynamic (e.g., leases and escrow). The former severely constrains the kind of transactions that can be allowed, and the latter is a pessimistic technique.

30.5 Solutions

In summary, OR is superior to pessimistic replication because clients can make progress even when the network is disconnected and commitment is blocked. Furthermore an OR commitment protocol can run in the background, thus improving performance. However, existing OR systems either do not enforce any correctness constraints, or revert to a pessimistic commitment protocol in order to reach consistency.

Because of physical constraints (latency and failures) and human ones (collaborative work styles), optimistic replication is here to stay. However, a general-purpose, large-scale write-sharing system is likely to rely on centralisation for commitment. Fortunately, previous studies [30.14] show that data that is widely shared tends to be read-only. To build a scalable OR system, we can learn from past successes and limitations:

- If updates can be centralised and data items are completely independent from one another, a single-master algorithm will serve the application well.
- For independent data items with weak safety requirements, the Thomas Write Rule (last writer wins) often works well.
- Otherwise, partition data so that there will never be a dependency between operations in different partitions.
- Failing all of the above, design the application to have a narrow set of mostly-commuting operations. A system such as IceCube will be helpful to to schedule updates and avoid conflicts by exploiting the semantics.

Based in part on the lessons learned here, the authors are engaged in designing large-scale systems for sharing mutable data: the Pangea file system [30.11], and the IceCube reconciliation system driven by object semantics [30.8].

References

30.1 S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, October 1998.

30.2 Per Cederqvist, Roland Pesch, et al. Version management with CVS, 2001. http://www.cvshome.org/docs/manual.

30.3 Richard A. Golding. Design choices for weak-consistency group communication. Technical report, UC Santa Cruz, December 1992.

30.4 Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. Dangers of replication and a solution. In *Int. Conf. on Management of Data*, pages 173–182, Montréal, Canada, June 1996.

30.5 Brian Kantor and Phil Rapsey. RFC977: Network news transfer protocol. http://info.internet.isi.edu/in-notes/rfc/files/rfc977.txt, February 1986.

30.6 Patrick E. O'Neil. The Escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

30.7 Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *16th Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, St. Malo, France, October 1997.

30.8 Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.

30.9 Norman Ramsey and Elod Csirmaz. An algebraic approach to file synchronization. In *9th Int. Symp. on the Foundations of Softw. Eng. (FSE)*, Austria, September 2001.
30.10 Neil Rhodes and Julie McKeehan. *Palm Programming: The Developer's Guide*. O'Reilly, December 1998.
30.11 Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, 2002.
30.12 Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, Hewlett-Packard Laboratories, February 2002.
30.13 Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS)*, 4(2):180–209, June 1979.
30.14 An-I Andy Wang, Peter Reiher, and Rajive Bagrodia. A simulation framework and evaluation for optimistically replicated filing environments. Technical Report CSD-010046, Computer Science Department, University of California, Los Angeles, Los Angeles CA (USA), 2001.
30.15 Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th Symp. on Op. Sys. Design and Implemen. (OSDI)*, pages 305–318, San Diego, CA, USA, October 2000.

Part IV
System Solutions: Challenges and Opportunities in Applications of Distributed Computing Technologies

Our ability to build networked computing platforms continues to outpace our ability to design sophisticated and reliable distributed applications for these platforms. In the development of distributed applications, considerable effort is devoted to making applications resilient to the changes in the distribution requirements and the evolution in the underlying network topologies, and to making the applications robust in the face of typical processor and communication failures. Because of these compound complexities, constructing such systems continues to be challenging. Available middleware frameworks such as those based on CORBA, DCE, and Java provide only limited benefits. Middleware, in many cases, provides functions (e.g., remote procedure and object calls, and multicasts) that are at a lower level of abstraction than those required by complex applications. Even when higher-level functions are available, they often have informal specifications and provide unclear guarantees with respect to performance reliability, security, scalability, and consistency. In particular, it is often difficult to reason about the properties of distributed applications that are built using middleware, and about the compositional semantics of the resulting services and applications.

Among the impediments to qualitative improvements in the practice of distributed system development is the fact that there still exists a communication gap between engineering groups developing systems in a commercial enterprise, and the research groups advancing the scientific state-of-the-art in academic and industrial laboratory settings. The paper by Whetten (page 173) observes that system developers in industry expend substantial effort on interactions with the user communities and stakeholders in general. These activities often have low technical density, while the research community focuses its energy on finding answers to interesting and challenging (almost exclusively) technical questions. This leads to delays in the deployment of relevant research results in practical settings. Additional reasons for such delays is that vendors are concerned with time-to-market thus often pursuing proprietary solutions. On the other hand researchers conduct open investigation that commonly reaches market through standardization channels, incurring substantial time delays as well as introducing new technology in the context of standards that may or may not be broadly accepted.

The success of the world-wide web is in part attributable to the valuable services provided by means of limited yet effective client-server and user-driven interactions. These services are typically provided without precise semantics and performance guarantees. Modern distributed systems are surpassing the utility offered by the client-server approach and require increasing flexibility, scalability, and autonomy offered by the *peer-to-peer* approach. The paper by Gupta and Birman (page 180) deals with large-scale

sensor networks that are subject to severe resource limitation and poor dependability of individual components. The operations in this context include multicast, data aggregation, leader election, and many others. The notion of *holistic operations* encompasses operations involving large parts of a network and that guarantee their service in a scalable and reliable fashion. This approach uses epidemic broadcast, that is a broadcast that relies on gossip, to implement such operations in dynamic and *ad hoc* networks. The research direction based on holistic operation addresses several challenges in large-scale sensor networks.

The dramatic growth of the global networks, and the relentlessly increasing numbers of computing devices and installed software systems, raises the questions of how to ensure the dependability of these pervasive systems. Given the infeasibility of even managing the coordination among these systems and their suppliers, the article of Fetzer and Högstedt (page 186) proposes a very different approach of substantially improving dependability. Authors observe that “Moore’s Law", while anticipating the growth of global computing environments, also predicts the exponential increase in resources, such as computation, storage and networking. The new approach increases the dependability of the individual systems by automatically insulating them from inputs and usage patterns that may cause them to malfunction.

The paper by Malek (page 191) re-examines the challenges associated with an almost uncontrollable growth of the web. The author considers an approach where dependable, efficient and secure solutions for the global computing enterprise are achieved through a balanced interplay of three complementary design directions: acquiring global knowledge within the relevant scope of interest, achieving agreement with cooperating peers in a collaborative setting, and relying on local knowledge where autonomous operation can yield the desired results.

The next three papers consider broad use of distributed computing technologies in modern networked enterprises. The paper by Vogels (page 197) examines the challenges presented by a real-time enterprise and the requirements it poses in the applications of distributed computing technology. The author points out that while one can apply distributed computing know-how, one also needs to understand the business drivers underlying the enterprise, including the requirements of scalability and timing. The solutions to the technological challenges presented by a future global enterprise, and the operational models of the enterprise, will emerge with the participation of all stakeholders. The article by Shrivastava (page 202) delves deeper into the intercorporate interactions that will be required to integrate individual enterprises into a global cooperative enterprise. The author discusses the middleware for inter-enterprise interaction with the focus on enabling trusted interaction that preserves policy rules established by each enterprise. Here autonomy and privacy need to be carefully maintained while enabling global integration. The third paper is by González-Barahone and de-las-Heras-Quirós (page 207) who consider a very different business model, that of open source software development and maintenance. They describe the general issues and challenges in providing peer-to-peer solutions for a distributed open source software enterprise.

The final paper by Bershad and Grimm (page 212) describes an approach for system support of pervasive application. Their framework aims to make easier the development of such applications, and to make more effective the user interactions with these

applications. The framework supports discovery, migration, composition, events, and checkpointing.

To summarize, the papers in Part IV deal with the challenges and opportunities in applying distributed computing technologies in a modern networked enterprise. As we move to global integration of the computing infrastructures of modern enterprises, we need to consider innovative systems solutions that are developed with full understanding of both the technological and business challenges, and with active participation of the research, industrial and business stakeholders.

Alex A. Shvartsman

31. Building a Bridge between Distributed Systems Theory and Commercial Practice

Brian Whetten

Author was most recently Chief Scientist of Talarian Corporation
brian@whetten.net, www.whetten.net

31.1 Introduction

There is a significant chasm that exists between the distributed systems research community and the commercial marketplace. The call for papers for the FuDiCo conference summarized it beautifully.

"The distributed systems research community knows a great deal about reliability and security. Yet, our tools often scale poorly, rarely are seen as practical, and even more rarely have any significant impact on general commercial practice. Nonetheless, the need for distributed systems offering strong guarantees has never been more acute."

There is a major communication paradigm shift presently underway in industry, from client-server computing to the message based computing that has been studied by the research community for so many years. Hence, the opportunity for the distributed research community to positively and dramatically influence the commercial world has never been larger. However, it is not presently succeeding due to this chasm. This makes answering two questions particularly important and urgent. First, why does this chasm exist? Second, what can be done to bridge it?

This paper proposes that there are fundamental structural barriers that underlie the chasm, and attempts to describe two of them. There are two primary paths through which computer science research has in the past been able to influence commercial practice – through the education of commercial developers of proprietary products, and through the participation of academics in standards bodies. In the distributed systems research field, the first path is largely blocked due to a structural *communication gap* that results from the one-way nature of communication between these communities. Researchers communicate their findings to industry, but industry is rarely able to communicate their understanding of customer and system requirements. The second path is largely blocked due to a structural *deployment gap* that currently faces Internet and distributed systems standards.

In the experience of the author, there are a number of research directions that hold significant potential to help bridge this chasm. Two of the most important are *overlay networking* and *contract based object routing*. Architectures for overlay networking

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 173–179, 2003.

involve deploying software infrastructures that duplicate the functionality of the networks they are deployed on top of, at least in concept requiring the modification of the OSI 7-layer networking model. Overlay networking helps address the deployment gap by enabling graceful deployment of new technologies, and provides a path across the communication gap, whereby which academic research can more readily influence commercial products. *Contract based object routing* is a new form of cross-process interface, which enables seamless, incremental bridging of the communication islands created by deployment of proprietary systems and overlay networks, helping bridge the deployment gap. It is the position of this paper that these two innovations are crucial starting points for building a bridge between the theoretical and commercial worlds, and to accelerating the deployment of software infrastructures capable of highly reliable, secure, and globally scaleable message delivery.

31.2 Structural Foundations of the Chasm

There are two principal structural components to the chasm between academic research and commercial practice in the distributed systems arena – a communication gap and a deployment gap. These are tied to the two processes for deployment of new technologies in the commercial world, which we will term *market driven* and *technology driven*. A market driven deployment is the result of vendors listening to specific customer requirements, building out proprietary solutions that are targeted at these requirements, and then deploying these in customer networks. The technology driven approach starts with research, open consensus building, and technology standardization. Vendors then implement the standards, and customers theoretically start deploying implementations of these standards. For a given technology, the risk of failure and time to market are dramatically longer for the technology driven approach than for the market driven one. However, the market driven approach creates proprietary network islands that cannot communicate with each other. For a given technology, the size of the potential market opportunity is much larger with standards, but vendors have to give up their proprietary barriers to entry in order to pursue this approach.

31.2.1 The Communication Gap

Figure 31.1 illustrates these technology and market driven processes, as well as the two gaps that help make up the chasm. The architects and developers of proprietary commercial solutions are often educated by our graduate schools, and in theory should be receptive to the latest research from the academic community. However, there is a fundamental communication gap, due to the unidirectional nature of this communication; information flows out from the research community but is not returned by the commercial world. Distributed systems architectures have unusually complicated design spaces, and the process of developing a set of technologies that meets industry needs has to be an iterative one with the customers. As an example, my previous company is on its fifth completely revised architecture for pub-

lish/subscribe messaging. These were developed iteratively over ten years, due to the experience of dealing with the engineering tradeoffs of many of the most demanding consumers of distributed systems infrastructure, such as the financial stock exchanges. In my experience, while having a solid theoretical foundation is important, we found it very difficult to derive value from the work of the research community, because of the absence of communication on requirements and design tradeoffs that have gone on between the vendors and the academic community. Part of this is due to biases on the part of the research community, such as a common lack of interest in learning about these requirements. However, the majority of it is due to the inability of most commercial organizations to communicate these requirements and tradeoffs, for fear of weakening their market position.

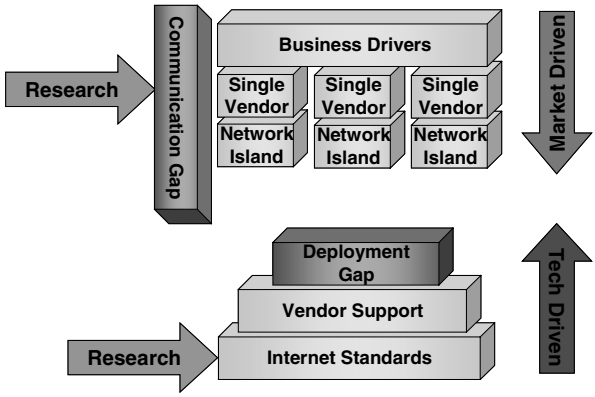

Fig. 31.1. Structural Foundations of the Chasm

While this is an issue for many fields, it is particularly acute in the distributed systems arena, where it is usually more important for commercial success to ask the right questions, than to find the best answers for those questions. On the other hand, the research community is extremely good at finding answers to theoretically interesting questions, and finding interesting answers is often rewarded more highly than spending time finding or answering questions that are interesting to industry.

