Lecture Notes in Computer Science 1914
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Maurice Herlihy (Ed.)

Distributed Computing

14th International Conference, DISC 2000
Toledo, Spain, October 4-6, 2000
Proceedings

Springer

Series Editors

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

Volume Editor

Maurice Herlihy
Brown University, Department of Computer Science
115 Waterman Street, Providence, RI 02912, USA
E-mail: herlihy@cs.brown.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Distributed computing : 14th international conference ; proceedings /
DISC 2000, Toledo, Spain, October 4 - 6, 2000. Maurice Herlihy (ed.).
- Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ;
Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1914)
ISBN 3-540-41143-7

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1, D.4.4-5

ISSN 0302-9743
ISBN 3-540-41143-7 Springer-Verlag Berlin Heidelberg New York

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

Printed in Germany

Typesetting: Camera-ready by author
Printed on acid-free paper SPIN 10722751 06/3142 5 4 3 2 1 0

Preface

DISC, the International Symposium on DIStributed Computing, is an annual forum for research presentations on all facets of distributed computing. DISC 2000 was held on 4 - 6 October, 2000 in Toledo, Spain. This volume includes 23 contributed papers and the extended abstract of an invited lecture from last year's DISC. It is expected that the regular papers will later be submitted in a more polished form to fully refereed scientific journals. The extended abstracts of this year's invited lectures, by Jean-Claude Bermond and Sam Toueg, will appear in next year's proceedings.

We received over 100 regular submissions, a record for DISC. These submissions were read and evaluated by the program committee, with the help of external reviewers when needed. Overall, the quality of the submissions was excellent, and we were unable to accept many deserving papers.

This year's *Best Student Paper* award goes to "Polynomial and Adaptive Long-Lived $(2k-1)$-Renaming" by Hagit Attiya and Arie Fouren. Arie Fouren is the student author.

October 2000 Maurice Herlihy

Organizing Committee

Chair:	Angel Alvarez (U. Polytécnica de Madrid)
Treasurer:	Sergio Arévalo (U. Rey Juan Carlos)
Publicity:	Pedro de-las-Heras-Quirós (U. Rey Juan Carlos)
	Vicente Matellán-Olivera (U. Rey Juan Carlos)
Registration/Hotel:	Ricardo Jiménez (U. Polytécnica de Madrid)
	Marta Patiño (U. Polytécnica de Madrid)
Transportation:	Antonio Fernández (U. Rey Juan Carlos)
	Jesús M. González-Barahona (U. Rey Juan Carlos)
Communications:	Francisco Ballesteros (U. Rey Juan Carlos)
	José Centeno (U. Rey Juan Carlos)
On-site support:	Juan Carlos López (U. de Castilla-La Mancha)
	Francisco Moya (U. de Castilla-La Mancha)
	José Manuel Moya (U. de Castilla-La Mancha)

Steering Committee

Faith Fich (U. of Toronto)
Maurice Herlihy (Brown U.)
Prasad Jananti (Dartmouth)
Shay Kutten (Technion)
Michel Raynal (IRISA)
Andre Schiper (EPF Lausanne)
Shmuel Zaks (chair) (Technion)

Program Committee

Rida Bazzi (Arizona State U.)
Ajoy Datta (U. of Nevada)
Peter Dickman (U. of Glasgow)
Panagiota Fatourou (Max-Planck Inst.)
Paola Flocchini (U. of Ottawa)
Maurice Herlihy (chair) (Brown U.)
Lisa Higham (U. of Calgary)
Christos Kaklamanis (CTI)
Idit Keidar (MIT)
Michael Merritt (ATT Laboratories)
Enrico Nardelli (L'Acquila)
Luis Rodrigues (U. of Lisbon)
Peter Ruzicka (Comenius U.)
Assaf Schuster (Technion)
Mark Tuttle (Compaq Research)
Roger Wattenhofer (Microsoft Research)
Jennifer Welch (Texas A&M)
Peter Widmayer (ETH Zurich)

Outside Referees

Nancy Amato
Tal Anker
Hagit Attiya
Amotz Bar-Noy
Ioannis Caragiannis
Srinivas Chari
Gregory Chockler
Giorgio Delzanno
Adriano Di Pasquale
Dulce Domingos
Wayne Eberly
Michael Factor
Fabio Fioravanti
Michele Flammini
Luca Forlizzi
Maria Gradinariu
Michael Greenwald
Rachid Hadid
Colette Johnen
Yuh-Jzer Joung
Shmuel Katz
Jalal Kawash
Roger Khazan
Zhiying Liang
Carl Livadas
Victor Luchangco
Frederic Magniette
Giovanna Melideo
Mark Moir
Henry Muccini
Evi Papaioannou
Andrzej Pelc
Franck Petit
Benny Pinkas
Guido Proietti
Sergio Rajsbaum
Michel Raynal
Ohad Rodeh
Alessandro Roncato
Fabrizio Rossi
Eric Ruppert
Nicola Santoro
David Semé
Alex Shvartsman
Ioannis Stamatiou
Jeremy Sussman
Sébastien Tixeuil
Gil Utard
Vincent Villain

Table of Contents

Lower Bounds in Distributed Computing

Faith Fich[1] and Eric Ruppert[2]

[1] Department of Computer Science, University of Toronto
[2] Department of Computer Science, Brown University

1 Introduction

What can be computed in a distributed system in which faults can occur? This is a very broad question. There are many different models of distributed systems and many different kinds of faults that can occur. Unlike the situation in sequential models of computation, small changes in the model of a distributed system can radically alter the class of problems that can be solved. Another important goal in the theory of distributed computing is to understand how efficiently a distributed system can compute those things which are computable. There are a variety of resources to consider, including time, contention, and the number and sizes of messages and shared objects.

This paper discusses results that say what *cannot* be computed in certain environments or when insufficient resources are available. A comprehensive survey would require an entire book. As in Nancy Lynch's excellent 1989 paper, "A Hundred Impossibility Proofs for Distributed Computing" [86], we shall restrict ourselves to some of the results we like best or think are most important. Our aim is to give you the flavour of the results and some of the techniques that have been used. We shall also mention some interesting open problems and provide an extensive list of references. The focus will be on results from the past decade.

We begin in Sections 2 and 3 with a brief description of aspects of the models, terminology, and problems that are discussed throughout the paper. The rest of the paper presents a wide variety of lower bound results. Section 4 describes the valency argument, a fundamental technique that has been adapted to prove lower bounds for many different models. One systematic approach to understanding the computational power of different models is to obtain efficient simulations of some models by others. This allows lower bounds derived in one model to be extended to other models. Some simulation techniques will be described in Section 5. Another systematic approach to studying computability for distributed systems is to characterize the models that can solve a particular problem. Some results along these lines will be discussed in Section 6. Alternatively, one can characterize the set of problems solvable in a given model. Results of this type are also described in Section 6, and more fully in Section 7, which covers an important development of the past decade: the use of ideas from topology to prove lower bounds in distributed computing. Section 8 examines the question of whether weak shared object types can become more powerful when they are used in combination with other weak types. A number of techniques that have

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 1–28, 2000.

been used to prove lower bounds on the complexity of solving problems are discussed in Section 9. The last section contains some general remarks about the value of lower bounds.

2 Models

There are a number of excellent descriptions of distributed models of computation, including motivation and formal definitions [16, 80, 87]. Therefore, we shall only briefly mention some aspects of these models which are necessary for the results we present.

There are different ways processes can communicate with one another. In *message-passing* models, processes send messages to one another through communication channels. In *shared-memory* models, processes communicate by performing operations on shared data structures, called *objects*. The `typewriter` font is used to denote object types. Each type describes the set of operations that can be performed on an object, and the responses that it should return if it is accessed by one operation at a time. The most common type of object is the `register`, which stores a value that can be read or written by all processes. Throughout this paper, we assume objects are *linearizable* [64], so operations that may actually run concurrently all appear to happen instantaneously in some order.

Processes may run *synchronously*, so all processes take steps at exactly the same speed, or *asynchronously*, where processes may run at arbitrarily varying speeds. The latter case is generally modelled by thinking of an adversary scheduler that chooses the order in which processes take steps. Algorithms must work correctly regardless of the schedule the adversary chooses.

Many different kinds of faults are considered in distributed systems. Processes may fail and perhaps recover, their states can become corrupted, or they can behave maliciously. Unless otherwise noted, we assume that faulty processes fail by halting permanently. An algorithm that can tolerate up to t failures is called *t-resilient*. A *wait-free* algorithm ensures that every non-faulty process will correctly complete its task even if any number of other processes fail. Thus, it has no infinite executions. Communication channels can fail, lose or delay messages, or deliver them out of order. Shared objects can also be corrupted or fail to respond.

An object type is *deterministic* if its response to each operation is uniquely determined by its previous history. *Non-deterministic* objects may have multiple possibilities. Algorithms must work correctly for all possible responses. Similarly, in *randomized* algorithms, a process may have many choices for its next step, but the choice is made according to some probability distribution. Generally, for randomized algorithms, termination is required only with high probability and one considers expected time, rather than worst-case time.

3 Consensus and Related Problems

The *consensus problem*, first studied by Pease, Shostak and Lamport [81, 93], is perhaps the most thoroughly investigated problem in distributed computing. It is simply stated and it is a primitive building block for many distributed systems.

Consensus is an example of a *decision task*, in which each process gets a private input value from some set and must eventually terminate after having produced an output value. The task specification describes which output values are legal for given input values. For consensus, there are usually two correctness properties that must be satisfied:

Agreement: the output values of all processes are identical, and
Validity: the output value of each process is the input value of some process.

In the *binary consensus* problem, all input values come from the set $\{0, 1\}$. Many variants of the consensus problem have been studied for a variety of different models of distributed computing [41, 86].

There are easy reductions from consensus to other problems, such as *leader election*. In this problem, there are no inputs, exactly one process (called the leader) must output 1, and all other processes must output 0. Once processes have elected a leader, they can solve consensus by using the leader's input value as their common output value. Thus, lower bounds for consensus give lower bounds for leader election and other problems.

Consensus is a good candidate problem for a systematic study of computability, since Herlihy [53] showed that it is universal: a system equipped with `registers` and objects that can solve wait-free consensus can implement any other object type in a wait-free manner.

Objects types can be classified according to the ability of a shared-memory distributed system to solve consensus using objects of that type. Specifically, the *consensus number* $\text{cons}(\mathcal{T})$ of a set of object types $\mathcal{T}$ is the maximum number of processes for which consensus can be solved using objects in $\mathcal{T}$ and `registers` [53, 66]. Then $\text{cons}(\{\mathtt{T}\}) < \text{cons}(\{\mathtt{T}'\})$ implies that $\mathtt{T}'$ cannot be implemented in a wait-free manner from objects of type `T` and `registers`. It follows from this observation and Herlihy's universality result that this classification, called the *consensus hierarchy*, gives a great deal of information about the power of different models of asynchronous, shared-memory systems.