31.2.2 The Deployment Gap

These communication problems are relatively well understood. What is less understood are the natural deployment challenges facing the actual technology industry itself. There is a near ubiquitous belief that open standards are the way to wide industry deployment of new distributed systems and advanced networking technology. This approach avoids the communication gap, because academics and vendors cooperate together in an open process. As shown in Figure 31.1, the technology driven model of adoption is to create standards to fulfill a perceived market requirement, get vendors to implement them, and then wait for customer to deploy.

Unfortunately, for standards that add qualitatively new functionality to the network, like most distributed systems do, this does not work. The commercial value of a network technology is exponential with the number of connected users and applications. For new horizontal technologies that are developed and standardized, if they are driven by a specific "killer app" that requires this technology (such as HTTP or SMTP), the adoption rate is exponential. However, if the technologies are being standardized to improve a range of applications by adding new functionality to the network (such as IP multicast, reliable multicast, IP security, QoS, etc.) then they usually enter a terminal "chicken or egg" phase, where they never reach critical mass. To use a metaphor, the information superhighway has been laid in concrete. We are able to add more lanes to it, but not change the nature of the roads.

31.3 Important Future Research Directions

It is essential that much of academic research be relatively unfettered by current commercial requirements, particularly theoretical and forward-looking work. However, to the extent that the academic world wishes to see its research influence commercial practice, it is important to both understand the nature of the impeding chasm, and to look at ways of bridging it. Luckily, there are some very fertile areas for academic research that are both interesting for purely academic reasons, and also directly address the chasm. It is the opinion of the author that overlay networking and contract based object routing are two of the most important and promising.

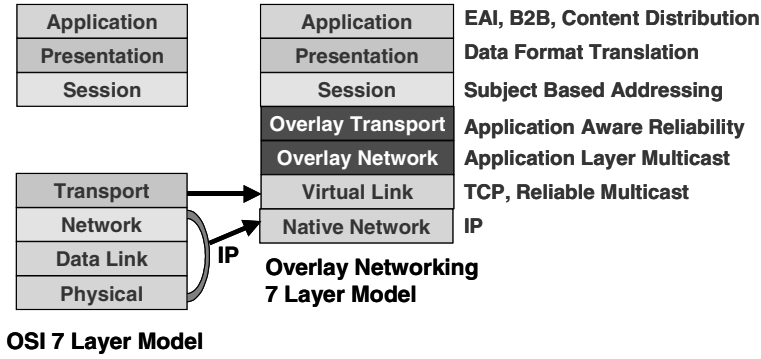

Fig. 31.2. Overlay Networking and the OSI 7 Layer Model

31.3.1 Overlay Networking

Overlay networking in effect creates a whole new network on top of the existing one. An overlay network is typically proprietary, driven by specific customer requirements, and does not connect to other overlay networks. If we can't rebuild the highways, we can build a FedEx overlay system, which optimizes the existing infrastructure. Rather than trying to upgrade the routers and hosts, you add new ones. Examples

of successful commercial overlay networks include content distribution networks such as Akamai, virtual private networks, TIBCO and Talarian's publish/subscribe customers, and content networking customers. Overlay networking research includes [31.1], [31.2], [31.3].

The most widely known overlay networks, such as the multicast Mbone [31.2], have attempted to run the same network protocols in an overlay fashion. These overlays are fundamentally technology driven deployments, where incremental deployment is the only primary value proposition of the overlay. While incremental deployment is a crucial value proposition of overlay networks, by itself it is rarely sufficient. Most commercial overlay networks follow a market driven approach, with their functionality tuned to the business requirements of the applications running on top of them. When coupled with the infrastructure class availability and robustness of network level hardware, these three business drivers – incremental deployment, application aware functionality, and infrastructure class robustness – define the business requirements of general overlay networks. From a technical perspective, the general model for overlay networking requires redefinition of the OSI 7 layer model. Figure 31.2 demonstrates this. From the perspective of the overlay network infrastructure, the same seven layers exist, but in modified form. At the lowest level, layers 1-3 look like the physical layer, and the transport layer acts as the link layer. As an example, an overlay router in the Internet sees a TCP connection as the equivalent of a frame relay interface, and a reliable multicast connection as the equivalent of an Ethernet interface. The overlay network then implements its own network and transport layers, providing routing and reliability protocols that are typically (but not necessarily, as in the case of technology driven overlays) configured or designed in application specific ways.

31.3.2 Contract Based Object Routing

Overlay networking helps solve the deployment problems for specific customers, but leaves us with islands of connectivity and proprietary development interfaces. One solution that has been proposed is to standardize the interfaces, such as the Java Messaging Service (JMS). This allows programmers to reuse their skills across products, and allows customers to avoid vendor lock-in. However, this does not allow islands of connectivity to communicate in a way that security and reliability guarantees are preserved, and so does not bridge the deployment gap between technology driven and market driven processes. In order to solve this, a solution is needed that is both deployable across an extremely wide range of languages and systems, is simple enough to be widely implemented by many vendors, and which provides end to end security and reliability guarantees.

A solution to this is called contract based object routing, which is presently being researched by the author. Contract based object routing has the following key characteristics. First, it aims to standardize the business level and application level semantics of message delivery, rather than the technical programming interface. Second, it uses a self-describing language (XML) to describe the dictionary of potential service level agreements (SLAs), as well as the semantics of a particular SLA. Third, it specifies a

procedure for negotiation of a SLA between any two applications or service providers. Fourth, it specifies a procedure for hierarchically extending a SLA negotiation from a single service provider out to multiple service providers, allowing for delivery semantics to be guaranteed across multiple islands of proprietary connectivity. Fifth, message delivery semantics extend to both real time (i.e. publish/subscribe), non real time (i.e. caching), and partial update (i.e. the latest stock quote in a web page) scenarios. Sixth, it is cross language, as opposed to a Java only solution like JMS. Seventh, a SLA can be negotiated either in-band (i.e. in a given message) or out-of-band (i.e. through a long term agreement with a service provider). Eight, it specifies semantics for monitoring and notification of the fulfillment status of delivery contracts. Ninth, it can be integrated with a range of interfaces, including proprietary messaging systems, SOAP, HTTP, TCP, or file-based systems. Tenth, it covers both unicast and multicast semantics.

While it is beyond the scope and purpose of this document to describe contract based object routing in detail, a few of the salient details follow. Applications first interface to the infrastructure using a *control channel interface*, which allows the application to signal and negotiate its needed delivery semantics, using XML based *object routing contracts*. The semantics of these contracts are specified in a logically centralized *contract dictionary*, and are typically negotiated both with the sender's *primary delivery server* as well as each of the receivers' primary delivery servers. There are different *contract negotiation protocols* for unicast and multicast delivery. The application sends data, for delivery according to an associated object routing contract, over a *data channel interface*, which both provides acknowledgement of receipt by one or more delivery servers, and may notify the application when delivery is complete. In addition to the contract dictionary and the primary delivery server, the infrastructure may also support routing servers, which provide overlay network routing without end to end delivery semantics, and gateway servers, which are responsible for bridging heterogeneous application or administration domains together.

Example implementations for the control channel interface are an in-band contract request inside a data message, or a standardized unicast (TCP, SOAP, HTTP, SMTP, etc.) connection with the primary delivery server. Example implementations for the data channel interface include a standardized unicast connection or a file interface, where messages to be sent are placed in an outgoing directory, and are automatically delivered to a different incoming directory. Example implementations for the contract negotiation protocols include a simple request protocol – where the sender specifies the desired object routing contract, and the infrastructure either delivers it with these semantics or reports a failure – and a three phase commit algorithm, that includes both the sender, the receiver, and each of their primary delivery servers.

31.4 Conclusion

There exists a significant chasm that separates the research of the distributed systems community from widespread deployment. Given the current transition from third wave web based computing to the fourth wave of message based computing, it is of

paramount importance to understand and attempt to bridge this chasm. This paper has argued that the reasons for this chasm are largely structural, and that with understanding of the structure of this chasm, solutions present themselves. It is the opinion of the author that the distributed systems research community should make "bridge building" a high priority, and that specific academic research areas look particularly promising, most notably overlay networking and contract based object routing. This paper has outlined some of the defining characteristics of these research areas, but tremendous work – and opportunity – remains in these areas.

References

31.1 Andersen, D., Balakrishnan, H., Kaashoek, M., Morris, R. "Resilient Overlay Networks", SOSP 2001.

31.2 Eriksson, H. "Mbone: The Multicast Backbone", Communications of the ACM 37, 8 (1994) 54-60.

31.3 Touch, J. and Holtz, S. The X-Bone. In Proc. 3rd Global Internet Mini-Conference (Sydney, Australia, Nov 1998), pp. 75-83.

32. Holistic Operations in Large-Scale Sensor Network Systems: A Probabilistic Peer-to-Peer Approach

Indranil Gupta and Kenneth P. Birman*

Dept. of Computer Science
Cornell University
Ithaca NY 14853
{gupta,ken}@cs.cornell.edu

32.1 Sensor Nodes and Large-Scale Networks

Smart sensor nodes integrate multiple sensors (e.g., temperature, humidity, accelerometers), processing capability (microprocessor connected to the sensors using I2C technology), wireless communications, and a battery power source. A sensor node could be as small as a few millimeters. See references [32.11, 32.12, 32.14, 32.15].

Sensor nodes have low processing (few MHzs), memory (few KB of RAM, few MB of flash memory), and wireless communication capabilities (few hundreds of Kbps). These ranges are often determined by application domain constraints such as cost, power consumption, and deployment space, and thus are likely to stay despite improvements in fabrication techniques. For example, retinal sensor sizes depend on ganglial nerve separation [32.11]. Space limits on the wireless antenna restricts maximum frequency and range of transmission. Availability of power is limited due to the difficulty or impossibility of recharging nodes in inhospitable terrains, e.g., military applications [32.12].

On the other hand, it is increasingly easy to produce sensor nodes in large numbers [32.14, 32.15]. An autonomous system consisting of a large number of sensor nodes deployed over an area and integrated to collaborate through a (wireless or wired) network, would encourage several novel as well as existing applications. Examples include high fidelity image processing in retinal prosthesis chips with hundreds to millions of nodes [32.11], battlefield applications such as vehicle tracking, environmental observation and forecasting systems (EOFS) [32.13], etc.

32.2 Holistic Operations

In such sensor network systems, operations spanning a large number of sensor nodes (perhaps only a subset of all the nodes) assume greater importance than the exact reading at a specific node. See references [32.2, 32.4, 32.10, 32.11, 32.13]. A user monitoring the area is usually more interested in collecting data from sensor nodes, disseminating commands to them, and in general being able to control the system *holistically* . An

* The authors were supported in part by DARPA/AFRL-IFGA grant F30602-99-1-0532 and in part by NSF-CISE grant 9703470, with additional support from the AFRL-IFGA Information Assurance Institute, from Microsoft Research and from the Intel Corporation.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 180–185, 2003.

autonomous system-wide application (e.g., one that takes corrective action when the average humidity in the area crosses a threshold) would require streaming updates of global aggregates of sensor node readings. Moreover, individual sensor node readings are inaccurate, making a system-wide estimate more valuable.

We term these as *holistic operations*, because they involve large parts of the sensor network rather than individual or small groups of sensor nodes. We are interested in finding *scalable* and *reliable* solutions to holistic operations such as reliable multicast, data aggregation, leader election, group membership, distributed indexing of files and data, etc. We believe that implementing holistic operations scalably and reliably, along with the ability to trade off between the two properties, is an important step towards the development of truly autonomous large-area sensor network systems. We limit our discussion in this paper to reliable multicast and data aggregation. Data aggregation can be used to calculate global aggregates across sensor nodes (e.g., average, variance, or maximum, of a measured quantity such as temperature), for collaborative signal processing (e.g., as in [32.1, 32.11]), etc. A multicast protocol can be used to disseminate commands or data into the sensor network, e.g., in an EOFS system [32.13].

Solving these problems over an asynchronous network is known to be difficult, and often a provably impossible task [32.5]. Nevertheless, protocols for sensor network systems are required to be tolerant to permanent and transient failures arising in individual sensor nodes (e.g., from discharge of power, destruction from external causes) and in packet transmission (e.g., due to interference in the wireless medium, inhospitable terrain). They have to scale in terms of overhead (computation and communication) and power consumption at nodes, network load (which affects per-node overhead under multi-hop routing), and variation of these loads with system size.

Properties of these protocols should be applicable even when the assumed sensor network model differs from the real-life one. Protocol behavior must degrade gracefully due to interference from other concurrently present strategies or protocols in the sensor network. Examples of such influences include anonymity of sensor nodes (i.e., lack of unique identifiers), presence of power saving strategies such as periodic sleeps, etc. This makes protocol design a challenge, since these constraints are determined by models and standards for the sensor node, network, basic protocols and applications, all of which are constantly evolving.