However, the consensus number of an object type does not say everything about the power of a shared-memory model which provides objects of that type and `registers`. For example, there are object types `T` and $\mathtt{T}'$ with consensus numbers 1 and n, respectively, such that 2-set consensus for $2n+1$ processes can be solved using objects of type `T` and `registers`, but not using only objects of type $\mathtt{T}'$ and `registers` [96]. The *k-set consensus problem*, introduced by Chaudhuri [34], is similar to the consensus problem, but relaxes the agreement property. Instead of requiring that all output values be identical, it requires that the set of output values produced has cardinality at most k. Thus, consensus is a special case of k-set consensus, with $k = 1$.

4 Valency Arguments

The valency argument has become the most widely-used technique for impossibility proofs in distributed computing. It was introduced by Fischer, Lynch and Paterson [46] to prove that 1-resilient (and, hence, wait-free) consensus is impossible in an asynchronous message-passing system. Loui and Abu-Amara [84] and Herlihy [53] adapted the valency argument to show impossibility results for several asynchronous shared-memory models.

We shall give an outline of these proofs and then mention some other examples of valency arguments. Valency arguments also play a supporting role in many of the results surveyed in other sections of this paper.

A configuration is a "snapshot" of a distributed system during the execution of an algorithm: it consists of the state of every process and the environment (messages in transit for a message-passing system, or states of all shared objects for a shared-memory system). A configuration of a consensus algorithm is called *univalent* if every possible execution continuing from that configuration gives the same output value, and *multivalent* otherwise. In other words, from a multivalent configuration, there are two or more executions that produce different outputs.

There are three parts to the valency argument that 1-resilient consensus is impossible in an asynchronous system. The first is the observation that any configuration where some process has produced an output is univalent, by definition. Secondly, the validity condition of the consensus problem can be used to show that any consensus algorithm has a multivalent initial configuration. It follows that any consensus algorithm must have a *critical* configuration: a multivalent configuration where a single step by any process will move the system into a univalent configuration. (Otherwise, one could construct an infinite execution containing only multivalent configurations where no process ever produces an output.) The third part of the argument shows that a critical configuration cannot exist. Assuming that such a configuration does exist, one can derive a contradiction using a case argument that considers the possible pairs of steps s_0 and s_1 that could be taken from the critical configuration to lead to univalent configurations with different decision values. For example, if the two steps s_0 and s_1 involve message channels with different destinations or access different shared objects, then performing s_0 followed by s_1 leads to the same configuration as doing the two steps in the reverse order. This contradicts the fact that the two steps lead to different decision values. The other cases show that configurations obtained by taking one or both of these steps are either identical or differ only in the state of one process, which can then be permanently halted by an adversarial scheduler.

Attiya, Dwork, Lynch, and Stockmeyer [12] used valency arguments to give lower bounds on the time required to solve consensus in a semi-synchronous message-passing model, where messages are delivered within time d and there is a bound, r, on the ratio of process speeds. They proved that the worst-case running time of a t-resilient consensus protocol is at least $(r + t - 1)d$. Alur, Attiya and Taubenfeld [5] considered semi-synchronous models where processes communicate using shared `registers`. They proved that each process requires

$\Theta(\frac{U \log U}{\log \log U})$ time, where U is an (unknown) upper bound on the time between steps of a process. To do this, they carefully assign times (consistent with the parameter U) to the steps of any sufficiently long asynchronous execution.

Taubenfeld and Moran [106] used a valency argument to provide a general impossibility result for a large class of problems in the asynchronous shared `register` model, in the case where failures can occur and in the more benign case where faulty processes do not take any steps. (An earlier paper [105] proved similar results for message-passing systems.)

Recently, Moses and Rajsbaum [91] gave a unified framework for proving lower bound results, based on the valency argument, that applies to both message-passing and shared-memory systems, and for synchronous and asynchronous schedulers. Their approach is to restrict the adversary scheduler to a nicely structured subset of the possible executions. (Some earlier work by Lubitch and Moran [85] used a similar approach.) For example, Moses and Rajsbaum considered computation using `single-writer registers`, restricted types of `registers` to which only a single, fixed process can write. They showed that consensus is impossible if processes communicate using these objects, even when they are guaranteed to be scheduled in slightly asynchronous rounds where, in each round, at least $n-1$ processes write to a `register` and then read the values written in that round by at least $n-1$ processes.

This is an instance of an important observation: even though impossibility results and lower bounds with restricted adversaries are stronger (i.e. they imply the same lower bound against a more general adversary), they may be easier to understand and have more elegant proofs (because there are fewer cases to consider). Such proofs also help us identify which aspects of the problem or model make the problem unsolvable. The key to such proofs is coming up with the right adversary. One must discard any unnecessary complications while ensuring that the adversary is still strong enough to prove the impossibility result.

Valency arguments have been generalized in several ways. The definitions of univalence and multivalence have been adapted to fit other models and problems as will be seen in Sections 6, 8 and 9.4. When valency arguments are used for other types of faults, it is necessary to adapt the way in which the scheduler conceals evidence about the first step taken after a critical configuration. It suffices to construct, from any multivalent configuration, two successor configurations which are indistinguishable to some process, yet can lead to executions producing different outputs. This approach is used by Jayanti, Chandra and Toueg [70] to prove the impossibility of certain implementations in a model where objects, instead of processes, fail. Related work by Afek, Greenberg, Merritt and Taubenfeld [2] shows that consensus is impossible when processes communicate with any types of objects, if half the processes can fail and the states of half the objects can become corrupted.

Lo [82] used a valency argument, adapted to deal with non-deterministic objects, to prove that there is an object type with consensus number 1 which can be used together with `registers` to solve 1-resilient consensus for n processes. In contrast, for $t > 1$, Chandra, Hadzilacos, Jayanti, and Toueg [32] proved that

an object type has consensus number greater than t if and only if it can be used together with `registers` to solve 1-resilient consensus for $n > t$ processes.

Dwork, Herlihy and Waarts [42] use a valency argument to obtain a lower bound on the contention of any wait-free consensus algorithm that uses shared memory.

Valency arguments have also been used to study quantum-based schedulers, where the scheduler guarantees that processes will run for a certain length of time (called a quantum) without being pre-empted by other processes running on the same processor. They give lower bounds on the size of quantum necessary for n-process consensus to be universal in a system of n processors executing any number of processes [6].

5 Simulations

Simulations provide a way of extending lower bounds to different settings. For example, suppose that one system A can simulate another system A'. Then, if a problem has been proved impossible in A, it follow that the problem is impossible in A' too. Similarly, lower bounds in A imply lower bounds in A', although the bounds obtained for A' may be smaller, depending on the efficiency of the simulation. In this section, we survey some of the simulations that have been useful for establishing such results.

The Borowsky-Gafni (BG) simulation [21, 24] describes how a system of n processes can simulate an algorithm designed for a system with m processes. They consider asynchronous systems where processes communicate using `registers` and up to t processes may fail. This important technique has been used and extended by others [36, 62, 96].

In the following brief description, we call the simulated processes *threads* and the simulating processes *processors*, to improve clarity. A key element of the simulation is the safe agreement subroutine. It satisfies the agreement and validity properties of the consensus problem, but might not terminate. However, if processors are running several copies of the safe agreement routine in parallel, a processor failure can prevent at most one of the copies from terminating. Although the BG simulation technique is applicable more generally, we consider how to simulate an algorithm with m threads for the k-set consensus problem, defined in Section 3. Every processor simulates the steps of every thread. This is done in parallel for all processors and threads. The processors use safe agreement to ensure that a simulated step has the same result in the simulations carried out by different processors. Each processor visits the threads in round-robin order and tries to execute the next step of each thread it visits. Whenever a processor observes that a thread has terminated, the processor can also terminate, using the output of the thread as its own output. Since the threads will output at most k different values, it follows that this simulation provides a correct solution to the k-set consensus problem. The safe agreement routine is designed so that a processor failure will block the simulation of at most one thread. This ensures that, if the original m-process algorithm was t-resilient, then the n-process al-

gorithm constructed will also be t-resilient. As described in Section 7.2, there is no wait-free (i.e. k-resilient) k-set consensus algorithm for $k+1$-processes. Thus, the BG simulation implies that there is no k-resilient k-set agreement algorithm for $m > k$ processes.

Jayanti, Chandra, and Toueg [70] used a different simulation to prove that there is no algorithm to solve consensus for two processes in an asynchronous shared-memory system in which at most one object can fail by delaying its responses forever. To do this, they showed how $m+2$ processors that communicate using non-faulty `registers` could simulate a consensus algorithm for two threads that uses m objects and tolerates one object failure. The actions of each thread and object are simulated by a different processor. The resulting algorithm would solve 1-resilient consensus using only `registers`, which is impossible [84].

Afek and Stupp [4] obtained complexity results from computability results by using a simulation. They considered the problem of electing a leader in a system of n processors using `registers` and one `compare&swap` object that can store one of v different values, and proved that some process must take $\Omega(\log_v n)$ steps. Given such a leader election algorithm in which no process ever takes more than d steps, they showed how $\lfloor n/(d+1) \rfloor$ processes can simulate it, using only `registers`, and thereby solve $(v-1)^d$-set consensus. In the simulation, different processes may actually simulate different executions of the leader election algorithm. However, the number of different simulated executions is at most $(v-1)^d$. The lower bound on d follows from the fact that `registers` alone cannot solve set consensus when $(v-1)^d < \lfloor n/(d+1) \rfloor$ (see Section 7.2).

6 Deciding When Problems are Solvable

Because there are so many different models of distributed systems, it would be useful to have a general technique to determine whether a given model can solve a given problem, or implement a given data structure. Unfortunately, this is not possible in general.

6.1 Undecidability

Jayanti and Toueg [73] proved that there is no algorithm that, given the description of a type and an initial state, determines if it can be implemented from `registers`. They use a reduction from the halting problem. Given a (deterministic) Turing machine M, they construct a type $\mathtt{T}(M)$ whose state stores a configuration of M and a boolean flag. Suppose the object is initially in a state corresponding to M's initial configuration on a blank input tape and the boolean flag is set to false. The type $\mathtt{T}(M)$ is equipped with a single operation. The operation updates the configuration stored in the state by simulating one step of M and returns 0 as long as M has not halted. The first operation applied to $\mathtt{T}(M)$ after the simulated machine M has halted sets the flag to true and returns 1. Any operation on $\mathtt{T}(M)$ after the flag is set returns 2.

If M halts on a blank tape, then $\mathtt{T}(M)$ can be used to solve leader election (and hence consensus) for two processes: each process repeatedly accesses the object until it returns a non-zero value and the process that receives the value 1 becomes the leader. This means `registers` cannot implement $\mathtt{T}(M)$. However, if M never halts on a blank input tape, then $\mathtt{T}(M)$ can be implemented using a `register` initialized to 0 and having each operation applied to $\mathtt{T}(M)$ replaced by a read of this `register`. It follows that one cannot decide whether the type $\mathtt{T}(M)$ can be implemented from `registers`. A similar construction can be used to show that the consensus number of a given type is undecidable. Further undecidability results are described in Section 7.3.

6.2 Decidability of Consensus Numbers

For some natural classes of types, decision procedures for consensus number do exist. They follow from theorems that characterize types in the class in terms of their consensus number.