32.3 Why a Peer-to-Peer Approach?

Environmental monitoring and biomedical systems [32.11, 32.13] appear to use either a centralized architecture, where sensor node data (e.g., measurements) are centrally processed. Sensor nodes communicate data either (a) directly to a single or small number of base stations, or (b) through a cluster-based overlay network formed among the nodes (often structured hierarchically). Data compression and application-dependent optimization (e.g., segmentation in image processing [32.11]) are used to reduce the power used by communication. Cyclic redundancy checks are used to counter packet loss or corruption. Cluster-based systems reduce the effect of node failures through fault-tolerant backup within and across clusters.

Using a centralized architecture to aggregate or multicast data to the sensor network has the disadvantages of (a) high communication overhead to a distant base station (due to the absence of the ability to set up a denser base station infrastructure, a characteristic property of isolated application domains), and (b) a turnaround time (latency) in the holistic operation that grows linearly with the number of nodes. For example, the bandwidth offered by the ATM-845/851 acoustic modems used in the centralized CORIE EOFS system was found to suffice for transmitting commands, but inadequate for the amount of data generated by nodes [32.13]. Using a cluster-based technique alleviates some of these problems but introduces a fault-tolerance problem [32.7], e.g., transient or permanent failure of cluster leaders or the communication medium around them would either affect completeness of continuous global aggregate calculation or cause a behavior akin to thrashing if aggressive failure detection initiates frequent leader election runs. Migrating leadership across nodes in a cluster also entails additional overhead. Often, it is also difficult to mathematically guarantee or prove any properties about the turnaround time between initiation and completion of a one-shot holistic operation in the system. A quick turnaround time is necessary to achieve soft real-time properties that are essential to a self-managing system.

32.4 The Probabilistic Approach

A promising approach to implementing holistic operations in large sensor networks is based on the use of probabilistic protocols [32.3, 32.6, 32.7, 32.8, 32.9, 32.10]. These protocols are peer-to-peer in nature, and do not require infrastructure such as base stations or central points of control. They impose low computation and communication overhead on sensor nodes, and this load is equally distributed across all sensor nodes, avoiding hot-spots. The protocols are simple, and thus have a small code footprint. They provide probabilistic guarantees on correctness or reliability of the operation. A tradeoff between scalability and reliability is achieved through tuning of the overhead at individual sensor nodes. High probability reliabilities can be obtained even in the presence of node failures and high packet loss rates, and within turnaround times that increase slowly with system size. Finally, deterministic correctness is provided by a concurrent protocol that inexpensively backs up the probabilistic protocol.

We briefly discuss probabilistic protocols for multicast and aggregation.

Multicast: Epidemic- (or gossip-) based protocols for data dissemination in an ad-hoc network employ a probabilistic style of dissemination, where each node independently makes probabilistic decisions about how it forwards (gossips about) received data to its neighboring and remote nodes [32.6, 32.8, 32.9]. Gossip rates can be tuned to trade off between the reliability of dissemination and communication overhead. Epidemic-style of data dissemination is an efficient alternative to data flooding.

Epidemic protocols can be imparted properties such as topology awareness and adaptivity by using weak overlays and topological mappings [32.6, 32.7, 32.8]. Topological awareness causes packets to traverse fewer hops, reducing routing overhead. Adaptivity lowers overhead at small failure rates in the system.

The dissemination protocols in [32.6] require sensor nodes to construct a virtual weak overlay called the Leaf Box Hierarchy (or the Grid Box Hierarchy, which is an instance of the former)[1]. Hierarchical epidemic algorithms within the Leaf Box Hierarchy impose an overhead per node that grows as the square of the logarithm of system size, and disseminate multicast data to interested nodes with very high probability. The dissemination latency grows very slowly with system size (a sublinear variation). This protocol can then be augmented with an adaptive scheme that ensures constant average per-node costs (independent of system size) in the presence of a few failures. The scheme adapts overhead to maintain reliability with increasing failure rate.

Data Aggregation: In [32.7], we have reasoned about traditional approaches to the problem of calculating composable global functions (such as average, variance, maximum etc.) over a large set of readings from sensor nodes - considered system sizes ran into thousands of nodes. Schemes based on traditional leader election of nodes into clusters and cluster heads have inherent drawbacks with respect to tolerance of the final estimate to node failures or packet losses. For example, during a global aggregate calculation, failure of a node holding an aggregate from a large subset of sensor nodes would cause exclusion of those readings from the final estimate. Our work in [32.7] proposes a new protocol that uses gossiping within the Leaf/Grid Box Hierarchy to disseminate an estimate of the global aggregate to *all* sensor nodes. Evaluation of the protocol shows that with a per-node message overhead varying with the square of the logarithm of system size, the protocol produces global aggregate estimates with very high completeness (i.e., fraction of sensor node readings included in the estimate).

32.5 Continuing Work and Future Directions

Probabilistic protocols appear to not only match the hardware specifications of sensor nodes but also solve a variety of holistic operation specifications essential to the proliferation of truly autonomous sensor network systems. Future directions for research in this class of solutions include the following.

- *Overlay Self-Assembly:* Protocols for constructing this overlay in a distributed fashion, even in the absence of location services such as GPS.
- *Overlay Self-Management and -Reconfiguration:* Overlay reconfiguration might be (a) application initiated, e.g., a change in the set of sensor nodes involved in the operation, or (b) required automatically, e.g., due to failures and mobility of nodes. This assumes importance for long-running holistic operations.
- *Energy Efficiency:* Studying the increase in power consumption introduced by redundant transmissions of the same information in probabilistic protocols, and comparison with centralized approaches.
- *Applicability under Altered Models:*
  - *Anonymous Nodes:* An overlay such as the Leaf Box Hierarchy could be used to provide a coarse addressing scheme in a network with anonymous sensor nodes.

[1] Other examples of such overlays include the Amorphous Computing hierarchy [32.2].

  - *Interference from other concurrently present strategies:* Our approach accommodates a range of concurrently present strategies in the network. For example, a concurrent power-saving strategy, where each node sleeps periodically for a while, causes graceful degradation because of the fault-tolerance properties of probabilistic protocols.
- *Performance Tuning:* Variation across applications of tradeoff between sensor node overhead (e.g., power consumption) and required reliability of the holistic operation.
- *Interaction with Ad-hoc Routing Protocols:* Adapting to variation in route length and quality arising from factors such as the multi-hop ad-hoc routing protocol used, arbitrary placements of nodes causing large disparity between routing and geographical distances.
- *Improving Existing Probabilistic Protocols:* For example, the probabilistic aggregation protocol could be used to calculate streaming aggregates, or used on an arbitrary-sized region, or to accommodate non-composable aggregate functions as might be required by collaborative signal processing applications.

Broader research goals include exploration of:

- Promising areas of applicability of probabilistic solutions to holistic operations, e.g., environmental observation systems.
- Combination of collaborative in-network signal processing with protocols for holistic operations.
- Holistic operations that are atypical in Internet-based process group computing, yet assume significance in sensor networks.
- New sensor network applications that might be enabled by efficient and scalable solutions to holistic operations.

References

32.1 S.R. Blatt, "Collaborative signal processing", BAE Systems' presentation at *SensIT PI Meet*, DARPA IXO, Apr 2001.
32.2 D. Coore, R. Nagpal, R. Weiss, "Paradigms for structure in an Amorphous Computer", A.I. Memo No. 1614, A.I. Laboratory, MIT, Oct 1997.
32.3 A. Demers, et al, "Epidemic algorithms for replicated database maintenance", *Proc. 6^{th} ACM PODC*, pages 1-12, Aug 1987.
32.4 D. Estrin, R. Govindan, J. Heidemann, S. Kumar, "Next century challenges: scalable coordination in sensor networks", *Proc. 5^{th} ACM/IEEE MobiCom*, pages 263-270, Aug 1999.
32.5 M.J. Fischer, N.A. Lynch, M.S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journ. ACM*, 32:2, pages 374-382, Apr 1985.
32.6 I. Gupta, A.-M. Kermarrec, A.J. Ganesh, "Efficient epidemic-style protocols for reliable and scalable multicast", *Proc. 21^{st} SRDS*, pages 180-189, Oct 2002.
32.7 I. Gupta, R. van Renesse, K. P. Birman, "Scalable fault-tolerant aggregation in large process groups", *Proc. 2001 DSN*, pages 303-312, Jul 2001.
32.8 D. Kempe, J. Kleinberg, A. Demers. "Spatial gossip and resource location protocols", *Proc. 33^{rd} ACM STOC*, pages 163-172, Jul 2001.
32.9 L. Li, Z. Haas, J.Y. Halpern, "Gossip-based ad hoc routing", *Proc. 21^{st} IEEE INFOCOM*, pages 1707-1716, Jun 2002.
32.10 R. van Renesse, K. Birman, "Scalable management and data mining using Astrolabe", *Proc. 1^{st} IPTPS, LNCS 2429*, pages 280-294, 2002.

32.11 L. Schwiebert, S.K.S. Gupta, J. Weinmann, "Research challenges in wireless networks of biomedical sensors", *Proc. 7th ACM/IEEE MobiCom*, pages 151-165, Jul 2001.
32.12 Smart Dust Project, http://robotics.eecs.berkeley.edu/~pister/SmartDust
32.13 D.C. Steere, A. Baptista, D. McNamee, C. Pu, J. Walpole, "Research challenges in environmental observation and forecasting systems", *Proc. 6th ACM/IEEE MobiCom*, pages 292-299, Aug 2000.
32.14 http://www.cs.berkeley.edy/~culler/cs294-8, http://www.cs.rutgers.edu/~mini
32.15 Crossbow Technology Inc., http://www.xbow.com

33. Challenges in Making Pervasive Systems Dependable

Christof Fetzer and Karin Högstedt

AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932, USA
{christof,karin}@research.att.com

33.1 Introduction

Due to "Moore's Law" [33.20] we have been witnessing an exponential increase in processing power, disk capacity and network bandwidth for more than four decades. Fueled by the underlying exponential increase in circuit density, which is expected to continue for at least another decade, it becomes economically feasible to build pervasive systems.

A pervasive system consists of a large set of networked devices, seemingly invisibly embedded in the environment. Pervasive systems research started in the late 1980s at Xerox PARC [33.26]. A variety of application domains have been proposed for pervasive systems, e.g., education [33.1, 33.5], public spaces [33.11], health care [33.24], and home control systems [33.15, 33.16, 33.21].

Most of the past and current work in pervasive systems have been focusing on the human/computer interface. Our work focuses instead on the dependability issues, i.e., reliability, availability, security, and manageability issues of pervasive systems. While it is recognized that the dependability of pervasive systems is an important issue (e.g., [33.8]), not much research has been done in this area. A notable exception is [33.25] that describes a dependable infrastructure for home automation. However, that work focuses mainly on a soft-state approach for home automation and many questions remain open.

Pervasive systems have to become more dependable before they become deployable on a larger scale. In particular, pervasive systems will not be widely deployed if they require users to invest a substantial amount of time or money to keep them operating. Our goal is to find dependability mechanisms and policies that (1) maximize the dependability of pervasive systems, while (2) minimizing the cost of operation (including deployment and maintenance costs). Informally, we use the term *pervasive dependability* to refer to these two requirements.

If we succeed to make pervasive system sufficiently dependable such that they become deployed widely, they will become an integral part of the infrastructure upon which our society depends. While only a few pervasive systems might be safety or mission critical (e.g., health care related systems), most of them will likely to become *society critical*: while the unavailability of a few such systems might be a mere inconvenience, the concurrent outage of a large number of systems might have broad economical consequences. For example, it might be acceptable if a few pervasive ho me networks go down, but if a large number fail concurrently, our society might be adversely impacted (added traffic by telecommuters, lost productivity, etc.). Hence, pervasive dependability needs to address the problems raised by society critical systems.

Achieving pervasive dependability is difficult due to the large number of hardware and software components in a pervasive systems and the cost constraints (e.g., this lim-

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 186–190, 2003.

its the amount of manual labor for system administration). Our approach in achieving pervasive dependability is to harness the power of both "Moore's Law" and the Scientific Method [33.7]. The idea is to keep as much run-time data as possible (using the exponentially increasing disk sizes), perform automated data mining (using the exponentially increasing network speeds to collect distributed data) and automated fault-injection experiments (using the exponentially increasing processor speeds) to automatically increase the dependability of pervasive systems without the need of human intervention.

Automation is achieved by mechanizing the Scientific Method. Based on the properties of the software components, the system automatically selects appropriate dependability mechanisms. In particular, to derive properties of software components, our system tests a set of hypotheses (e.g., function f crashes if its first argument is negative) using fault-injection to reject invalid hypotheses. Based on the non-rejected hypotheses, the system then selects a set of appropriate dependability mechanisms (e.g., creates a wrapper for f that checks that the first argument of f is not negative).

In the Scientific Method, incorrect hypotheses are rejected by thorough experimentation and the hypothesis that explains the experimentation results is accepted. A previously accepted hypothesis might be rejected and replaced by a more precise hypothesis at some later point if more experimental data becomes available. In our automated system, a previously accepted hypothesis might be rejected due to logged field data. If the optimality (or correctness) of a dependability mechanism depends on the validity of a hypothesis, one cannot guarantee optimality (or correctness) since new data can lead to rejection of the hypothesis. Except for maybe safety critical systems, we believe the potential benefits (e.g., higher availability and lower operational cost) will outweigh the potential risks (e.g., incorrect failure masking).

We are investigating this idea in the context of pervasive home systems. Initial results demonstrate that one can increase the dependability of C libraries automatically using automated fault-injection experiments [33.10]. We are currently building a new middleware platform for pervasive systems that is designed to support automated fault-injection experiments [33.9].

33.2 State of the Art

The most fundamental challenge in making pervasive systems dependable is complexity. The sheer number of devices and software components, and the requirement to facilitate cooperation between all devices result in very complex systems. Furthermore, there will be a large number of pervasive systems all of which are different from each other. For example, consider the context of pervasive home systems. If sensors and actuators become sufficiently inexpensive, future homes might deploy pervasive systems consisting of hundreds or even thousands of devices. One has to expect that each home will use a different set of devices and software components. Managing such devices will be a challenge: diagnosing why something does not work in a complex system is non-trivial but will frequently be needed because the failure frequency increases with the number of devices and software components.

The complexity of pervasive systems has many aspects. Marc Weiser articulated some of these aspects as follows [33.26]: "*If the computational system is invisible as*

well as extensive, it becomes hard to know what is controlling what, what is connected to what, where information is flowing, how it is being used, what is broken [...], and what are the consequences of any given action [...]". His statement is valid both at the human interface level and at the software level. In our approach we attempt to address the issue of complexity in a systematic way.

We have been designing and implementing a middleware infrastructure for pervasive systems [33.9]. The focus of this platform is to help us to deal with the complexity of pervasive systems. We achieve this by facilitating automated fault-injection and data collection of application software components and the middleware platform itself.

We decided to design and implement a data-flow oriented system. In this system, an application is designed by a set of components connected by explicit links. Communication of data (and control) is restricted to these links. In this way the system can keep track of data (and control) flow. Logging the data flowing via the links permits the system to track how components are being used and to determine what is broken. Using fault-injection, the system can also determine the consequences of faults.

Data-flow oriented systems have been investigated in different contexts in the recent years [33.12, 33.17, 33.4, 33.27, 33.13]. For example, data-flow oriented systems facilitate adaptation [33.6, 33.12, 33.22], throughput optimization [33.27], threading optimization [33.17], and media streaming [33.4]. Our investigation focuses on the applicability of data-flow oriented systems for pervasive dependability. The major questions that we are addressing in this context are how one performs (1) self-configuration, (2) self-tuning, and (3) self-diagnosis (see Section 33.3).

There has been a large body of work addressing the issue of reconfiguration of component-based systems [33.2, 33.3, 33.12, 33.14, 33.18, 33.19, 33.23]. However, with the possible exceptions of [33.12, 33.19], which provide (non-trivial) automatic reconfiguration of pipelined applications, all of these approaches require an amount of user (or system administrator) involvement that is inappropriate for pervasive dependable systems. When addressing this target domain, we need to develop a more general system that requires less user involvement, and that handles arbitrary component graphs and more failure types.

33.3 Future Research Directions

Currently, we see the following challenges in achieving pervasive dependability: self-configuration, self-diagnosis, and self-tuning.

Self-configuration: By *self-configuration* we mean how one can automatically modify an application to optimize its dependability at configuration time. The dependability aspects of an application (e.g., availability and timeliness) depend heavily on the available hardware and software resources and the failure frequency of the underlying computing system. In a pervasive system we expect a wide range of processing speeds, network speeds, disk capacity, and failure rates. Due to "Moore's Law", over time this range will widen even further when new hardware is added to a system while old hardware stays in the system. The self-configuration challenge is to automatically determine how

the dependability of an application can be improved, e.g., automatically selecting an appropriate replication mechanism and replication degree.

Self-diagnosis: We use the term *self-diagnosis* to refer to the process of automatic fault-diagnosis. For example, the cause of a too high end-to-end delay might be a link experiencing excessive failure rates. When an application is configured appropriately to increase its dependability, it can cope with a certain number of failures. When the number of failures becomes too high, the application will not be able to mask all failures and a reconfiguration (i.e., change of application structure) might be needed. To guide the reconfiguration, a fault-diagnosis will be needed.

Experiences collected in the context of adaptation [33.12] indicate that diagnosis can in some situations be too time consuming to perform since a reconfiguration must be performed quickly. We view it as a challenge to configure an application such that there is enough redundancy to be able to perform a fault-diagnosis before a reconfiguration.

Self-tuning: By term *self-tuning* we refer to the optimization of the dependability of an application during run-time. The application dependability can be tuned by changing the structure of the application (e.g., changing the links between software components), or by changing the behavior of components (e.g., changing the sending rate of a component). The challenge is to find an appropriate combination of behavioral and structural changes that optimizes the dependability of an application.

References

33.1 G. D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4):508–530, 1999.

33.2 M. R. Barbacci, C. B. Weinstock, D. L. Doubleday, M. J. Gardner, and R. W. Lichota. Durra: a structure description language for developing distributed applications. *IEE Software Engineering Journal*, 8(2):83–94, 1993.

33.3 T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

33.4 G. Bond, E. Cheung, A. Forrest, M. Jackson, H. Purdy, C. Ramming, and P. Zave. DFC as the basis for ECLIPSE, an IP communications software platform. In *Proceedings of IP Telecom Services Workshop*, 19–26, 2000 Sept.

33.5 A. Chen, R.R. Muntz, S. Yuen, I. Locher, S.I. Park, and M.B. Srivastava. A support infrastructure for the smart kindergarten. *IEEE Pervasive Computing*, 1(2):49–57, 2002.

33.6 S.-W. Cheng et al. Using architectural style as a basis for self-repair. In *IEEE/IFIP Conference on Software Architecture*, 2002.

33.7 Morris R. Cohen and Ernest Nagel. *An Introduction to Logic and Scientific Method*. Simon Publications, 2002.

33.8 N. Davies and H.W. Gellersens. Beyond prototypes: Challenges in deploying ubiquitous systems. *IEEE Pervasive Computing*, 1(2):26–35, 2002.

33.9 C. Fetzer and K. Högstedt. Self*: A component based data-flow oriented framework for pervasive dependability. In *Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

33.10 C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of C libraries. In *International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

33.11 M. Fleck, M. Frid, T. Kindberg, E. O'Brien-Strain, R. Rajani, and M. Spasojevic. From informing to remembering: ubiquitous systems in interactive museums. *IEEE Pervasive Computing*, 1(2):13 – 21, 2002.

33.12 X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

33.13 M.M. Gorlick and R.R Razouk. Using weaves for software construction and analysis. In *13th International Conference on Software Engineering*, 23–34, 1991.

33.14 C. Hofmeister, E. White, and J. Purtillo. Surgeon: a packager for dynamically reconfigurable distributed applications. *IEE System Engineering Journal*, 8(2):95–101, March 1993.

33.15 S.S. Intille. Designing a home of the future. *IEEE Pervasive Computing*, 1(2):76 – 82, 2002.

33.16 C. D. Kidd et al. The aware home: A living laboratory for ubiquitous computing research. In *2nd International Workshop on Cooperative Buildings*, 1999.

33.17 R. Koster, A.P. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *International Workshop on Multimedia Middleware*, 2001.

33.18 J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

33.19 V. Martin and K. Schwan. ILI: An adaptive infrastructure for dynamic interactive distributed applications. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, 224– 231, 1998.

33.20 Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114 – 117, April 1965.

33.21 M. Mozer. The neural network house: An environment that adapts to its inhabitants. In *AAAI Spring Symposium on Intelligent Environments*, 110–114, 1998.

33.22 P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

33.23 James M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):151–174, 1994.

33.24 V. Stanford. Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1):10 – 13, 2002.

33.25 Y. M. Wang, W. Russell, and A. Arora. A toolkit for building dependable and extensible home networking applications. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

33.26 M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal*, 38(4):693–693, 1999.

33.27 M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

34. Towards Dependable Networks of Mobile Arbitrary Devices – Diagnosis and Scalability

Miroslaw Malek

Institut für Informatik
Humboldt-Universität Berlin
malek@informatik.hu-berlin.de

34.1 Introduction

After a period of explosive, unmitigated growth, the web is ready for the second chapter of the Internet revolution, namely, the support of pervasive (ubiquitous) computing where dynamic change, flexibility and on-demand components and services configurability will be expected.. While consolidation, emerging maturity and survival of the fittest will rule at the corporate level, the web with virtually billions of devices attached to it and trillions of bytes of data will have to transform into an information, knowledge and remote control utility available to unprecedented numbers of novice as well as mature users anywhere, anytime. To lay out a foundation for this challenge the dream of adaptive, maintenance-free, self-relying systems must become a reality as public dependence on those systems will continue to rise and there will be insufficient human resources to continually support and maintain the computing/communication infrastructure. It is argued that this goal can only be achieved by addressing first the problems of scalability and fault detection and isolation (fault diagnosis) followed by a rapid recovery, mainly by the use of, for example, hierarchical partitioning and consensus.

With over one billion users of cellular phones alone who are already migrating to the web, billions of devices (sensors, actuators and terminals) appended to the net, billions lines of code, billions of web pages, many of them unmanaged or unmanageable, the web might become a cobweb. The complexities are enormous and the methods of dealing with them must be pursued. The vision suggested in this paper is that of **NOMADs** (**N**etworks **O**f **M**obile, **A**rbitrary **D**evices) where each network (cluster, overlay or a peer-to-peer group) focuses on guaranteeing a specific level of quality-of-service. The networks are connected over the web and sensors, actuators, terminal devices such as PC's, PDA's and phones and other arbitrary devices such as webcams, robots, mechanical games are attached to it. The users and devices may be mobile and can control, observe and/or be controlled via the network, be it physical or wireless (see Figure 34.1).

Although most of us happily use such web daily (as more and more aforementioned functionalities continue to emerge), rely on it and are pleased with the overall performance, the web's design has been a social rather than technical phenomenon beyond control from its beginning so not surprisingly the structure of such system is rather chaotic, the security compromised and readiness to perform demanding real-time applications, practically non-existent. The web is similar to a social system where malfunction or misbehaviors are tolerated or dealt with better or worse efficiency. Since it is a social movement, stopping the growth, controlling, regulating, or radically redesigning the web

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 191–196, 2003.

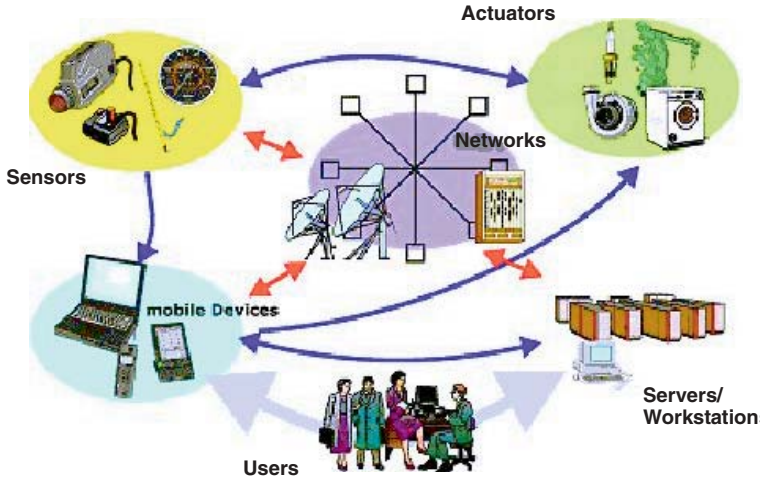

Fig. 34.1. NOMADs – Networks Of Mobile Arbitrary Devices.

is extremely difficult unless a revolution or a seismic event, be it a technological breakthrough or a massive cyber attack occur. Over a billion users spend cumulatively billions of hours on the web daily and millions work on expanding and maintaining it. To facilitate the work of billions the challenge is not performance but mainly functionality and dependability with focus on maintenance-free systems. Another challenge is a gradual transformation of the web into NOMADs, an environment of static and nomadic entities, be it persons or devices, always online, always accessible with a vast range of device interfaces, protocols and diverse functionality. The NOMADs will have to cope with diversity and work reliably even in the most obscure circumstances and unconstrained size of the system. In this paper, the problems of unprecedented and unstoppable growth of the Internet or NOMADs and the demand for dependability are addressed, The models that might meet these challenges are introduced in the next sections with focus on scalability and adaptive diagnosis.

34.2 Omniscience, Consensus and Autonomy

There are, in principle, three potential directions (see Figure 34.2) to take when it comes to the design, development and maintenance of systems like internet [34.1].

These directions might also be useful to devising strategies to ensure properties such as dependability and security.

Omniscience, i.e., knowing the exact state after execution of each instruction and being able to optimize every decision might lead to prohibitively high complexity and overhead. It can mainly be applied to a small, well-defined systems or components and the web is certainly not one of them. It might be an acceptable approach to specification and design of low-complexity components.