One such class consists of the read-modify-write (RMW) object types [76]. A RMW operation updates the state of the object by applying some function, and returns the old value of the state. For example, the *test&set* operation is a RMW operation that applies the function $f(x) = 1$, and *fetch&add* applies the function $f(x) = x + 1$. Other RMW operations include *read* and *compare&swap*. A RMW type is one where all permitted operations have this form. Ruppert [98] gave a characterization of the RMW types that can solve consensus among n processes. The characterization uses a restricted form of the consensus problem, called team consensus, where processes are divided into two teams and all processes on the same team receive the same input. A RMW type T has consensus number at least n if and only if there is an algorithm for solving team consensus among n processes in which every process performs exactly one step on an object of type T. A valency argument was used to show the necessity of this condition: by examining the behaviour of processes as they each take their first step after the critical configuration of a consensus algorithm, one can obtain the required one-step algorithm for team consensus. For finite types, this condition is decidable.

A similar characterization was also given for readable types [98], which allow processes to read the state of the object without changing it. Together, these two classes of objects contain many of the common shared-memory primitives. These characterizations were used to prove impossibility results for consensus in the multi-object model [99] (where processes can access more than one shared object in a single atomic action), following the work of Afek, Merritt and Taubenfeld [3].

Recently, Herlihy and Ruppert [62] gave a characterization of one-shot types that can solve wait-free consensus among n processes. (A *one-shot* type is one that can only be accessed once by each process.) This characterization, too, is decidable for finite types.

A natural open question is to obtain an algorithm that decides the consensus number of any type with finite state set. An interesting special case would be to consider non-deterministic RMW and readable types. One might also be able to

gain a better understanding of the relative power of different types by studying their ability to solve other problems such as set consensus and consensus-with-reset.

6.3 Characterizing Solvable Tasks

A related question is to determine whether a given problem (from some class) is solvable in a particular fixed model. Here, the approach has been to find characterizations of the solvable problems.

In the asynchronous message-passing model, Biran, Moran and Zaks [18], building on earlier work by Moran and Wolfstahl [90], gave a combinatorial characterization of the decision tasks that can be solved 1-resiliently in an asynchronous message-passing system. This characterization, described below, is in terms of the task's input and output vectors, with one coordinate corresponding to each process. Suppose there is a 1-resilient message-passing algorithm to solve a given task for n processes. Let $G(\boldsymbol{x})$ denote the set consisting of all output vectors produced by the algorithm with input vector $\boldsymbol{x}$. First, for each input vector $\boldsymbol{x}$, they consider the *similarity graph* with vertex set $G(\boldsymbol{x})$ and edges between any two vectors that differ in exactly one coordinate. They prove this similarity graph is connected using a valency argument with slightly different definitions: a configuration C is univalent if all executions from C lead to an output vector in the same connected component and multivalent otherwise. Secondly, they show that, if I is any set of input vectors that differ only in coordinate j, then there is a set of output vectors, one from $G(\boldsymbol{x})$ for each $\boldsymbol{x} \in I$, that differ only in coordinate j. This follows from consideration of those executions in which process P_j is non-faulty, but takes no steps until all other processes have produced an output.

Conversely, suppose there is a task for n processes such that, there is a set $G(\boldsymbol{x})$ of allowable output vectors for each input vector $\boldsymbol{x}$, which has the following two properties: the similarity graph with vertex set $G(\boldsymbol{x})$ is connected, and if I is a set of input vectors that differ only in coordinate j, then there is a set of output vectors, one from $G(\boldsymbol{x})$, for each $x \in I$, that differ only in coordinate j. Then Biran, Moran and Zaks proved that there is a 1-resilient message-passing algorithm to solve the task. In later papers, they also showed that determining whether a task has these properties is NP-hard for more than two processes [19], and gave very precise bounds on the round complexity of solving any task that satisfies them [20].

Attiya, Gorbach and Moran [13] gave a simple characterization of the tasks that are solvable in systems where asynchronous processes have no names, run identical programmes, do not know how many processes are in the system, and communicate using `registers`. The characterization (and the proof of its necessity) is similar in flavour to the results by Biran, Moran and Zaks, described above. Other impossibility results for systems with anonymous processes appear in [39, 72].

Chor and Nelson [38] studied *interactive* tasks, where each process receives a sequence of input values and must produce the output value corresponding to its

current input value before being given its next input value. They characterized the interactive tasks that can be solved in an asynchronous system if a consensus subroutine is available. Their conditions ensure, among other things, that the set of allowable output vectors does not depend on the value of input values which have not yet been received. Note that the specification of interactive tasks do not necessarily ensure linearizability, so Herlihy's universality result [53] does not apply.

7 Topology

Perhaps the most interesting development in the theory of distributed computing during the past decade has been the use of topological ideas to prove results about computability in fault-tolerant distributed systems. Other connections between topology and distributed computing have been discussed in the literature (see [50, 51, 74, 97]), but the results described in this section represent new and powerful uses of topology in distributed computing, particularly for proving lower bounds.

7.1 Simplicial Complexes

We begin with some brief definitions of ideas from the topology of simplicial complexes. Several papers contain good introductions to the connections between distributed computing and simplicial complexes [47, 57, 60].

A d-dimensional *simplex* (or d-simplex) is a set of $d+1$ independent *vertices*. Geometrically, the vertices can be thought of as (affinely) independent points in Euclidean space. A 0-simplex is a single point, a 1-simplex is represented by a line segment, a 2-simplex is represented by a filled-in triangle, and so on. A (simplicial) *complex* is a finite set of simplexes closed under inclusion and intersection. The *dimension* of a complex is the maximum dimension of any simplex that appears in it. Examples of simplicial complexes appear in Figure 1.

A vertex can be used to represent the internal state (or part of the internal state) of a single process. A d-simplex whose vertices correspond to different processes represents compatible states of $d+1$ processes. As an example, consider the binary consensus problem for three processes, P, Q and R. The possible starting configurations of an algorithm for this problem are shown in Figure 1(a). Each vertex is labelled by a process and the binary input value for that process. The complex consists of eight 2-simplexes arranged to form a hollow octahedron. Each 2-simplex represents one of the eight possible sets of inputs to the three processes. The corresponding output complex in Figure 1(b) shows the possible outputs for the binary consensus problem. In the upper 2-simplex, all processes output value 0, while in the lower 2-simplex, all processes output value 1. Not all output simplexes are legal for every input simplex: by the validity condition of consensus, if all processes start with the input value 0, then only the upper output simplex is legal.

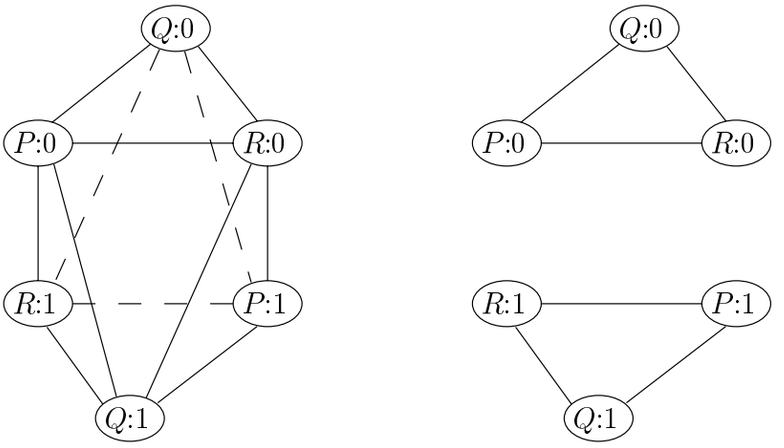


Fig. 1. (a) Input complex and (b) output complex for three-process binary consensus

More generally, any decision task for n processes can be modelled in a similar way. The input complex I contains one $(n-1)$-simplex for each possible input vector. The output complex O contains one simplex for each possible output vector. A map Δ that takes each simplex S of I to a set of simplexes in O (labelled by the same processes) defines which output vectors are legal for each input vector.

Simplicial complexes are used as a means of describing whether processes can distinguish different configurations from one another. In that sense, they are similar to, though more general than, the similarity graphs of Biran, Moran and Zaks [18] discussed in Section 6.3. Nodes in those graphs correspond to $(n-1)$-simplexes. The situation where two output vectors differ in only one coordinate, which is modelled by an edge in a similarity graph, is represented in the complex by having the two simplexes share $n-1$ common vertices. Complexes can capture more information about the degree to which two configurations are similar: two simplexes that have d common vertices are similar to exactly d processes. This fact is useful in studying t-resilient algorithms in general, whereas similarity graphs are useful only for the case $t = 1$.

Consider a wait-free protocol for n processes that solves some task. One can define a corresponding $(n-1)$-dimensional *protocol complex*. Each vertex is labelled by a process and the state of that process when it terminates. Given any input vector and any schedule for the processes (as well as a description of the results of any coin tosses or non-deterministic choices), the final state of every process is determined. This final configuration is represented by a simplex in the protocol complex.

Each process must decide on an output value for its task based solely on its internal state information at the end of the protocol. This defines a decision

map δ that takes each vertex of the protocol complex to a vertex of the output complex (labelled by the same process). Let S be a simplex of the protocol complex. Since S represents a configuration of compatible final states for some set of processes, $\delta(S)$ must be a simplex of the output complex, representing a compatible set of outputs for those processes. Furthermore, δ must "respect" the task specification: If S represents a configuration reached by some execution whose inputs come from the simplex I of the input complex, then $\delta(S)$ must be in $\Delta(I)$.

The basic method of proving lower bounds using the topological approach can now be summarized. One uses information about the model to prove that any protocol complex has some topological property which is preserved by the map δ. The specification of the task is used to show that the image of δ cannot have the property, implying that no such map δ can exist.

For example, it can be shown that, in the asynchronous model where processes use `registers` to communicate, any protocol complex (that begins from a connected input complex) is connected [63]. The connectivity property is preserved by any map δ, since δ maps simplexes to simplexes. As shown in Figure 1, the input complex for three-process binary consensus is connected. The image of δ must include vertices in both triangles of the output complex, since the task specification requires that, for any run where all processes get the same input value v, all processes output v. Thus the image of δ is disconnected, and hence three-process binary consensus is impossible in this model.

7.2 Set Consensus Results

Much of the inspiration for the early topological impossibility results came from Chaudhuri [34], who defined the k-set consensus problem. She observed that Sperner's Lemma [104], a tool often used in topology, could be applied to study the task. In papers that first appeared at STOC in 1993, three different groups of researchers [21, 63, 100] used Sperner's Lemma to prove that $k+1$ processes cannot solve wait-free k-set consensus in an asynchronous model using shared `registers`. In addition to proving the result about set consensus, each of the three developed interesting techniques that led to proofs of different or more general results. This is a great example of the important role that lower bounds for a well-chosen problem can play in opening up new areas of research. Similar tools have also been used to provide lower bounds for the set agreement problem in a synchronous message-passing model [35]. Attiya reproved the impossibility of set consensus using more elementary tools [11].