Our second direction, *consensus* [34.2], is based on democratic rules, where various types of group communication, consensus, leader election and group memberships are

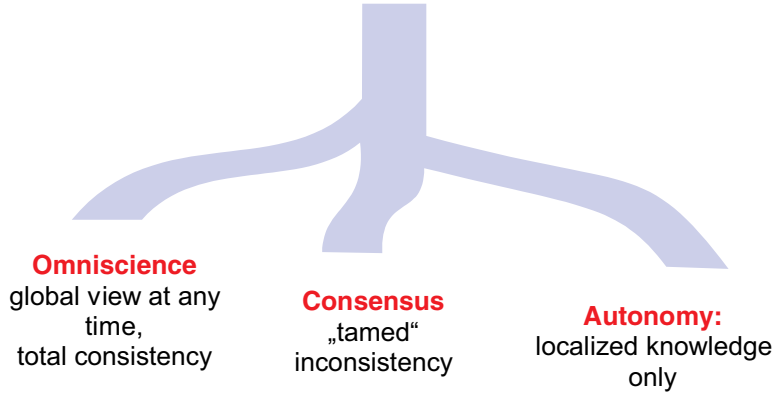

Fig. 34.2. Three tempting roads to design, development and maintenance of web-based systems.

formed. Depending on circumstances (e.g., fault types) the consensus may require just an agreement with any other node (a member of a population) if every fault in the system is assumed to be different, or a simple majority or an agreement of all nodes when it comes to, e.g., a database commitment. This approach has proven to be effective though somewhat inefficient. E.g., we have used it effectively in our unstoppable orchestra and robot balancing applications [34.3].

Obviously, when it comes to NOMADs, only a small subgroup (an overlay) consensus would be feasible and/or periodic hierarchical consensus which substantially reduces the number of required messages for all fault models should be considered..

The third direction is *autonomy* where the mechanism is similar to that of a stock exchange or a game in which every participant is there for himself/herself and tries to maximize his/her gain. In web environment this corresponds to establishing secure, dependable one-to-one connections which maximize the gain from a perspective of participating parties (e-commerce being a typical application). If there is substantial redundancy in a system, the autonomy might be a promising approach but from our experience a combination of consensus and autonomy results in a highly-dependable, self-relying system if a “carved out" cluster or an overlay in the web is of a manageable size.

34.3 Scalability

To change the cobweb back to the web we need to start with two fundamental problems, namely, scalability (to become independent of the number of users and units) and adaptive diagnosis, i.e., a diagnosis method capable of adapting itself to changing fault models and environments. The scalability challenge cannot be avoided nor ignored. Every day thousands of new nodes are being added to the web with irregular, often bursty, traffic patterns. The further challenges will include the ability to cope with rapid changes and a system’s increasing dynamics. A critical part of incorporating such capability is adaptive diagnosis. Once adaptive diagnosis methods are found the recovery, self-healing and self-

reliance will have to follow on the way to autonomic [34.4, 34.5] and trustworthy [34.6] computing.

The most viable approach to scalability that has accompanied us for years seems to be partitioning (clustering, grouping, coalition building) and a formation of an appropriate hierarchy. We, as a community, have been successful in building such systems starting with naming domains and database systems. Yet, the web is far cry from an orderly database. Chaos and game theories might be applied to bring some order to the wealth of data from which extracting useful and credible information and creating knowledge and eliminating noise become increasingly difficult. Today's search engines despite a formidable progress still lack robustness and sophistication of that associated with a solid database.

In all categories, be it users, devices, lines of code, web pages, structured design and transparent architecture combined with clear but rather flat hierarchy where the members of groups or clusters are capable of self-organization and self-reliance seems to be the most promising approach to scalability. We have investigated these issues in heterogeneous environments [34.7, 34.8] and have identified that interoperability, separation of concerns and translucency (ability to adjust attributes at optimal level) play an important role.

34.4 Adaptive Diagnosis

The fundamental step in the design and development of configurability/diagnosis/recovery/reconfigurability infrastructure for the web composed of hundreds of millions of elements is the status identification, namely systems diagnosis. In order to cope with scalability problems we have developed hierarchical diagnosis methods that deal with faults ranging from crashes to Byzantine [34.9]. These methods are being further extended to include scenarios of dynamically appearing clusters of fault-free and faulty elements and thus adapting to varying environments.

What are the alternatives to fault diagnosis in a complex system such as NOMADs? One should consider, for some specific environments and applications, alternative approaches such as "roll-forward" where the error logs, generated by a given node, are acknowledged, the workload transferred to fault free nodes and the issue of detailed diagnosis postponed once a faulty node has been determined. This approach can be taken in the case of simple fault models where an error detection is sufficient to signal a load reassignment and potential damages from a node being down are negligible. By the use of an autonomy paradigm we may strive to eliminate or minimize certain node dependencies and resort to the most popular recovery procedure, namely, a retry if an application can tolerate it.

For a more complex fault scenarios adaptive, scalable diagnosis should be used applying, e.g., a *multi-view hierarchical partitioning method*. The static hierarchical diagnosis method [34.9] has resulted in significant savings with respect to the number of consensus messages, while being applicable to a full range of fault models but a smaller number of fault scenarios. The new multi-view method should optimize the expense (the number of messages) while assuring the highest possible coverage including a relatively

large number of clustered faults. Gossiping might be an effective approach for this case [34.10].

The same or similar multi-view hierarchy would also serve as the basis for a robust task decomposition, configurability, recovery and dynamic reconfiguration.

34.5 Research Issues

With ever-increasing system complexity and the cost of maintenance, the main challenge will be to specify, design and develop maintenance-free, dependable, distributed systems such as NOMADs which can cope with rapid changes and mobility. To achieve these goals, the problems of diagnosis and/or "rolling-forward," scalability, online replacement and dynamic components-binding with specific properties must be solved first. Our group pursues these issues by forming overlays for specific applications such as remote control of experiments with robots or other devices and using components with specific properties.

34.6 Conclusion

The web is slowly but surely transforming in part into NOMADs but it might become a cobweb if sound engineering paradigms and principles are not followed. The hybrid model of consensus in small partitions and autonomy combined with partitioning and hierarchy may lead to robust scalability, diagnosis and online replacement methods which are fundamental in ensuring system properties such as performance and dependability including security and real time in systems such as NOMADs.

References

34.1 M. Malek. Omniscience, consensus, autonomy: Three tempting roads to responsiveness. In *Proceedings of the 14th Symposium on Reliable Distributed Systems, Keynote Address*, Bad Neuenahr, Sep 1995.

34.2 M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

34.3 M. Malek, M. Werner, and A. Polze. The unstoppable orchestra - a responsive distributed application. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 154–160, Annapolis, USA, May 1996. IEEE Computer Society Press.

34.4 IBM's Autonomic Computing Website. http://www.research.ibm.com/autonomic/.

34.5 M. Malek. Building autonomic systems with consensus and autonomy paradigms. Boeblingen, December 2001. Workshop on Autonomic Computing. Available on CD.

34.6 B. Gates. Bill Gates' e-mail on Trustworthy Computing, January 15, 2002. http://www.wired.com/news/business/0,1367,49826,00.html.

34.7 A. Polze, J. Richling, J. Schwarz, and M. Malek. Towards predictable corba-based services. In *Special Issue of International Journal of Computer Systems Science & Engineering*. CRL Publishing, London, 2001.

34.8 A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant corba-service. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS)*, Santa Barbara, USA, August 2000. IEEE Computer Society Press.

34.9 M. Barborak and M. Malek. Partitioning for efficient consensus. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, volume 2, pages 438–446, Maui, January 1993.

34.10 K. Jenkins and K. Birman. A gossip protocol for subgroup multicast. In *Proceedings of International Workshop on Applied Reliable Group Communication (WARGC 2001)*, Phoenix, Arizona, 2001.

35. Technology Challenges for the Global Real-Time Enterprise*

Werner Vogels

Dept. of Computer Science, Cornell University,
Upson Hall, Ithaca, NY 14853
vogels@cs.cornell.edu

35.1 Introduction

If there is one business concept that will drive distributed systems technology to the hilt in the coming years it is that of the real-time enterprise. The push for zero-latency access to a complete up-to-date view of all the business processes, inter-nally within a corporation, as well as to customers, will dominate the thinking of system architects for the years to come. Many of the technology components that need to become the building blocks for constructing and managing the real-time global information flow within large corporations do not exist yet. It is certain that issues such as security, robustness, manageability, combined with legacy system integration, and all enveloped with a scalability coating will be central in the dis-tributed system tools needed. It will require us to develop new technologies, re-package old ones, forge new tools and practices and to place these in an architec-tural vision in which many of these incompatible components can play together.

35.2 What Is the Real-Time Enterprise?

There is a strong push in business development to be able to react faster to changes within the corporation as well as in the world immediately around it. All of this is based on the notion that to remain competitive the enterprise needs to have a real-time view of operations internally, of the partners in the supply chain, of its customers interest and of the competition, all to enable a more proactive management.

We have seen that the enabling of real-time operations in the front-office has led to staggering problems in other parts of the operations. This is largely because most of the business process remains implemented as a batch process. Customers are able to inspect goods and order online, but are no longer able to change orders once they have been accepted. This may be acceptable for direct sales to the end user but for the services that Dell Computer wants to offer to large distributors such as Frye's, who order up to 10,000 PCs at the time, this is not acceptable. Dis-tributors must be able to change or augment their orders while they are being built, and the factory floor operation needs to be able to adapt to this instantly.

* This research is funded in part by DARPA/AFRL-IFGA through grant F30602-99-1-0532 and in part by grants from the Microsoft Corporation.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 197–201, 2003.

The pinnacle of real-time distributed business operations has for a long time been trading-floor automation. A matter of seconds was often the difference be-tween success and failure. One of the buzzwords used in this context is that of the zero-latency, as in general the latency associated with the batch process, which still drives large parts of the business operations, inhibits the enterprise to react swiftly to changes internally and externally. Not that the trading environments were perfect as even though the trades may actually placed in real-time, the com-pletion of the trade is a business process that can easily take up to 4-5 days be-cause of its batch style processing.

It is not only internally that enterprise needs to significantly improve its re-sponse times to events in the business process. For example the European low-cost airline ValueJet adjusts its fare pricing in real-time by monitoring the pricing of its competitors, to ensure that it offers the lowest fare, but not by too much. Almost the strongest push for real-time business operation comes from supply chain man-agement divisions. In the past 5 years automating large parts of the supply chain have enabled businesses to reduce inventories significantly, making them leaner and more profitable. However it has also left them more vulnerable to shortages and hick-ups in the supply chain and the lack of buffering forces them to respond to changes at their suppliers in real-time. An area where various experiments are underway is that of real-time Customer Relationship Management. The traditional approach of calculating "stable clus-ters" in a batch process at night is considered to be too conservative and does not meet the goals of the enterprise that wants to support customer business operations by online mechanisms. Instant monitoring of customer behavior to find relations with small variants such as changing a web-site background color or product packaging is considered essential for future adaptive on-line customer tracking. Targeted instant advertising as the customer is shopping online is considered to be one of the major growth areas for businesses.

It is not only traditional business that is focused on the reducing latency be-tween the different information processes. The US Military is driving an effort that should also enable it to respond more quickly to change, whether it is the lo-cation of spare parts or the availability of new intelligence information. The Joint Battlespace Infosphere (JBI) architecture is one of the efforts to enable the global real-time information systems [35.1].

Other key markets that are adopting the real-time enterprise as a future business model are for example the telecommunications industry which will use real-time techniques for fraud-detection, and the airline reservation industry where every-thing is still main-frame, transaction driven but planners need real-time views.

For the coming 5 years Gartner estimates that for the multinational enterprise between 30 and 60 percent of the total IT budget will need to be assigned for de-veloping real-time capabilities – equivalent to 1.6 and 3.2 percent of the total en-terprise revenue. For a $5 billion a year business this translates into $80 and $160 million a year [35.2].

35.3 The Different Faces of the Real-Time Enterprise

While I am focusing on the technical aspects that underpin the real-time enterprise one must understand that more than technology is needed to make the transforma-tion. Many

business processes will need to be changed or adopted, and ultimately the decision will be human actions who must also adapt to the new timescale of doing business.

The examples in the previous section show three major aspects to the evolution of a corporation into a real-time enterprise:

1. *Internal monitoring and information collection*: Each aspect of the business needs to be enabled to provide instant and up to date information to stake-holders, which maybe automated information fusion engines. Feedback cycles needs to be shortened and the control process needs to become adaptive to high-level directives.
2. *Acquiring and responding to external events*: The ability to be proactive in business is necessary to create a competitive edge. Essential is that information about competitors and partners is available in real-time
3. *Opening up the real-time process to partners and customers*: Customers, part-ners, investors, regulators, etc. will all demand access to real-time information from the corporation, which may have a different face depending on the re-ceiver.

35.4 The Technology Base

The real-time enterprise is by nature event driven. This leads us to believe that the event notification technology is the natural candidate to help us to construct a loosely coupled distributed architecture that is flexible and extensible enough to support the evolution of the enterprise into the real-time information world. Event based systems allow us to construct the various components independent of the new or original implementations, which gives us simple paths for software up-grades and the insertion of new information management components. Essential for the real-time enterprise is that we can dynamically add information fusion components that consume event flows and react to the events or managed state.