Borowsky and Gafni's impossibility proof [21] uses the protocol complex for $k+1$ processes, in the case where each process uses its process name as its input to the set consensus problem. They introduce the *immediate snapshot model*, which puts restrictions on the adversarial scheduler and show that this model can be simulated in the standard asynchronous model. Additionally, protocols are assumed, without loss of generality, to be *full-information protocols*: each process repeatedly takes a snapshot of shared memory, appending the result to its local state, and then writes its local state to shared memory. With these

simplifications of the model, Borowsky and Gafni show that protocol complexes have a very regular form. This allows them to apply a variant of Sperner's Lemma to show that, for some simplex of any protocol complex, each of the $k+1$ processes outputs a different value. Using the BG simulation technique, described in Section 5, they extend the impossibility result from the wait-free setting to one where the number of failures is bounded by t.

The impossibility proof by Saks and Zaharoglou [100], which uses point-set topology, has a different flavour from the other results described in this section. They use a simplified model similar to that of Borowsky and Gafni, and consider the space of all (finite and infinite) schedules. Saks and Zaharoglou define a topology on this set, where open sets are sets of schedules that can be recognized (that is, if there is an algorithm where some process eventually writes "accept" for exactly those runs that follow a schedule in the set). Now, suppose a k-set consensus algorithm exists for $k+1$ processes (where each process has its name as its input). Then the set D_i of schedules in which some process outputs the value i is an open set, and it does not contain any schedule where i does not take any steps. These facts can be used, together with Sperner's Lemma, to show that there is one schedule contained in $\cap_i D_i$. In this schedule, the processes output $k+1$ different values, which contradicts the correctness of the algorithm. An interesting direction for future research is to investigate the structure of this topological space of schedules. Perhaps theorems from point-set topology could then be applied to prove other results in distributed computing.

7.3 The Asynchronous Computability Theorem

The third paper that proved impossibility of k-set consensus has since been developed into a more general result that characterizes the tasks that can be solved in a wait-free manner using `registers`. Herlihy and Shavit [63] proved that a task is solvable if and only if it is possible to subdivide the simplexes of the input complex into smaller simplexes (with any newly created vertices being appropriately labelled by processes) that can then be mapped to the output complex. This mapping μ must satisfy properties similar to those of a decision map δ. It must preserve the process labels on the vertices, map simplexes to simplexes, and it must respect the task specification: if a simplex I of the input complex is subdivided into smaller simplexes, the smaller ones must all be mapped to simplexes in $\Delta(I)$. This characterization is called the Asynchronous Computability Theorem. It reduces the question of whether a task is solvable to a question about properties of the complexes defined by the task specification. A key step in proving the necessity of the condition is a valency argument that shows the protocol complexes in this model contain no holes. To prove the impossibility of set consensus, they show (using Sperner's Lemma) that no mapping μ can exist for the set consensus task. The paper also gives results on the impossibility of renaming, a problem where all processes must choose distinct values from a small set.

Gafni and Koutsoupias [49] used the Asynchronous Computability Theorem to show that it is undecidable whether a given task has a wait-free solution using

`registers`, even for finite tasks with three processes. They use a reduction from a problem known to be undecidable: loop contractibility, where one must decide whether or not a loop on a 2-dimensional simplicial complex can be contracted, like an elastic band, to a point while staying on the surface of the complex. Suppose the input complex (for three processes) is simply a 2-simplex. Given a loop on a 2-dimensional output complex, they define a task that requires the boundary of the input simplex to be mapped to the loop by the function μ of the Asynchronous Computability Theorem. This map μ can be extended to the whole (subdivided) input simplex if and only if the loop can be contracted. Herlihy and Rajsbaum [58] extended this undecidability result to other models.

Havlicek [52] used the Asynchronous Computability Theorem to identify a condition that is necessary for a task to have a wait-free solution using `registers`. This condition is computable for finite tasks.

Several researchers have presented characterizations similar to the Asynchronous Computability Theory. These alternative views give further insight into the model, and the proof techniques are quite different in some cases. Herlihy and Rajsbaum [56] showed how to prove impossibility results in distributed computing using powerful ideas developed in the homology theory of simplicial complexes. They discussed models where the shared memory consists of `registers`, or of `registers` and `set consensus` objects. They reproved impossibility results for the set consensus problem, and gave some new results for the renaming problem. Attiya and Rajsbaum [15] used purely combinatorial arguments to develop a characterization of tasks solvable using `registers`, similar to the Asynchronous Computability Theorem. In particular, they showed that the protocol complexes for a simplified model have a very regular form. Both of these papers eliminated the need to subdivide the input complex by introducing maps that take simplexes of the input complex directly to subcomplexes of the output complex.

Borowsky and Gafni [23] gave an elegant proof of a version of the Asynchronous Computability Theorem without using topological arguments. They introduced the iterated immediate snapshot model and prove that it is capable of solving the same set of tasks as the ordinary `register` model. They prove the equivalence of the models by giving algorithms to simulate one model in the other [22, 23]. The protocol complex of a (full-information) protocol in their simplified model is a well-understood subdivision of the input complex. Thus, a problem is solvable in either model if and only if there is a decision map from a subdivision of this form to the output complex that respects the task specification.

7.4 Other models

Herlihy and Rajsbaum [55] undertook a detailed investigation of the topology of set consensus. They gave conditions about the connectivity of protocol complexes that are necessary for the solution of set consensus. They also described connectivity properties of the protocol complexes in a model where the primitive objects are `set consensus` objects and `registers`. They later used this work to give a computable characterization of tasks that can be solved t-resiliently

in various models that allow processes to access `consensus` or `set consensus` objects [58]. The characterization uses topological tools but also builds on the characterization given by Biran, Moran and Zaks (see Section 6.3) for systems using only `registers`.

Herlihy and Rajsbaum [59] also considered an interesting class of decision tasks, called loop agreement tasks [58]. Using topological properties of the output complexes, they describe when one loop agreement task can be solved using `registers` and a single copy of an object that solves another loop agreement task.

Herlihy, Rajsbaum and Tuttle [61] used the topological approach to give unified proofs of lower bounds for set consensus in several message-passing models with varying degrees of synchrony. There has been other work presenting impossibility results for several different models in a unified way [48, 85, 91].

7.5 Directions for future research

A desirable goal is a better understanding of the structure of protocol complexes for different models. The complexes tend to be quite complicated, but to obtain impossibility results, it is often sufficient to prove that they have certain properties, without fully describing their form. Restrictions on the adversarial scheduler can also simplify the structure of protocol complexes, making them easier to study while simultaneously strengthening any lower bounds obtained. Most of the research has focused on one-shot objects or tasks; extensions of these techniques to long-lived objects is a subject of current research.

8 Robustness

The consensus number of an object type provides information about the power of a system that has objects of that type and `registers`. However, the classification of individual types into the consensus hierarchy does not necessarily provide complete information about the power of a system that contains several different types of objects: it is possible that a collection of weak types can become strong when used in combination. This issue was first addressed by Jayanti [66, 68] in his papers on *robustness* of the consensus hierarchy. The hierarchy is robust (with respect to a class of object types) if it is impossible to obtain an n-process implementation of a type at level n of the hierarchy from a finite set of types that are each at lower levels. Robustness is a desirable property since it allows one to study the synchronization power of a system equipped with several types by reasoning about the power of each of the types individually.

8.1 Non-Robustness Results

A variety of non-robustness results have been proved during the past decade. Typically, one defines a pair of objects that are tailor-made to work together to solve consensus easily. To complete the proof, one must show that each of

the types, when used by themselves, cannot solve consensus. These impossibility results often required ingenious lower bound techniques.

Some of the early results on the robustness question showed that the hierarchy is not robust under slightly different definitions of consensus number [40, 66, 68, 75]. These results are, in part, responsible for the choice of the now-standard definition of consensus number. There have also been a number of non-robustness results when the response that objects return can depend on the identity of the process that invoked the operation [31, 32, 88, 94].

One of Jayanti's proofs [66] used an interesting simulation technique. He defined a simple type, called `weak-sticky`, with infinite consensus number. He gave an implementation of `weak-sticky` from `registers` that is not wait-free but has the property that at most one operation on the object will fail to terminate. He used this to show that there is no consensus algorithm for three processes that uses a single `weak-sticky` object and `registers`; if there were, one could use the implementation to obtain a 1-resilient consensus algorithm for three processes using only `registers`, which is impossible [84]. The use of "imperfect" implementations to prove results has been used elsewhere (see Section 5).

Schenk [102, 103] proved that the consensus hierarchy is not robust by considering a type with unbounded non-determinism, i.e. an operation may cause an object to choose non-deterministically from an infinite number of possible state transitions. In this case, he said an algorithm is wait-free if the number of steps taken by a process must be bounded, where the bound may depend on the input to the protocol. For objects with bounded non-determinism, this definition of wait-freedom is equivalent to the requirement that every execution is finite. Lo and Hadzilacos [83] improved Schenk's result by showing that the hierarchy is not robust even when restricted to objects with bounded non-determinism.

Schenk defined two types, called `lock` and `key`. The `key` object is a simple non-deterministic object that can easily be used to solve the *weak agreement* problem: All processes must agree on a common output value and, if all processes have the same input value, the output value must differ from it. He used a counting argument to show that, for any consensus algorithm using `keys` and `registers`, there exists a fixed output value for each `key` which is consistent with every execution. This allows all the `key` objects to be eliminated, which is impossible, unless $\text{cons}(\{\texttt{key}\}) = 1$.

The `lock` object was specially constructed to provide processes with a solution to the consensus problem if and only if processes can "convince" the object that they can solve weak agreement. The `lock` object non-deterministically chooses an instance of the weak agreement problem and gives this instance to the processes as a challenge. It then reveals the solution to the original consensus problem if and only if processes provide the `lock` object with a correct solution to the challenge. (The idea of defining an object that only provides useful results to operations when it is accessed properly, in combination with another type of object, was originated by Jayanti [68] and is common to many of the non-robustness proofs.) If processes have access to both a `lock` and `key` object, they can use the `key` to solve the `lock`'s challenge and unlock the solution to

consensus. Schenk used a type of valency argument developed by Lo [82], to show that weak agreement and, hence, consensus for two processes cannot be solved using only `locks` and `registers`.

8.2 Robustness Results

Although the consensus hierarchy is not robust in general, the practical importance of the non-robustness results is unclear, since the objects used in the proofs are rather unusual. The hierarchy has been shown to be robust for some classes of objects that include many of the objects commonly considered in the literature.

Ruppert [98] showed the the hierarchy is robust for the class of all RMW and readable types. The proof uses the characterization of the types that can be used to solve consensus for n processes, described above in Section 6.2. It is easy to show, using a valency argument, that any consensus algorithm for n processes built from such objects must include an object whose type satisfies the conditions of the characterization. Therefore, n-process consensus can be solved using only that type and `registers`.

Recently, Herlihy and Ruppert [62] used the topological approach to characterize the one-shot types that can be combined with other types to solve consensus. It follows from their characterization that the class of deterministic one-shot types is robust. The key tool in one direction of the proof is a simulation technique that builds on the BG simulation (see Section 5). The other direction is a generalization of the non-robustness results [83, 103] described above.

8.3 Directions for Future Research

In proving the robustness result for readable and RMW types, two important properties are used: such objects are deterministic and their state information can be accessed in some simple way by each process. The robustness result for one-shot types uses a similar property: when accessing a one-shot object, a process gets all of the state information that it will ever be able to obtain directly from the object by doing a single operation. Can robustness results be extended to other natural classes of types that do not have these kinds of properties? By finding the line that separates those types that can be combined with others to violate robustness from those that cannot, we gain insight into the way that types behave when used in complex systems. Work on the robustness question has produced a number of interesting proof techniques, and has required very careful definitions to avoid using "obviously true" properties which are not. Clarifying definitions is one of the important contributions of lower bounds.