Legacy applications can simply be extended into generating events, as the avoidance of synchronous operations does not alter the process flow of the origi-nal application. Nor does the introduction of new or updated components alter the flows in the system.

35.5 Technology Challenges

Even though the scope and urgency of building the real-time enterprise are spec-tacular by itself, at first sight the technology side is not very challenging. We have been building event based systems for workflow management and pub-lish/subscribe systems for real-time data routing for some time now, and it appears a matter of technology integration to just extend these to other parts of the enter-prise.

The main and major challenge however is that of scale. The real-time enterprise needs to be active at a global scope using the internet or internet like networks to support its main connectivity. The heterogeneity in connectivity as well as partici-pant capability will require extremely flexible transport and content management techniques than that are currently not available in any single technology solution.

A second component of the scalability challenge will be that of sheer massive-ness of the amount of events generated at each moment anywhere in the global space of the enterprise. It will be impossible to take the traditional information bus approach as used in generic pub/sub systems and route the events anywhere in the enterprise. The architecture will need to include mechanisms for using information fusion components to reduce the overall event flow and to allow the system to move information fusion components throughout the network to places optimal to sources and consumers.

A third problem area related to scale is that the components can not predict the exact nature of the events they will need to deal with. In a global system where new event sources can come online each moment, new events type can be intro-duced any time. Requiring strict synchronization and registration will introduce too much coupling to build such a massive scalable system on, as such event proc-essors will need to be flexible in dealing with a variety of data sources. It is likely that one can draw somewhat from the experiences with content based subscription and routing techniques, but none of these have been really applied to large business cases and at such large scale it is likely that significant additional research needs to be performed.

The fourth main problem is that of the robustness of the overall system. If the overall success of a corporation becomes depended on the timeliness characteris-tics of the information architecture it becomes essential that the overall system op-erates within clear bounds. Failures and disturbances are unavoidable at any time in a global operation, but their effect on the overall operation should be limited. A few slow or uncooperative components should not drag the overall performance down, and the effects of the saturation of parts of the infrastructure should be lim-ited to its immediate locality and should not roll over into other areas. We will need to sort our refuge to probabilistic techniques for guaranteeing the data deliv-ery, using probabilistic redundancy to achieve the needed robustness of these criti-cal components. At the same time we need to establish mechanisms for assigning a reliability metric to information, such that fusion engines and/or the users can base their actions taking a certain level of uncertainty into account. Essential to the overall success will be the scalability of the event infrastruc-ture. Even though we can try to build the overall system as loosely coupled as pos-sible, some infrastructure components will need to have information about the overall system. The event routing infrastructure will need to be able to potentially route events from any producer to any consumer in the system, based on quite complex specifications. This cannot be done in a scalable manner by simply using traditional routing tables, especially since there may be many possible receivers for any single event. The solution here lies in the use of routing hierarchies were each virtual node in the hierarchy contains the aggregation of the routing informa-tion in its children tables. Each router contains the tables between itself and the virtual root such that any given event can be forwarded those local consumers it knows about or to other routers that represent consumers with an interest in this event. For a description of these mechanisms and the epidemic communications that makes the maintenance of these tables scalable see the paper on the Astrolabe by Robbert van Renesse [35.3].

Even though event driven systems seem to address the needs of the real-time enterprise quite closely, there are a number of non-real-time requirements that may impact the technology that will eventually be seen as the best foundation for a solution. One of

the related requirements is access to historical data. This can be an application driven requirement, for example when a context needs to be given for current events. It is also very likely that several legal requirements will exist that will force extensive logging and storage of event data. It is an active research area to determine where this data should be stored and how it can be accessed. Whether only raw data needs to be stored or that the data-fusion engines should also store their composite information streams, given that it may be difficult to re-play the causal relationships between the event streams. It is expected that relevant solutions will come out of the collaboration between distributed systems and data-mining researchers.

35.6 Related Work

There is a wealth of research and products that have generated technologies that will be part of the real-time enterprise architectures; Internet scale event messag-ing, high-performance content based routing, distributed subscription indexing, dynamic data-fusion engines, overlay networks, federated security, distributed data-mining, etc.

It would not do justice to the good work that has produced these technologies to provide a comparison or review at this point in the paper. What sets the new re-search directions apart from what has already been achieved is the focus on the problems of scale. In the coming years it will become clear that some of the tech-nologies cannot make the transition to a large-scale environment, while others will be able to adjust to heterogeneous demands of scale.

35.7 Summary

In this short paper I have tried to give a brief introduction into the scalability challenges the research and development community face to enable the global real-time enterprise of the future. Even though there is a strong push for corporations to transition their operations into real-time, the current technology is by no means ready to face the challenges of true global scalability.

References

35.1 United States Air Force Scientific Advisory Board. Report on building the joint bat-tlespace infosphere, volume 1: Summary. Technical report, SAB-TR-99-02, December 1999.

35.2 A. Drobrik. The challenges of the real-time enterprise. Technical report, Gartner Research Report AV-14-9268, November 2001.

35.3 Birman K.P. Renesse, R. and Vogels W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 2003. Accepted for publication.

36. Middleware for Supporting Inter-organizational Interactions

Santosh K. Shrivastava

School of Computing Science, Newcastle University, Newcastle-Upon-Tyne, UK

36.1 Understanding the Problem

We consider an emerging class of distributed applications that make use of a variety of Internet services provided by different organizations. This naturally leads to information sharing and interactions across organizational boundaries. However, despite the need to share information and services, organizations will want to maintain their autonomy and privacy. Organizations will therefore require their interactions with other organizations to be strictly controlled and policed. By this we mean that: (i) relationships between organizations for information access and sharing will need to be regulated by contracts, which will need to be defined and agreed and then enforced and monitored; and (ii) organizations will need to establish appropriate trust relationships with each other and implement corresponding security policies before permitting interactions to take place.

As an example, consider a company that runs a Marketplace application that enables buyers to order goods and services from various suppliers. The process of ordering includes: requisition; agreement; delivery and payment. The company running the Marketplace would like to monitor the process to ensure that the company policy is adhered to (for example, that a buyer is credit-worthy) and that agreements between the parties are observed (for example, that the supplier does not arbitrarily modify an order). There is also a requirement that payment is made if and only if the items or services ordered are delivered. For simple orders, this last aspect of the process is the most significant. When the ordering process is more complex, requisition and agreement can acquire greater significance. For example: requisition may include a procurement process involving multiple parties; there may be a need to negotiate non-standard terms and conditions; order fulfillment may entail commitments from more than one supplier or from delivery agents; or the order may govern delivery of an on-going service that should itself be regulated. In these more complex cases it is arguable that business can be better supported if the organizations involved are able to share information.

Figure 36.1 shows the various organizations and the corresponding contractual relationships. The company providing the Marketplace has set up contracts with its customers (buyers and suppliers) and with companies whose services it needs; in the figure we show just two, a trusted third party (TTP) to meet the security requirements of buyers and suppliers (such as certificate management) and a credit rating agency. The credit rating agency itself is shown implementing its services using data obtained from account history services provided by retail banks.

We are assuming that the organizations involved might not trust each other, so an important part of our work is concerned with developing infrastructure support (middleware services) for interactions between two or more mutually distrusting organizations

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 202–206, 2003.

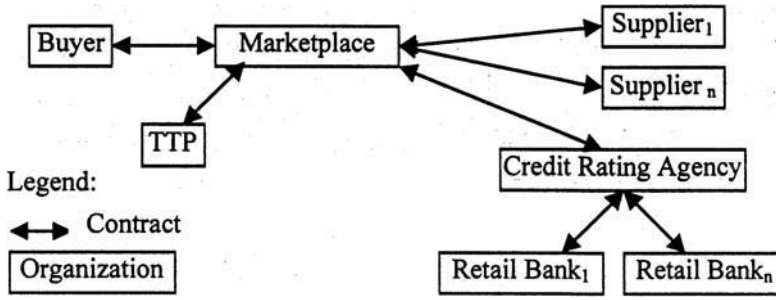

Fig. 36.1. Marketplace application

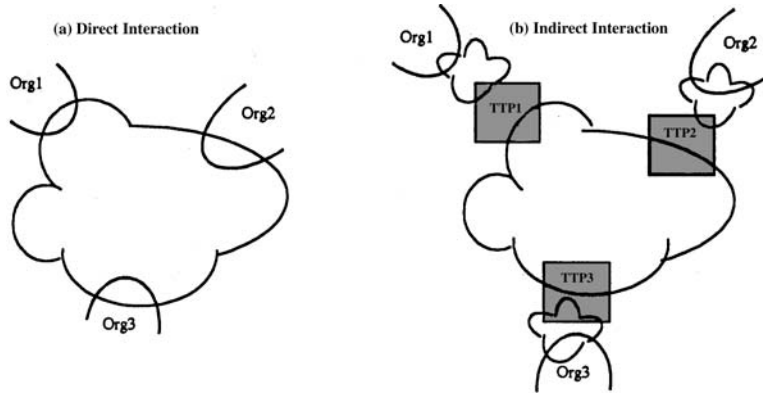

Fig. 36.2. Direct and indirect interactions

(see [36.1, 36.2] for a discussion on trust related issues). Figure 36.2 shows two basic interaction styles, where a cloud represents middleware services shared between the organizations: (a) organizations trust each other sufficiently to interact directly with each other; and (b) no direct trust between the organizations exist, so interactions take place through TTPs acting as intermediaries. Referring to figure 36.1, a given contractual relationship could be implemented by any one of the two interaction styles. It may be that interactions initially take place through TTPs and once sufficient trust has been established, organizations agree to interact directly. The interactions involved can vary in nature:

1. an interaction may be short- or long-lived: from fulfilment of an order to provision of a complex, evolving service;
2. interactions may be composed from sub-interactions: service delivery may involve the sub-contracting of obligations to other parties whilst ensuring compliance with the obligations imposed by the “parent" interaction;
3. interactions may be dynamic: the rules governing the interaction may be subject to re-negotiation and the parties to the interaction may change;
4. the relationship between parties may be asymmetric (customer-supplier) or symmetric (peer-to-peer).

The infrastructure should support interactions in a (potentially) hostile environment: interactions between organizations over open networks. Given these observations, development of the middleware for supporting inter-organization interactions pose very interesting research problems.

36.2 Approaches and Research Directions

We would like to know if it is possible to come up with a reasonably small set of generic services that can be used to support arbitrarily complex interactions between organizations. As indicated earlier, in figure 36.2, the clouds represent the places where such middleware will be deployed. What does it mean for inter-organizational interactions to be 'strictly controlled and policed'? From the viewpoint of each party involved, the overarching requirements are (i) that their own actions meet locally determined policies; and that these actions are acknowledged and accepted by other parties; and (ii) that the actions of other parties comply with agreed rules and are irrefutably attributable to those parties. These requirements imply the collection, and verification, of non-repudiable evidence of the actions of parties who interact with each other.

We have built an experimental middleware platform that implements the abstraction of information sharing between organizations [36.3]. It is assumed that each organization has a local set of policies for information sharing that is consistent with the overall information sharing agreement between the organizations (this agreement can be viewed as a business contract between organizations). The safety property of our system ensures that local policies of an organization are not compromised despite failures and/or misbehavior by other parties; whilst the liveness property ensures that if all the parties are correct (not misbehaving), then agreed interactions would take place despite a bounded number of temporary network and computer related failures.

Essentially, our middleware resembles a transactional object replica management system where each organization has a local copy of the object(s) to be shared. Any local updates to the copy by an organization ("proposed state changes" by the organization) are propagated to all the other organizations holding copies in order for them to perform local validation; a proposal comprises the new state and the proposer's signature on that state. Each recipient produces a response comprising a signed receipt and a signed decision on the (local) validity of the state change. All parties receive each response and a new state is valid if the collective decision is unanimous agreement to the change. The signing of evidence generated during state validation binds the evidence to the relevant key-holder. Evidence is stored systematically in local non-repudiation logs.

There are a number of ways of extending this work further. We would like to map well-specified ACID and non-ACID transaction models onto our middleware services. Our middleware is currently intended for supporting information sharing where state must be disclosed to other parties to be validated. If state disclosure is conditional, then there does not appear to be any way out other than to make use of TTPs through which information exchange takes place 'fairly'. A protocol is fair if no protocol participant can gain any useful advantage over the other participant by misbehaving. Fair exchange protocols that make use of a TTP have been studied extensively in the literature [36.4, 36.5]; these protocols maintain fairness even if the dishonest participant can tamper with

the execution of the protocol in an unrestricted (malicious) manner. How to incorporate these ideas into our middleware is as yet not clear to us.

At a higher level (above this middleware), one would like services for contract management. In the paper-based world, a contract is a document that stipulates that the signatories agree to observe the clauses stipulated in the document. Each entry in the document is called a term or a clause, which lists the rights and obligations of each signing party. Contract management services should therefore provide ways for representing contracts in terms of rights and obligations so that they can be enforced and monitored. Recent work on 'law-governed interaction' [36.6, 36.7] and electronic contract management [36.8, 36.9, 36.10] indicates possible ways of representing contracts and ensuring that inter-organizational interactions are consistent with those contracts.