9 Complexity Lower Bounds

Once we know that a particular problem is solvable in a certain distributed system, we would like to have algorithms that solve these problems as efficiently

as possible, for example, using shared objects that are as small as possible, using as few shared objects as possible, and using as little time as possible. In this section, we discuss lower bounds on the resources needed to solve various problems. In some cases, these bounds show that certain algorithms are optimal or close to optimal. They also help us to understand the inherent difficulty of these problems.

9.1 Lower Bounds on Space

Burns, Jackson, Lynch, Fischer, and Peterson [26] considered deterministic solutions to the mutual exclusion problem using one shared object. In this problem, processes repeatedly compete for exclusive access to a critical section, where they are allowed to use a shared resource. A counting argument was used to show that, if the object has an insufficient number of states, there are two configurations which will appear identical to a group of processes: one in which no processes are in the critical section and one in which some other process is in the critical section. If only the processes in this group are scheduled, they will behave the same way starting from both of these configurations, resulting in an incorrect execution in one of the two cases (with either no process ever entering the critical section or more than one process in the critical section at the same time). For randomized computation, Kushilevitz, Mansour, Rabin, and Zuckerman [77] obtained lower bounds on the size of the shared object based on an analysis of Markov chains.

Burns and Lynch [27, 87] introduced the following technique to prove that any mutual exclusion algorithm for $n \geq 2$ processes that communicate using `registers` uses at least n `registers`, no matter how large the `registers` are. If an algorithm uses an insufficient number of objects, one can construct an execution that exhibits incorrect behaviour by combining pairs of different executions that look the same to a group of processes. An adversary scheduler runs the algorithm until there are processes covering every object that the algorithm uses. (A process *covers* an object if it will write to it when next allocated a step by the scheduler.) The effects of subsequent steps by other processes can be hidden by later performing these writes. Their lower bound is optimal, matching the number of `registers` used by known mutual exclusion algorithms [78, 79].

Moran, Taubenfeld and Yadin [89] used the same approach to prove that any wait-free implementation of a `mod m counter` for n processes from objects with only 2 states must use at least $\min(\frac{n+1}{2}, \frac{m+1}{2})$ such objects.

Fich, Herlihy, and Shavit [45] considered a very weak termination condition, *non-deterministic solo termination*: at any point, if all but one process fails, there is an execution in which the remaining process terminates. In particular, wait-free and randomized wait-free algorithms satisfy non-deterministic solo termination. They proved that $\Omega(\sqrt{n})$ `registers` are needed by any asynchronous algorithm for n-process consensus that satisfies this property. The proof uses the covering technique together with a valency argument, showing that from any multivalent configuration, there is another multivalent configuration in which more `registers` are covered. Because consensus is a decision task, the proof is

more difficult than for mutual exclusion, where processes can repeatedly request exclusive access to the critical section, or for implementing a `counter` that processes can repeatedly increment. They overcome this problem by a new method of cutting and combining executions. Although there are algorithms for randomized wait-free consensus among n processes that use $O(n)$ `registers` of bounded size [9], it remains open whether this is optimal.

Fich, Herlihy, and Shavit extended their result to algorithms using *historyless objects*. An object is historyless if its state depends only on the last non-trivial operation that was applied to it. Some examples of historyless types are `register`, `swap`, and `test&set`. Using this extension, they showed that $\Omega(\sqrt{n})$ historyless objects are necessary for randomized wait-free implementations of objects such as `compare&swap`, `fetch&increment`, and bounded `counters`, in an n-process system. Jayanti, Tan and Toueg [71] improved these bounds, showing that $n-1$ historyless or `resettable consensus` objects are necessary for randomized wait-free implementations of these objects. Much work remains to be done to obtain space complexity lower bounds for other problems and in models with more powerful objects. Attiya, Gorbach and Moran [13] used similar techniques in a fault-free model, where processes have no identifiers, run identical programmes, and communicate via `registers`. They showed that $\Omega(\log n)$ shared `registers` and $\Omega(\log n)$ rounds are required for n processes to solve consensus.

9.2 The Complexity of Universal Constructions

Herlihy's universality result, discussed in Section 3, and subsequent similar papers [1, 33, 54, 73], provide *universal constructions*, which automatically give a distributed implementation of any object type, using sufficiently powerful shared-memory primitives. Jayanti [67, 69] has studied some of the limitations of this approach to providing implementations. He showed that a process that performs a wait-free simulation of an operation using a universal construction requires $\Omega(n)$ steps of local computation in the worst case, where n is the number of processes [67]. This bound does not depend on the nature of the communication between processes, and even holds in an amortized setting. The key idea in the proof is the design of an object type that conspires with the scheduler to reveal as little information about the behaviour of the object as possible. This ensures that each process, simulating a single operation *op*, must do some computation for each simulated operation that precedes *op*. The bound is tight [53].

Jayanti [69] also proved a lower bound of $\Omega(\log n)$ on the number of shared-memory operations that must be performed by a universal construction in the worst case. This bound applies to a shared-memory model that has quite powerful primitive types of shared objects and holds (for expected complexity) even if randomization is permitted. Jayanti proved the bound by considering the wakeup problem, where some process must detect when all processes have begun taking steps, and studying how information propagates through the system. Roughly speaking, each shared-memory operation at most doubles the size of the the set of processes that are known (by some process or memory location) to have woken up. This lower bound is also tight [1].

9.3 Lower Bounds on Time

To understand the relative power of different models, it is important to obtain separation results by proving a lower bound for a problem in one model that is larger than its complexity in another model.

One problem that has been used to obtain a separation result is approximate agreement, a variant of agreement that can be solved in asynchronous systems. In this problem, each process receives a real input value in the range [0,1]. Their output values must lie within the range of the input values and must differ from one another by no more than some given parameter $\epsilon > 0$. Attiya, Lynch, and Shavit [14] proved that any wait-free approximate agreement algorithm for n processes and $\epsilon = 1/2$, using `single-writer registers` (of unbounded size), has a failure-free execution in which no process decides the value of its output before round n. They do this by obtaining an upper bound on the number of processes that can influence the state of a particular process during the first t rounds of a round-robin execution. Then they show that if a process P is not influenced by another process P', it cannot decide; otherwise, there is another execution which is indistinguishable to P in which P' runs to completion before the other processes begin and outputs an incompatible value.

Schenk [101] proved that any wait-free approximate agreement algorithm for n processes that uses b-bit `registers` must take $\Omega(\log(1/\epsilon)/b)$ rounds and use $\Omega(\log(1/\epsilon)/b)$ `registers`. The proof considers the amount of information a process needs to determine its output value after all the other processes have decided.

Schenk also gave an algorithm using 1-bit `registers` that matches these lower bounds. Together with the `single-writer register` lower bound, this implies that any wait-free implementation of `registers` from `single-writer registers` has round complexity $\Omega(\log n)$. However, there is a large gap between this lower bound and the best known implementation. It also remains open whether approximate agreement can be solved faster using larger `registers`.

Hoest and Shavit [65] used topological techniques to determine the time complexity of approximate agreement in a generalization of Borowsky and Gafni's iterated immediate snapshot model. Essentially, they related the time complexity of the task to the degree to which the input complex must be subdivided before one can map it to the output complex (see Section 7.3). Although, in terms of computability, their model is equivalent to the standard asynchronous model containing only `registers`, their complexity results do not carry over. Much work remains to find additional ways of applying topology to prove complexity lower bounds.

Sometimes, the choice of problem to use for a separation result comes from identifying the essential part of a simulation. One such example is the *write-all* problem: given n `registers`, all initially 0, set them all to 1. It has been used as the basis of simulations of synchronous algorithms on asynchronous shared-memory models. Buss, Kanellakis, Ragde, and Shvartsman [28] proved that any asynchronous algorithm for this problem that uses n processes, at most half of which can fail, must perform $\Omega(n \log n)$ writes to the bits of the array. Their

proof holds in a very strong model, where processes can flip coins and, in a single step, read the entire contents of shared memory. The idea is that an adversary schedules the processes to run until each covers a bit of the array. Among the bits with value 0, the adversary chooses the half which have the fewest processes covering them and schedules the $n/2$ or more processes which cover other bits to perform their writes. This can be repeated $\log_2 n$ times, each time reducing the number of 0 bits in the array by at most a factor of 2. They provide a matching upper bound when processes can perform atomic snapshots. For any $\epsilon > 0$, there is a deterministic algorithm using only `registers` that performs $O(n^{1+\epsilon})$ operations, in total [7, 92]. It is an open question whether there is an algorithm for the write-all problem that uses only `registers` and performs $n(\log n)^{O(1)}$ total operations.

9.4 Lower Bounds on Time for Randomized Computation

Adding randomness to a model can increase its computational power. This can make proving lower bounds in randomized models more difficult. Randomized consensus is a variant of consensus with a slightly weaker termination condition: all non-faulty processes must terminate within a finite expected number of steps. In contrast to consensus, randomized consensus can be solved in an asynchronous distributed message-passing system or in an asynchronous read-write shared-memory system (i.e. using only `registers`). For example, there are wait-free shared-memory algorithms for randomized consensus among n processes where the expected total number of operations performed is $O(n^2 \log n)$ [25] and where the expected number of operations performed by each process is $O(n \log^2 n)$ [10]. Most algorithms for randomized consensus are based on collective coin flipping, which is a way of combining many local coin flips into a single global coin flip. However, there is a complication: a malicious adversary can destroy some of the local coins after they are tossed but before they are used. The goal of a collective coin flip algorithm is to limit the degree to which the adversary can influence the outcome of the global coin flip.

Aspnes proved that any t-resilient algorithm for randomized consensus on an asynchronous message-passing or read-write shared-memory system performs $\Omega(t^2/\log^2 t)$ local coin flips (and, hence, work) with high probability [8]. This result, in fact, applies to all models that can be deterministically simulated by read-write shared memory, including models that have `counters` or constant time atomic snapshot primitives. The proof of his lower bound has two parts. One is a lower bound on the number of local coin flips needed to prevent an adversary from having too much influence on the outcome of a collective coin flip. The other is an extension of the valency argument to the randomized setting to show that an algorithm either performs a collective coin flip with small bias or spends lots of local coin flips to avoid doing so. Aspnes introduces the notion of an *a-univalent configuration*, a configuration from which an adversary scheduler can cause the algorithm to produce the output value a, with sufficiently high probability. Then a *bivalent configuration* is both 0-univalent and 1-univalent and a *nullvalent configuration* is neither. He shows that, with high probability,

an adversary scheduler can force any algorithm into a bivalent or nullvalent configuration from its initial configuration or whenever a local coin flip is performed. He also proves that a bivalent configuration always leads to a nullvalent configuration or to a configuration in which a local coin flip can be scheduled next. Finally, in nullvalent configurations, he shows that the coin flipping lower bound applies. A polylogarithmic gap remains between the upper and lower bounds for the amount of work to solve randomized consensus on asynchronous models.