Contract management must be made part of the business processes of the organizations involved. An organization's business processes can be divided into two broad categories. The business processes that are internal to the organization and the 'contract management processes' that involve interactions with trading partners. One has to look at issues of cross-organizational workflow management; some recent papers in this area are good starting points [36.11, 36.12]. A difficult problem is that of coordinating multiple workflows in a decentralised manner. Most commercial workflow systems are inherently centralised. A way out is to use a workflow system with decentralised coordination (e.g., [36.13]) for managing just the inter-organizational workflows.

Acknowledgements

This work is part-funded by the UK EPSRC under grant GR/N35953: "Information Co-ordination and Sharing in Virtual Enterprises", the European Union Project IST-2001-34069: "TAPAS (Trusted and QoSAware Provision of Application Services)", and project GridMist funded by DTI and Hewlett-Packard under the UK e-science initiative. Discussions with colleagues Paul Ezhilchelvan, Nick Cook, Carlos Molina-Jimenez, Stuart Wheater, Ellis Solaiman and John Warne have been very helpful.

References

36.1 T. Grandison and M. Sloman, "A survey of trust in Internet applications", IEEE Communications Surveys, Fourth Quarter 2000, www.comsoc.org/pubs/surveys.

36.2 D.D. Clark and M.S. Blumenthal, "Rethinking the Design of the Internet: the End to End Arguments vs. the Brave New World", ACM Trans. On Internet Technology, 1,1, August 2001.

36.3 N. Cook, S.K. Shrivastava and S.M. Wheater, "Distributed Object Middleware to Support Dependable Information Sharing between Organizations", IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2002), June 2002, Washington DC, pp.249-258.

36.4 N. Asokan., V. Shoup and M. Waidner, "Asynchronous protocols for optimistic fair exchange", Proc. IEEE Symposium on Research in Security and Privacy, pp. 86-99, 1998.

36.5 H. Vogt, H. Pagnia, and F. Gärtner. Modular Fair Exchange Protocols for Electronic Commerce. In Proc. IEEE Annual Comput. Security Applications Conf., Phoenix, Arizona, Dec. 1999.

36.6 N.H. Minsky and V. Ungureanu, “A mechanism for establishing policies for electronic commerce", The 18th Intl. Conf. on Distributed Computing Systems (ICDCS-18), Amsterdam, May 1998, pp. 322-331.

36.7 N.H. Minsky and V. Ungureanu, “Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems", ACM Trans. on Software Engineering and Methodology, 9(3), July 2000, pp. 273-305.

36.8 O. Marjanovic and Z. Milosevic, “Towards formal modelling of e-contracts", Proc. of 5th IEEE/OMG International Enterprise Distributed Object Computing Conference (EDOC 2001), September 2001, pp. 59-68.

36.9 A. Abrahams, D. Eyers, and J.M. Bacon, “A coverage-determination mechanism for checking business contracts against organizational policies", Proceedings of the Third VLDB Workshop on Technologies for E-Services (TES’02), 2002.

36.10 C. Molina-Jimenez, S.K. Shrivastava, E. Solaiman and J. Warne, “Contract Representation for Run-time Monitoring and Enforcement", Tech. Report, School of Computing Science, Newcastle University, December 2002, http://distribution.cs.ncl.ac.uk/publications/

36.11 H. Ludwig and K. Whittingham, “Virtual enterprise co-ordinator - agreement driven gateways for cross-organizational workflow management", ACM Software Engineering Notes, 24(2), PP. 29-37, March 1999 (Proc. of Work Activity Coordination Conference, WACC’99)

36.12 Rosettanet implementation framework: core specification, V2, Jan 2000. http://rosettanet.org

36.13 S.M. Wheater, S.K. Shrivastava and F. Ranno “A CORBA Compliant Transactional Workflow System for Internet Applications", Proc. Of IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 98, (N. Davies, K. Raymond, J. Seitz, eds.), Springer-Verlag, London, 1998, ISBN 1-85233-088-0, pp. 3-18.

37. Hosting of Libre Software Projects: A Distributed Peer-to-Peer Approach*

Jesús M. González-Barahona and Pedro de-las-Heras-Quirós

Universidad Rey Juan Carlos, Grupo de Sistemas y Comunicaciones
Móstoles, Spain
{jgb,pheras}@gsyc.escet.urjc.es

Summary. While current hosting systems for libre software projects are mainly centralized or client-server systems, several benefits arise from using distributed peer-to-peer architectures. The peculiarities of libre software projects make them a perfect test-bed for experimenting with this architecture. Among those peculiarities we can mention: the clear need for decentralization, the highly distributed nature of the user base, the high reliability requirements and the rich set of interactions among users and developers. Here we present a first attempt to describe the characteristics and complexities of this application area, and the expected future developments.

37.1 Hosting Services for Libre Software Projects

Libre software[1] projects are unique in several ways. One of the most noticeable is the distributed nature of their developer and user base. Most libre software projects are managed by groups of geographically distributed developers who design, build and maintain the software. They use Internet, almost exclusively, for communication and coordination.

Traditionally, libre software projects set up and maintained their own tools and resources for the management of the software, including communication systems (mail list managers, IRC servers), information downloading systems (ftp and www servers), version control systems (usually CVS), bug tracking systems, etc.

During the last few years, the trend has been to move to centralized hosting services for libre software projects. They provide the same set of resources, and integrate them as much as possible. Some of them are really huge (for instance, SourceForge had in early 2002 more than 32,000 registered projects and well above 300,000 registered users, mostly developers [37.1]), and suffer the usual problems of centralized services: unique point of control and failure, bottlenecks, etc.

Due both to technical and 'political' reasons [37.2]) some groups have begun to explore how to make those systems more distributed using peer-to-peer models [37.3]), evolving towards a network of nodes providing hosting services. There are several advantages in this approach, and the model fits well with the common practices in the libre

* Work supported in part by CICYT grants No. TIC-98-1032-C03-03 and TIC2001-0447 and by Comunidad Autónoma de Madrid, grant No. 07T/0004/2001

[1] We use the term *libre software* to refer to software which complies with the usual definitions for *free software* and *open source software*.

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 207–211, 2003.

software community. In short, individuals and organizations would volunteer some resources (by installing some software in their machines), which as a whole would provide services similar to those of the current hosting services for software development. The aim is to use the computational resources provided by the community.

This approach combines ideas from existing file sharing or distributed storage systems (Gnutella [37.4], OceanStore [37.5], CFS [37.6], PAST [37.7], etc.), CPU sharing systems (Distributed.Net [37.8] and SETI@Home [37.9]), or P2P instant messaging systems [37.10]. But the technical problems to be addressed are peculiar, if only because they include instances of those found in many of the above systems (although with particular characteristics).

37.2 Services Offered by a Project Hosting Service

The aim of a libre software project hosting service (from now on, a project hosting service) is to provide as much resources as possible to the hosted projects. The most well known example (SourceForge) provides the following services [37.11]:

- Version control (via CVS repositories), including several modes of access and simple, web-based interfaces to browse the files.
- Management support, providing tools for bug tracking, patch management, request management, etc.
- Information services to publish the results of the project, and to coordinate the developer team and inform the users (web pages, forums, mailing lists, event-tracking, etc.)
- Release services, to make the produced software available to end-users (via FTP and HTTP).
- Compile & execution farm.
- Directory services, to locate projects with certain characteristics.
- Administration interfaces to manage all the aspects of a given project, including statistical analysis of several parameters.

These services are frequently integrated. For instance, a user of the software produced by a project can set a tracker to know about new releases, or when a bug has been fixed. Developers can relate a given released software with the files in the version control system. The management of the hosting service is highly automated, but still requires a non-negligible amount of human labor for daily maintenance. This kind of service has proven to scale well to the tens of thousands of projects, and to the tens of millions (if not hundreds of millions) of lines of source code.

37.3 Making It Peer-to-Peer Distributed

The goal of a peer-to-peer project hosting service would be to have a network of nodes capable of providing the services described in the above section, and maybe some more. The human labor needed to maintain the infrastructure should be reduced to a minimum,

and all the coordination, resource allocation and management tasks should be completely automated.

When somebody would want to share all or part of the computational resources of a machine with the hosting network, she would just install and configure some software on its computer. The installation would be almost automatic, except for some parameters to be chosen during configuration (like bandwidth or disk space or CPU cycles to devote to the hosting system). Once this is done, the node would join the peer-to-peer network, and start to share its resources. The complexity of this process should be similar to the installation of a SETI@Home node or a Gnutella servent [37.4].

Some of the services outlined in the previous section could be implemented using existing peer-to-peer systems. For instance, release services can be built on top of Gnutella or Freenet [37.12]. Version control systems are evolving into highly distributed services (see arch [37.13] and CPCMS/OpenCM [37.14]). Standards for exchanging information about libre software projects are already being defined (like CoopX [37.15]). Jabber [37.10] is establishing itself as a de-facto standard for peer-to-peer instant messaging, a service not mentioned in the previous list of services, but rather useful for project hosting services.

However, a lot of work has yet to be done at least in the definition of requirements of the new peer-to-peer hosting services and in the evaluation of the suitability of existing proposals in the peer-to-peer research community in the following areas: search, replication, resource management, security and privacy.

Some of the problems which should be addressed are:

- Security and privacy model. The information provided by the hosting service must be that provided by the projects, and not by other parties. On the other hand, the security model should not impose too much performance penalties, or it would not be useful at all. Accountability, reputation and information preservation are also important aspects of any libre software project. Fortunately, relatively simple approaches based on signatures and chains of confidence could be enough to satisfy those needs.
- The hosting service should be immune to node failures, and should tolerate nodes non-permanently connected. Data and service replication would thus be needed, and done in an automatic and unmanaged way. Fortunately, in most services probabilistic approaches would provide enough availability, given a large enough set of providers (consider for instance, for the software distribution service, that any of the machines having installed a given package, potentially millions, can provide access to such a package).
- Most current peer-to-peer systems are focused on file/data or CPU sharing, but project hosting services must also provide other functionality (see services described in previous section) which has not received as much attention in the research community until now. And data sharing could be addressed in a different way to projects such as OceanStore, since here every client that wants any data is probably willing to provide it to future requesters (which may lead schemas based in large active proxies).
- The scale of this system could be very large, with millions of users willing to download released software, hundreds of thousands of developers working on their projects, and probably at least hundreds of thousands of nodes polling resources to make up the system. However, some parameters won't be very large, and the system can take

advantage of that fact. For instance, for any given project, the set of developers is usually not very large, and only they need write access to most portions of the service (which leads to models with little writers and a lot of readers, more simple to deal with in terms of consistency).

– Resource management (CPU, storage and bandwidth) is an important issue to be addressed. The system must provide a distributed, dynamic and automatic configuration of resources needed by different projects. Trading mechanisms should be evaluated as the way to guide the allocation of resources to the needs of hosted projects. The matching of resources needed by different projects with the resources offered by peers could be regulated based on parameters such as the amount of recent activity of a given project, the size and topological situation of the developer/user base, size of traded files, frequency of releases of the projects, availability of peer nodes, etc.

In addition, some other more general requirements must be satisfied, like efficient service discovery and data routing on an overlay peer-to-peer network (for which existing peer-to-peer service such as JXTA [37.16, 37.17] have to be evaluated), performance (comparable to that of current centralized services), etc. However, they do not offer new characteristics, and could be dealt with using techniques borrowed from other peer-to-peer systems which have already addressed them.

In some domains, naive solutions could be good enough. For instance, simple protocols like Gnutella have been shown to scale reasonably well to thousands of nodes when the set of searchable data is not very high (which could be a good guess for software package distribution). But in most cases, such simple designs will not be enough to address all the requirements. In fact, the most important design factor could be the tradeoff between existing, stable designs and more advanced, but maybe too uncertain ideas.

37.4 Conclusions

Some sectors of the libre software community have identified the centralized nature of services like SourceForge as a problem. They are starting to design and implement hosting services for libre software projects as networks of nodes donating resources. Participation in the network should be simple, and management of the basic infrastructure almost null.

The requirements of this kind of system make it one of the most demanding peer-to-peer distributed application domains, and therefore it is a very interesting and realistic environment to test research ideas.

Trying to foretell the future, it can be expected that the development will be incremental, not fulfilling all the requirements in the first versions. But given the precedents of the libre software community, once this architecture has been identified as a target, a lot of activity is expected in the following years. Being a domain in which developers are directly involved (to the point that they are not only their developers but also a part of their users), it is likely that this project will not be abandoned once it begins.

References

37.1 SourceForge. SourceForge.Net update. 2002-01-22 edition, January 2002.

37.2 Loïc Dachary. SourceForge drifting, 2001. http://fsfeurope.org/news/article2001-10-20-01.en.html.

37.3 Loïc Dachary. Savannah the next generation, September 2001. http://savannah.gnu.org/docs/savannah-plan.html.

37.4 Clip2. The Gnutella protocol specification 0.4. Document revision 1.2. http://www.clip2.com/GnutellaProtocol04.pdf.

37.5 Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September/October 2001.

37.6 Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

37.7 P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, Schloss Elmau, Germany, May 2001.

37.8 Brian Hayes. Collective wisdom. *American Scientist*, March-April 1998.

37.9 Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@Home: Massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1), 2001.

37.10 J. Miller, P. Saint-Andre, and J. Barry. Jabber. Internet draft, February 2002. http://www.jabber.org/ietf/draft-miller-jabber-00.html.

37.11 SourceForge. SourceForge services, 2001. http://sourceforge.net/docman/display_doc.php?docid=753&group_id=1.

37.12 Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandbergand, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, pages 40–49, January-February 2002.

37.13 Thomas Lord. The arch revision control system, 2001. http://regexps.com/src/docs.d/arch/html/arch.html.

37.14 Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *2002 USENIX Annual Technical Conference, FREENIX Track*, 2002.

37.15 Grant Bowman, MJ Ray, Loic Dachary, and Michael Erdmann. A conceptual framework for the distribution of software projects over the Internet, 2001. http://home.snafu.de/boavista/coopx/coopx.html.

37.16 Li Gong. JXTA: A network programming environment. *Internet Computing Online*, 5(3):88–95, May-June 2001.

37.17 Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA virtual network, February 2002. http://www.jxta.org/docs/JXTAprotocols.pdf.

38. System Support for Pervasive Applications

Robert Grimm[1] and Brian Bershad[2]

[1] New York University, New York, NY
rgrimm@cs.nyu.edu
[2] University of Washington, Seattle, WA
bershad@cs.washington.edu

38.1 Introduction

Pervasive, or ubiquitous, computing [38.4] has the potential to radically transform the way people interact with computers. The key idea behind pervasive computing is to deploy a wide variety of computing devices throughout our living and working spaces. These devices coordinate with each other and network services, with the goal of seamlessly assisting people in completing their tasks. Pervasive computing thus marks a major shift in focus, away from the actual computing technology and towards people and their needs.

To illustrate this shift in focus, consider, for example, researchers working in a biology laboratory. Their goal is to perform reproducible experiments. Yet, today they manually log individual steps in their paper notebooks. This easily leads to incomplete experimental records and makes it unnecessarily hard to share data with other researchers, as the biologists need to explicitly enter the data into their PCs. In contrast, a digital laboratory employs digitized instruments, such as pipettes and incubators, to automatically capture data, location sensors to track researchers' movements, and touch-screens throughout the laboratory to display experimental data close to the researchers. As a result, biologists working in the digital laboratory have more complete records of their experiments and can more easily share results with their colleagues.

For this vision to become a reality, we need system support that directly helps with building, deploying, and using pervasive applications. However, contemporary system services typically assume a static and well-administered computing environment. Furthermore, they tend to hide distribution from applications, with the result that, if changes occur, people need to adapt the system instead of the applications adapting for them. For example, with contemporary systems it is hard to move between machines, as people need to manually log in, start their applications, and load their documents. Similarly, it is hard to integrate new devices, as people need to first configure them, for example, to use the correct wireless network parameters. Furthermore, it is hard to share data, as people need to explicitly manage shared files and convert between different formats.

We argue that, to be successful, system support for pervasive computing needs to address three major requirements. First, as people move throughout the physical world—either carrying their own portable devices or switching between devices—an application's location and execution context changes all the time. As a result, system support needs to *embrace contextual change* and not hide it from applications. Second, users expect that their devices and applications just plug together. System support thus needs to *encourage ad hoc composition* and not assume a static computing environment with

A. Schiper et al. (Eds.): Future Directions in DC 2002, LNCS 2584, pp. 212–217, 2003.

a limited number of interactions. Third, as users collaborate, they need to easily share information. As a result, system support needs to *facilitate sharing* between applications and devices.

We also argue that it is not sufficient to create new system services that address the three requirements and layer them as middleware on top of existing systems. Rather, system support for pervasive computing needs to be designed from the ground up to embrace change, encourage ad hoc composition, and facilitate sharing. Sun's Jini [38.1], for example, provides distributed events and discovery, both of which are useful for building more adaptable applications. However, Jini is layered on top of Java RMI, a distributed object system that is targeted at static and well-administered computing environments. In particular, Java RMI requires a statically configured infrastructure to run name and, for Jini, discovery servers. Furthermore, Java RMI requires that applications are well-behaved, as it relies on synchronous remote invocations, provides no isolation between objects, and closely couples devices to each other (RMI's distributed garbage collection controls an object's lifetime, thus making one device dependent on other devices on the network). So, to avoid the short-comings of existing systems, we need to create the right services from scratch.

38.2 Architecture of *one.world*

To provide the appropriate system support for pervasive applications, we have designed and implemented a new system architecture for pervasive computing [38.3]. Our architecture, called *one.world* and illustrated in Figure 38.1, is centered around meeting the requirements of embracing change, encouraging ad hoc composition, and facilitating sharing. It employs a classic user/kernel split, with foundation and system services running in the kernel, and libraries, system utilities, and applications running in user space. *one.world*'s foundation services directly address the individual requirements. They also provide the basis for our architecture's system services, which, in turn, serve as common building blocks for pervasive applications.

The four foundation services are a virtual machine, tuples, asynchronous events, and environments. First, all code in *one.world* runs within a virtual machine, such as the Java virtual machine or Microsoft's common language runtime. Because of the inherent heterogeneity of pervasive computing environments, developers cannot possibly predict all devices their applications will run on, and the virtual machine ensures that applications and devices are composable. Second, all data in *one.world* is represented as tuples. Tuples define a common data model, including a type system, for all applications and thus simplify the sharing of data. They are records with named and optionally typed fields, and they are self-describing, so that an application can dynamically inspect a tuple's structure and contents. Third, all communications in *one.world*, whether local or remote, are expressed through asynchronous events. They provide the means for explicitly notifying applications of changes in their runtime context.

Finally, environments are the main structuring mechanism for *one.world*. They represent the units of local computation and, just like traditional operating system processes, isolate applications from each other. Environments also serve as containers for persistent data. In addition to hosting application code, they provide associative tuple storage

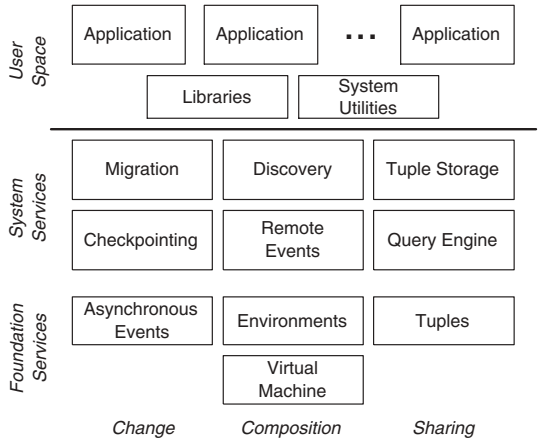

Fig. 38.1. Overview of *one.world*'s architecture.

and thus make it possible to directly group applications with their data. Furthermore, environments can be nested within each other, making it easy to compose applications. An outer environment has complete control over all nested environments, including the ability to interpose on inner environments' communications with *one.world*'s kernel and the outside world. Overall, environments can be thought of as a combination of file system directories and nested processes in other operating systems.

Table 38.1. Application needs and corresponding system services.

Applications need to...	***one.world* provides...**
Search	Query engine
Store data	Tuple storage
Communicate	Remote events
Locate	Discovery
Fault-protect	Checkpointing
Move	Migration

one.world's system services build on the foundation services and serve as common building blocks for pervasive applications. Specific application needs and the corresponding system services are summarized in Table 38.1. Out of these services, discovery and migration are probably the most interesting.

Discovery locates resources, that is, event handlers, by their descriptions. It leverages *one.world*'s uniform data model, in which all data, including events and queries, are tuples, to support a rich set of options, including early and late binding as well as anycast and multicast, with only three simple operations. More importantly, because discovery is an essential service for pervasive applications—after all, without discovery, applications cannot adapt to a new or changing runtime context—discovery is self-managing and relies on a server that is elected from all devices running *one.world* on the

local network. To ensure availability, elections are called aggressively and complete after a fixed period. The individual devices tolerate any resulting inconsistencies by making their discoverable resources accessible through all servers while looking up resources on only one server.

Migration moves or copies an environment and all its contents to a different device, thus simplifying the implementation of applications that follow a person through the physical world. Unlike traditional process migration, *one.world*'s migration is *not* transparent, and migrated application state is limited to the environments being migrated. References to resources outside the environment tree are automatically nulled out during migration, which is acceptable because applications already expect change. As a result, *one.world*'s migration can avoid the complexities of traditional process migration, and migration across the wide area becomes practical.

Overall, *one.world* has been specifically designed to support adaptable applications, with the goal that users do not need to manually adapt their systems. While several services have been explored before, our architecture's services differ in that they have been built from the ground up to embrace change, encourage ad hoc composition, and facilitate sharing.

Experiences. To evaluate *one.world*, we developed several applications, including a text and audio messaging system. We also recruited outside developers. Notably, the Labscape project at the University of Washington ported their electronic laboratory assistant, which works as outlined in the introduction and has been deployed at the Cell Systems Initiative, to *one.world* [38.2]. In contrast to an earlier version, which was implemented using Java sockets and their own application-specific migration, the *one.world* port required less than half the development time, has an order of magnitude faster migration times, two orders of magnitude longer mean time between failures, and can recover piecemeal from failures instead of requiring a restart of the entire lab. While the Labscape team also had some complaints about our event model and the lack of integration with Internet services, the resulting application is good enough to be used by biologists every day and thus establishes *one.world* as solid foundation for building pervasive applications.

38.3 Outlook

We see two major thrusts for future research into system support for pervasive applications, with the first aimed at creating more advanced system services and the second aimed at changing the way we build pervasive applications.

While *one.world*'s system services meet basic application needs, we see three areas for providing additional system services. First, we need better support for reflecting an application's runtime environment to the application, including an ontology for describing device characteristics, network connectivity, and location. For example, *one.world* provides only limited information about a device's capabilities (such as speed and memory capacity) and none about the current level of connectivity. Yet, remaining energy for battery operated devices and cost for network connectivity are important factors when deciding, for example, whether to migrate or to communicate. The main challenge here is to develop an appropriate ontology and the corresponding software sensors.

Second, we need better support for synchronizing and streaming data between devices. While we have experimented with a replication layer that supports disconnected operation and with streaming audio between devices, our current implementations are not sufficiently tuned and not fully integrated with other system services. More importantly, *one.world* does not provide any support for dynamically transforming data, for example, to reduce a video stream's fidelity when sending it across a low bandwidth cellular link. The main challenge here is not how to transform streaming video, but rather how to provide a unified framework for both discrete data (such as experimental data or personal contacts) and streaming data (such as audio or video).

Third, we need better support for changing and upgrading an application's code. *one.world*'s migration can already be used to easily install an application on a new device by simply copying it. However, our architecture does not provide the ability to upgrade applications while they are running. Since many pervasive applications can be expected to be long running, being able to upgrade such applications without disrupting them is an important capability. The main challenge here is to design a mechanism that is automatic, secure, and general enough to also upgrade the system platform itself.

While system support, such as that provided by *one.world*, can simplify the task of developing pervasive applications, developers still need to program all application behaviors by hand. We believe that a higher-level approach is needed to help developers be more effective.

To this end, we turn to the web for inspiration and observe that it uses two related technologies, declarative specifications and scripting, to great effect. For example, web servers typically rely on declarative configuration files that specify how to map a server's virtual name space to actual resources and how to process and filter content. Similarly, web browsers rely on style sheets to specify the appearance of accessed web pages. In both cases, more advanced behaviors are typically expressed through scripts that are embedded in web pages. The Mozilla project's XML-based user interface language (XUL) pushes these two technologies even further and relies on them to express an application's entire user interface.

Based on our experiences with *one.world*, which suggest that application-specific policies can be directly implemented on top of our architecture's services, we believe that a similar approach can be applied to the development of pervasive applications. For example, this approach can be used to specify policies for migrating pervasive applications or data integrity constraints for replicated storage. The key insight is that a declarative specification can provide a concise description of a system's properties, which can then automatically be translated into appropriate actions. This approach thus represents a push towards specifying a system's goals instead of programming its behaviors. In effect, it treats a pervasive systems platform, such as *one.world*, as the assembly language for implementing complex behaviors. As such, it holds the potential to significantly simplify the development of complex systems.

Acknowledgments

In addition to the authors of this paper, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Daniel Cheah, Ben Hendrickson, Tom Anderson, Gaetano Borriello,

Steven Gribble, and David Wetherall contributed to the development of *one.world*. More information on our architecture, including a source release, is available at `http://one.cs.washington.edu`.

References

38.1 K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

38.2 L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, S. B. Sigurdsson, J. Su, and G. Borriello. Systems support for ubiquitous computing: A case study of two implementations of Labscape. In *Proceedings of the 2002 International Conference on Pervasive Computing*, Zurich, Switzerland, Aug. 2002.

38.3 R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. Programming for pervasive computing environments. Submitted for publication, Jan. 2002.

38.4 M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

Author Index