Bar-Joseph and Ben-Or [17] extended Aspnes' result to synchronous message-passing systems, obtaining a lower bound of $\Omega(t/\sqrt{n \log n})$ rounds (with high probability) for t-resilient randomized consensus among n processes. They also gave a matching upper bound in this model. In contrast, for deterministic algorithms, $t+1$ rounds are needed [43]. If the power of the adversary scheduler is restricted so that its choices can only depend on the actions of the processes (and cannot depend directly on the outcome of coin flips), then faster algorithms are possible: there is a randomized consensus algorithm using `registers` with $O(\log^2 n)$ expected running time per process [30]. Against such non-adaptive adversaries, even expected constant time algorithms have been obtained [37, 44, 95]. Byzantine agreement, where faulty processes can behave maliciously is more difficult than consensus. However, no bigger lower bounds are known for randomized Byzantine agreement than for randomized consensus.

10 Conclusions

Why are lower bounds important for distributed computing? They help us to better understand the nature of distributed computing: what mades certain problems hard, what makes a model powerful, and how different models compare. They tell us when to stop looking for better solutions or, at least, which approaches will not work. If we have a problem that we need to solve, despite a lower bound, the lower bound may indicate ways to adjust the problem specification or the modelling of the environment to allow reasonable solutions. Finally, trying to prove lower bounds can suggest new and different algorithms, especially when attempts to prove the bounds fail.

This survey has presented many lower bound results and different techniques for proving them. We hope it will encourage you to try to prove lower bounds for the distributed computing problems you encounter.

If someone says "can't" that shows you what to do [29].

Acknowledgements

We thank Maurice Herlihy for numerous enlightening discussions. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario.

References

1. Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.
2. Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6), pages 1231–1274, November 1995.
3. Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. The power of multi-objects. *Information and Computation*, 153(1), pages 117–138, August 1999.
4. Yehuda Afek and Gideon Stupp. Optimal time-space tradeoff for shared memory leader election. *Journal of Algorithms*, 25(1), pages 95–117, October 1997.
5. Rajeev Alur, Hagit Attiya, and Gadi Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing*, 26(2), pages 539–556, 1997.
6. James H. Anderson and Mark Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, 1999.
7. Richard Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *ACM Symposium on Theory of Computing*, pages 370–380, 1991.
8. James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3), pages 415–450, 1998.
9. James Aspnes and Maurice Herlihy. Fast, randomized consensus using shared memory. *Journal of Algorithms*, 11(2), pages 441–461, September 1990.
10. James Aspnes and Orli Waarts. Randomized consensus in $O(n \log^2 n)$ operations per processor. *SIAM Journal on Computing*, 25(5), pages 1024–1044, October 1996.
11. Hagit Attiya. A direct proof of the asynchronous lower bound for k-set consensus. Technical Report 0930, Computer Science Department, Technion, 1998. Available from `http://www.cs.technion.ac.il/Reports`.
12. Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1), pages 122–152, 1994.
13. Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 49–61, 1998. Full version available from `www.cs.technion.ac.il/~hagit`.
14. Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4), pages 725–763, 1994.
15. Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, 1996. Full version available from `www.cs.technion.ac.il/~hagit`.
16. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.
17. Ziv Bar-Joseph and Michael Ben-Or. A tight lower bound for randomized synchronous consensus. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 193–199, 1998.

18. Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms*, 11(3), pages 420–440, September 1990.
19. Ofer Biran, Shlomo Moran, and Shmuel Zaks. Deciding 1-solvability of distributed task is NP-hard. In *Graph-Theoretic Concepts in Computer Science. 16th International Workshop*, volume 484 of *LNCS*, pages 206–220, 1990.
20. Ofer Biran, Shlomo Moran, and Shmuel Zaks. Tight bounds on the round complexity of distributed 1-solvable tasks. *Theoretical Computer Science*, 145(1–2), pages 271–290, July 1995.
21. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symposium on Theory of Computing*, pages 91–100, 1993.
22. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 41–52, 1993.
23. Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 189–198, 1997.
24. Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. Technical Report TM-573, Laboratory for Computer Science, Massachusetts Institute of Technology, December 1997.
25. Gadi Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Distributed Algorithms, 5th International Workshop*, volume 579 of *LNCS*, pages 143–150, 1991.
26. James Burns, Paul Jackson, Nancy Lynch, Michael Fischer, and Gary Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1), pages 183–205, January 1982.
27. James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2), pages 171–184, December 1993.
28. Jonathan Buss, Paris Kanellakis, Prabhakar Ragde, and Alex Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1), pages 45–86, 1996.
29. John Cage. An autobiographical statement. *Southwest Review*, 76(1), page 59, 1991.
30. Tushar Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 166–175, 1996.
31. Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. The h_m^r hierarchy is not robust. Manuscript, 1994.
32. Tushar Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 334–343, 1994.
33. Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 287–296, 1998.
34. Soma Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1), pages 132–158, July 1993.

35. Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. A tight lower bound for k-set agreement. In *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pages 206–215, 1993. A full version will appear in the *Journal of the ACM*.
36. Soma Chaudhuri and Paul Reiners. Understanding the set consensus partial order using the Borowsky-Gafni simulation. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, 1996.
37. Benny Chor, Michael Merritt, and David Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3), pages 591–614, 1989.
38. Benny Chor and Lee-Bath Nelson. Solvability in asynchronous environments II: Finite interactive tasks. *SIAM Journal on Computing*, 29(2), pages 351–377, April 2000.
39. Israel Cidon and Yuval Shavitt. Message terminating algorithms for anonymous rings of unknown size. *Information Processing Letters*, 54(2), pages 111–119, April 1995.
40. Robert Cori and Shlomo Moran. Exotic behaviour of consensus numbers. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 101–115, 1994.
41. Roberto De Prisco, Dahlia Malkhi, and Michael K. Reiter. On k-set consensus in asynchronous systems. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 257–265, 1999.
42. Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6), pages 779–805, 1997.
43. Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2), pages 156–186, October 1990.
44. Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM Journal on Computing*, 26(4), pages 873–933, August 1997.
45. Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5), pages 843–862, September 1998.
46. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), pages 374–382, April 1985.
47. Eli Gafni. Distributed computing. In Mikhail J. Atallah, ed., *Algorithms and Theory of Computation Handbook*, chapter 48. CRC Press, 1998.
48. Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.
49. Eli Gafni and Elias Koutsoupias. Three-processor tasks are undecidable. *SIAM Journal on Computing*, 28(3), pages 970–983, January 1999.
50. Eric Goubault and Thomas P. Jensen. Homology of higher dimensional automata. In *Proc. 3rd International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 254–268, 1992.
51. Jeremy Gunawardena. Homotopy and concurrency. *Bulletin of the EATCS*, 54, pages 184–193, October 1994.
52. John Havlicek. Computable obstructions to wait-free computability. *Distributed Computing*, 13(2), pages 59–83, 2000.

53. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pages 124–149, January 1991.
54. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5), pages 745–770, November 1993.
55. Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 324–333, 1994.
56. Maurice Herlihy and Sergio Rajsbaum. Algebraic spans. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, 1995.
57. Maurice Herlihy and Sergio Rajsbaum. Algebraic topology and distributed computing: A primer. In Jan van Leeuwen, ed., *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 203–217. Springer, 1995.
58. Maurice Herlihy and Sergio Rajsbaum. The decidability of distributed decision tasks. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 589–598, 1997. Full version available from `www.cs.brown.edu/people/mph/decide.html`.
59. Maurice Herlihy and Sergio Rajsbaum. A wait-free classification of loop agreement tasks. In *Distributed Computing, 12th International Symposium*, volume 1499 of *LNCS*, pages 175–185, 1998.
60. Maurice Herlihy and Sergio Rajsbaum. New perspectives in distributed computing. In *Mathematical Foundations of Computer Science: 24th International Symposium*, volume 1672 of *LNCS*, pages 170–186, 1999.
61. Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–142, 1998. Full version available from `www.cs.brown.edu/people/mph/hrt.html`.
62. Maurice Herlihy and Eric Ruppert. On the existence of booster types. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, 2000. To appear.
63. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6), pages 858–923, November 1999.
64. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), pages 463–492, July 1990.
65. Gunnar Hoest and Nir Shavit. Towards a topological characterization of asynchronous complexity. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 199–208, 1997.
66. Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4), pages 592–614, July 1997.
67. Prasad Jayanti. A lower bound on the local time complexity of universal constructions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192, 1998.
68. Prasad Jayanti. Solvability of consensus: Composition breaks down for nondeterministic types. *SIAM Journal on Computing*, 28(3), pages 782–797, September 1998.
69. Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

70. Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3), pages 451–500, 1998.
71. Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2), pages 438–456, 2000.
72. Prasad Jayanti and Sam Toueg. Wakeup under read/write atomicity. In *Distributed Algorithms, 4th International Workshop*, volume 486 of *LNCS*, pages 277–288, 1990.
73. Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality and decidability of consensus. In *Distributed Algorithms, 6th International Workshop*, volume 647 of *LNCS*, pages 69–84, 1992.
74. C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 285–296, 1990.
75. Jon M. Kleinberg and Sendhil Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–144, 1993.
76. Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4), pages 579–601, October 1988.
77. Eyal Kushilevitz, Yishay Mansour, Michael O. Rabin, and David Zuckerman. Lower bounds for randomized mutual exclusion. *SIAM Journal on Computing*, 27(6), pages 1550–1563, 1998.
78. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8), pages 453–455, 1974.
79. Leslie Lamport. The mutual exclusion problem. Part II: Statement and solutions. *Journal of the ACM*, 33(2), pages 327–348, 1986.
80. Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, volume B, chapter 18. Elsevier, 1990.
81. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), pages 382–401, July 1982.
82. Wai-Kau Lo. More on t-resilience vs. wait-freedom. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 110–119, 1995.
83. Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: wait-free hierarchies are not robust. In *Proc. 29th ACM Symposium on Theory of Computing*, pages 579–588, 1997.
84. Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, ed., *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, Greenwich, Connecticut, 1987.
85. Ronit Lubitch and Shlomo Moran. Closed schedulers: A novel technique for analyzing asynchronous protocols. *Distributed Computing*, 8(4), pages 203–210, 1995.
86. Nancy A. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–27, 1989.

87. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
88. Shlomo Moran and Lihu Rappoport. On the robustness of h_m^r. In *Distributed Algorithms, 10th International Workshop*, volume 1151 of *LNCS*, pages 344–361, 1996.
89. Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent counting. *Journal of Computer and System Sciences*, 53(1), pages 61–78, August 1996.
90. Shlomo Moran and Yaron Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3), pages 145–151, November 1987.
91. Yoram Moses and Sergio Rajsbaum. The unified structure of consensus: a layered analysis approach. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132, 1998.
92. Naor and Roth. Parallel algorithms with processor failures and delays. *Random Structures and Algorithms*, 1995.
93. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), pages 228–234, April 1980.
94. Gary L. Peterson. Properties of a family of booster types. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, page 281, 1999.
95. Michael Rabin. Randomized Byzantine generals. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.
96. Ophir Rachman. Anomalies in the wait-free hierarchy. In *Distributed Algorithms, 8th International Workshop*, volume 857 of *LNCS*, pages 156–163, 1994.
97. G. M. Reed, A. W. Roscoe, and R. F. Wachter, eds. *Topology and Category Theory in Computer Science*. Clarendon Press, Oxford, 1991.
98. Eric Ruppert. Determining consensus numbers. *SIAM Journal on Computing*. To appear.
99. Eric Ruppert. Consensus numbers of multi-objects. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 211–217, 1998.
100. Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5), pages 1449–1483, March 2000.
101. Eric Schenk. Faster approximate agreement with multi-writer registers. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1995.
102. Eric Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems*. Ph.D. thesis, Department of Computer Science, University of Toronto, September 1996.
103. Eric Schenk. The consensus hierarchy is not robust. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 279, 1997.
104. E. Sperner. Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes. *Abhandlungen aus dem Mathematischen Seminar der Hamburgischen Universität*, 6(3/4), pages 265–272, September 1928.
105. Gadi Taubenfeld, Shmuel Katz, and Shlomo Moran. Initial failures in distributed computations. *International Journal of Parallel Programming*, 18(4), pages 255–276, August 1989.
106. Gadi Taubenfeld and Shlomo Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1), pages 1–20, February 1996.

Adaptive Mutual Exclusion with Local Spinning*

James H. Anderson and Yong-Jik Kim

Department of Computer Science, University of North Carolina at Chapel Hill

Abstract. We present the first adaptive algorithm for N-process mutual exclusion under read/write atomicity in which all busy waiting is by local spinning. In our algorithm, each process p performs $O(min(k, \log N))$ remote memory references to enter and exit its critical section, where k is the maximum "point contention" experienced by p. The space complexity of our algorithm is $\Theta(N)$, which is clearly optimal.

1 Introduction

In this paper, we consider adaptive solutions to the mutual exclusion problem [7] under read/write atomicity. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [6, 11, 13]. Two notions of contention have been considered in the literature: "interval contention" and "point contention" [1]. These two notions are defined with respect to a history H. The *interval contention* over H is the number of processes that are active in H, i.e., that execute outside of their noncritical sections in H. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. In this paper, we consider only point contention. Throughout the paper, we let N denote the number of processes in the system, and we let k denote the point contention experienced by an arbitrary process over a history that starts when it becomes active and ends when it once again becomes inactive.

In previous work on adaptive mutual exclusion algorithms, two time complexity measures have been considered: "remote step complexity" and "system response time." The *remote step complexity* of an algorithm is the maximum number of shared-memory operations required by a process to enter and then exit its critical section, assuming that each "**await**" statement is counted as one operation [13]. The *system response time* is the length of time between critical section entries, assuming each enabled read or write operation is executed within some constant time bound [6]. Several read/write mutual exclusion algorithms have been presented that are adaptive to some degree under these time complexity measures. One of the first such algorithms was an algorithm of Styer that has $O(min(N, k \log N))$ remote step complexity and $O(min(N, k \log N))$ response time [13]. Choy and Singh later improved upon Styer's result by presenting an algorithm with $O(N)$ remote step complexity and $O(k)$ response time

* Work supported by NSF grants CCR 9732916 and CCR 9972211.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 29–43, 2000.

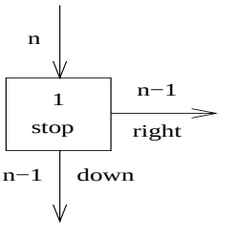
[6]. More recently, Attiya and Bortnikov presented an algorithm with $O(k)$ remote step complexity and $O(\log k)$ response time [5].

Recent work on scalable local-spin mutual exclusion algorithms has shown that *the* most crucial factor in determining an algorithm's performance is the amount of interconnect traffic it generates [4, 8, 10, 14]. In light of this, we define the *time complexity* of a mutual exclusion algorithm to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section. A *remote memory reference* is a shared variable access that requires an interconnect traversal. In local-spin algorithms, all busy-waiting loops are required to be read-only loops in which only locally-accessible shared variables are accessed that do not require an interconnect traversal. On a distributed shared-memory multiprocessor, a shared variable is locally accessible if it is stored in a local memory module. On a cache-coherent multiprocessor, a shared variable is locally accessible if it is stored in a local cache line.

The first local-spin algorithms were algorithms in which read-modify-write instructions are used to enqueue blocked processes onto the end of a "spin queue" [4, 8, 10]. Each of these algorithms has $O(1)$ time complexity; thus, adaptivity is clearly a non-issue if appropriate read-modify-write instructions are available. Yang and Anderson were the first to consider local-spin algorithms under read/write atomicity [14]. They presented a read/write mutual exclusion algorithm with $\Theta(\log N)$ time complexity in which instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree. They also presented a "fast-path" variant of this algorithm that allows the tree to be bypassed in the absence of contention. Although the contention-free time complexity of this algorithm is $O(1)$, its time complexity under contention is $\Theta(N)$ in the worst case, rather than $\Theta(\log N)$. In recent work, Anderson and Kim presented a new fast-path mechanism that results in with $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ time complexity under contention, when used with Yang and Anderson's algorithm [3].

All of the previously-cited adaptive algorithms are not local-spin algorithms, and thus they have unbounded time complexity under the remote-memory-references time measure. One could argue that for an algorithm to be considered truly adaptive, it must be adaptive under this measure. After all, the underlying hardware does not distinguish between remote memory references generated by **await** statements and remote memory references generated by other statements. Surprisingly, while adaptivity and local spinning have been the predominate themes in recent work on mutual exclusion, the problem of designing an adaptive, local-spin algorithm under read/write atomicity has remained open. In this paper, we close this problem by presenting an algorithm that has $O(min(k, \log N))$ time complexity under the remote-memory-references measure.

Our algorithm can be seen as an extension of the fast-path algorithm of Anderson and Kim [3]. This algorithm was devised by thinking about connections between fast-path mechanisms and long-lived renaming [12]. Long-lived renaming algorithms are used to "shrink" the size of the name space from which process identifiers are taken. The problem is to design operations that processes may in-




```
shared variables X: {⊥} ∪ {0..N − 1} init ⊥;
                 Y: boolean init false

private variable dir: {D, R, S} /* down, right, stop */

1: X := p;
2: if Y then dir := R
   else
3:      Y := true;
4:      if X = p then dir := S
                 else dir := D
        fi
   fi
```

Fig. 1. The splitter element and the code fragment that implements it.

voke in order to acquire new names from the reduced name space when they are needed, and to release any previously-acquired name when it is no longer needed. In Anderson and Kim's algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. Our adaptive algorithm can be seen as a generalization of Anderson and Kim's fast-path mechanism in which *every* name is associated with some "path" to the critical section. The length of the path taken by a process is determined by the point contention that it experiences.

2 Adaptive Algorithm

In our adaptive algorithm, code sequences from several other algorithms are used. In Sec. 2.1, we present a review of these other algorithms and discuss some of the basic ideas underlying our algorithm. Then, in Sec. 2.2, we present a detailed description of our algorithm.

2.1 Related Algorithms and Key Ideas

At the heart of our algorithm is the splitter element from the grid-based long-lived renaming algorithm of Moir and Anderson [12]. This splitter element was actually first used in Lamport's fast mutual exclusion algorithm [9]. The splitter element is defined by the code fragment shown in Fig. 1. (In this and subsequent figures, we assume that each labeled sequence of statements is atomic; in each figure, each labeled sequence reads or writes at most one shared variable.) Each process that invokes this code fragment either stops, moves down, or moves right (the move is defined by the value assigned to the variable *dir*). One of the key properties of the splitter that makes it so useful is the following: if n processes invoke a splitter, then at most one of them can stop at that splitter, at most $n-1$ can move right, and at most $n-1$ can move down.

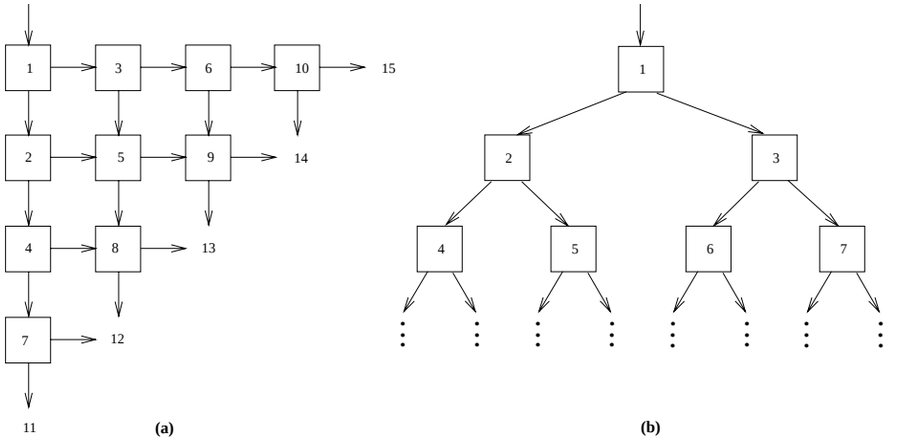


Fig. 2. **(a)** Renaming grid (depicted for $N = 5$). **(b)** Renaming tree.

Because of these properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid as shown in Fig. 2(a). A name is associated with each splitter. If the grid has N rows and N columns, then by induction, every process eventually stops at some splitter. When a process stops at a splitter, it acquires the name associated with that splitter. In the *long-lived* renaming problem [12], processes must have the ability to release the names they acquire. In the grid algorithm, a process can release its name by resetting each splitter on the path traversed by it in acquiring its name. A splitter can be reset by resetting its Y variable to *true*. For the renaming mechanism to work correctly, it is important that a splitter be reset *only* if there are no processes "downstream" from it (i.e., in the sub-grid "rooted" at that splitter). In Moir and Anderson's algorithm, it takes $O(N)$ time to determine whether there are "downstream" processes. This is because each process checks every other process individually to determine if it is downstream from a splitter. As we shall see, a more efficient reset mechanism is needed for our adaptive algorithm.

The main idea behind our algorithm is to let an arbitration tree form dynamically within a structure similar to the renaming grid. This tree may not remain balanced, but its height is proportional to contention. The job of integrating the renaming aspects of the algorithm with the arbitration tree is greatly simplified if we replace the grid by a binary tree of splitters as shown in Fig. 2(b). (Since we are now working with a tree, we will henceforth refer to the directions associated with a splitter as stop, left, and right.) Note that this results in many more names than before. However, this is not a major concern, because we are really not interested in minimizing the name space. The arbitration tree is defined by associating a three-process mutual exclusion algorithm with each node in the renaming tree. This three-process algorithm can be implemented in constant time using the local-spin mutual exclusion algorithm of Yang and Anderson [14]. We

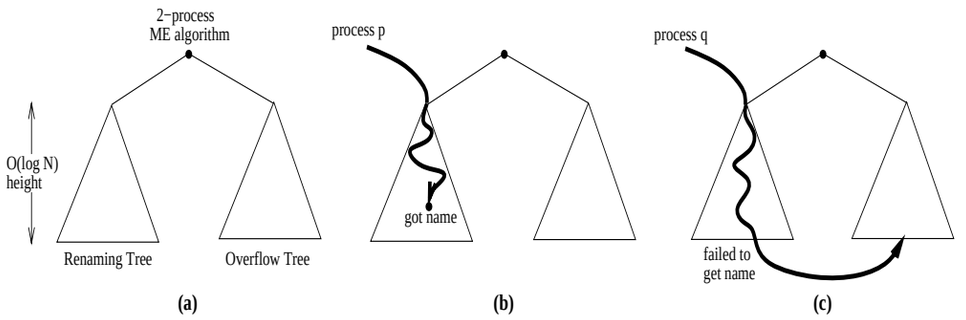


Fig. 3. **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

explain below why a three-process algorithm is needed instead of a two-process algorithm (as one would expect to have in an arbitration tree).

In our algorithm, a process p performs the following basic steps. (For the moment, we are ignoring certain complexities that must be dealt with.)

Step 1 p first acquires a new name by moving down from the root of the renaming tree, until it stops at some node. In the steps that follow, we refer to this node as p's *acquired node*. p's acquired node determines its starting point in the arbitration tree.

Step 2 p then competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root. Note that a node's entry section may be invoked by the process that stopped at that node, and one process from each of the left and right subtrees beneath that node. This is why a three-process algorithm is needed.

Step 3 After competing within the arbitration tree, p executes its critical section.

Step 4 Upon completing its critical section, p releases its acquired name by reopening all of the splitters on the path from its acquired node to the root.

Step 5 After releasing its name, p executes each of the three-process exit sections on the path from the root to its acquired node.

If we were to use a binary tree of height N, just as we previously had a grid with N row and N columns, then the total number of nodes in the tree would be $\Theta(2^N)$. We circumvent this problem by defining the tree's height to be $\lfloor \log N \rfloor$, which results in a tree with $\Theta(N)$ nodes. With this change, a process could "fall off" the end of the tree without acquiring a name. However, this can happen only if contention is $\Omega(\log N)$. To handle processes that "fall off the end," we introduce a second arbitration tree, which is implemented using Yang and Anderson's local-spin arbitration-tree algorithm [14]. We refer to the two trees used in our algorithm as the *renaming tree* and *overflow tree*, respectively. These two trees are connected by placing a two-process version of Yang and Anderson's algorithm on top of each tree, as illustrated in Fig. 3(a). Fig. 3(b)

illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Fig. 3(c) illustrates the steps that might be taken by a process q if contention is $\Omega(\log N)$.

A major difficulty that we have ignored until this point is that of efficiently reopening a splitter, as described in Step 4 above. In Moir and Anderson's renaming algorithm, it takes $O(N)$ time to reopen a splitter. To see why reopening a splitter is difficult, consider again Fig. 1. If a process does succeed stopping at a splitter, then that process can reopen the splitter itself by simply assigning $Y := \mathit{true}$. On the other hand, if no process succeeds in stopping at a splitter, then some process that moved left or right from that splitter must reset it. Unfortunately, because processes are asynchronous and communicate only by means of atomic read and write operations, it can be difficult for a left- or right-moving process to know whether some process has stopped at a splitter.

Anderson and Kim solved this problem in their fast-path mutual exclusion algorithm by exploiting the fact that much of the reset code can be executed within a process's critical section [3]. Thus, the job of designing efficient reset code is much easier here than when designing a *wait-free* long-lived renaming algorithm. As mentioned earlier, in Anderson and Kim's fast-path algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. In our adaptive algorithm, we must efficiently manage acquisitions and releases for a set of names.

2.2 Detailed Description

Having introduced the major ideas that underlie our algorithm, we now present a detailed description of the algorithm and its properties. We do this in three steps. First, we consider a version of the algorithm in which unbounded memory is used to reset splitters in constant time. Second, we consider a variant of the algorithm with $\Theta(N^2)$ space complexity in which all variables are bounded. Third, we present another variant that has $\Theta(N)$ space complexity. In explaining these algorithms, we actually present proof sketches for some of the key properties of each algorithm. Our intent is to use these proof sketches as a means for intuitively explaining the basic mechanisms of each algorithm. A formal correctness proof for the final algorithm is presented in the full version of this paper [2].

Algorithm U. The first algorithm, which we call Algorithm U (for unbounded), is shown in Fig. 4. Before describing how this algorithm works, we first examine its basic structure. At the top of Fig. 4, definitions of two constants are given: D, which is the maximum level in the renaming tree (the root is at level 0), and T, which gives the total number of nodes in the renaming tree. As mentioned earlier, the renaming tree is comprised of a collection of splitters. These splitters are indexed from 1 to T. If splitter i is not a leaf, then its left and right children are splitters $2i$ and $2i + 1$, respectively.

Each splitter i is defined by four shared variables and an infinite shared array: $X[i]$, $Y[i]$, $\mathit{Reset}[i]$, $\mathit{Rnd}[i]$ (the array), and $\mathit{Acquired}[i]$. Variables $X[i]$ and $Y[i]$ are as in Fig. 1, with the exception that $Y[i]$ now has an additional

```
const
    D = ⌊log N⌋;                          /* depth of renaming tree = O(log N) */
    T = 2^(D+1) − 1                       /* size of renaming tree = O(N) */
type
    Ytype = record free: boolean; rnd: 0..∞ end;      /* stored in one word */
    Dtype = {L, R, S};                    /* splitter moves (left, right, stop) */
    Ptype = record nd: 1..2T + 1; dir: Dtype end      /* path information */

shared variables
    X: array[1..T] of 0..∞;
    Y, Reset: array[1..T] of Ytype init (true, 0);
    Rnd: array[1..T][0..∞] of boolean init false;
    Acquired: array[1..T] of boolean init false

private variables
    nd, n: 1..2T + 1;
    lvl, j: 0..D + 1;
    y: Ytype;   dir: Dtype;
    path: array[0..D] of Ptype

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:      X[nd], dir := p, S;
3:      y := Y[nd];
        if ¬y.free then dir := R
        else
4:          Y[nd] := (false, 0);
5:          if X[nd] ≠ p ∨
6:                Acquired[nd] then
              dir := L
            else
7:              Rnd[nd][y.rnd] := true;
8:              if Reset[nd] ≠ y then
9:                  Rnd[nd][y.rnd], dir := false, L
        fi fi fi;
10:     path[lvl] := (nd, dir);
        if dir ≠ S then
            lvl, nd := lvl + 1, 2 · nd;
            if dir = R then nd := nd + 1 fi
        fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
11:     Acquired[nd] := true;
        for j := lvl downto 0 do
12:         ENTRY3(path[j].nd, path[j].dir)
        od;
13:     ENTRY2(0)
    else     /* didn't get a name */
14:     ENTRYN(p);
15:     ENTRY2(1)
    fi;
16: Critical Section;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
        if path[j].dir ≠ R then
17:         n := path[j].nd;
18:         y := Reset[n];
19:         Reset[n] := (false, y.rnd);
20:         if j = lvl ∨
                ¬Rnd[n][y.rnd] then
21:             Reset[n] := (true, y.rnd + 1);
22:             Y[n] := (true, y.rnd + 1)
            fi
        fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
23:     EXIT2(0);
        for j := 0 to lvl do
24:         EXIT3(path[j].nd, path[j].dir)
        od;
25:     Acquired[nd] := false
    else
26:     EXIT2(1);
27:     EXITN(p)
    fi
od
```

Fig. 4. Algorithm U: adaptive algorithm with unbounded memory.

integer *rnd* field. As explained below, Algorithm U works by associating "round numbers" with the various rounds of competition for the name corresponding to each splitter. In Algorithm U, these round numbers grow without bound. The *rnd* field of $Y[i]$ gives the current round number for splitter i. $Reset[i]$ is used to reinitialize the *rnd* field of $Y[i]$ when name i is released. $Rnd[i][r]$ is used to identify a potential "winning" process that has succeeded in acquiring name i in round r. $Acquired[i]$ is set when some process acquires name i.

Each process descends the renaming tree, starting at the root, until it either acquires a name or "falls off the end" of the tree, as discussed earlier. A process determines if it can acquire name i by executing statements 2-10 with $nd = i$. Of these, statements 2-5 correspond to the splitter code in Fig. 1. Statements 6-9 are executed as part of a handshaking mechanism that prevents a process that is releasing a name from adversely interfering with processes attempting to acquire that name; this mechanism is discussed in detail below. Statement 10 simply prepares for the next iteration of the **repeat** loop (if there is one).

If a process p succeeds in acquiring a name while descending within the renaming tree, then it competes within the renaming tree by moving up from its acquired name to the root, executing the three-process entry sections on this path (statements 11-12). Each of these three-process entry sections is denoted "$\texttt{ENTRY}_3(n, d)$," where n is the corresponding tree node, and d is the "identity" of the invoking process. The "identity" that is used is simply the invoking process's direction out of node n (S, L, or R) when it descended the renaming tree. After ascending the renaming tree, p invokes the two-process entry section "on top" of the renaming and overflow trees (as illustrated in Fig. 3(a)) using "0" as a process identifier (statement 13). This entry section is denoted "$\texttt{ENTRY}_2(0)$."

If a process p does *not* succeed in acquiring a name while descending within the renaming tree, then it competes within the overflow tree (statement 14), which is implemented using Yang and Anderson's N-process arbitration-tree algorithm. The entry section of this algorithm is denoted $\texttt{ENTRY}_N(p)$. Note that p uses its own process identifier in this algorithm. After competing within the overflow tree, p executes the two-process algorithm "on top" of both trees using "1" as a process identifier (statement 15). This entry section is denoted "$\texttt{ENTRY}_2(1)$."

After completing the appropriate two-process entry section, process p executes its critical section (statement 16). It then resets each of the splitters that it visited while descending the renaming tree (statements 17-22). This reset mechanism is discussed in detail below. Process p then executes the exit sections corresponding to the entry sections it executed previously (statements 23-27). The exit sections are specified in a manner that is similar to the entry sections.

We now consider in detail the code fragments that are executed to acquire (statements 2-10) or reset (statements 18-22) some splitter i. To facilitate this discussion, we will index these statements by i. For example, when we refer to the execution of statement $4[i]$ by process p, we mean the execution of statement 4 by p when its private variable nd equals i. Similarly, $18[i]$ denotes the execution of statement 18 with $n = i$.

As explained above, one of the problems with the splitter code is that it

is difficult for a left- or right-moving process at splitter i to know which (if any) process has acquired name i. In Algorithm U, this problem is solved by viewing the computation involving each splitter as occurring in a sequence of rounds. Each round ends when the splitter is reset. During a round, at most one process succeeds in acquiring the name of the splitter. Note that it is possible that *no* process acquires the name during a round. So that processes can know the current round number at splitter i, an additional *rnd* field has been added to $Y[i]$. This field will increase without bound over time, so we will never have to worry about round numbers being reused.

With the added *rnd* field, a left- or right-moving process at splitter i has a way of identifying a process that has acquired the name at splitter i. To see how this works, consider what happens during round r at node i. Of the processes that participate in round r at node i, at least one will read $Y[i] = (true, r)$ at statement $3[i]$ and assign $Y[i] := (false, 0)$ at statement $4[i]$. By the correctness of the original splitter code, of the processes that assign $Y[i]$, at most one will reach statement $7[i]$. A process that reaches statement $7[i]$ will either stop at node i or be deflected left. This gives us two cases to analyze: of the processes that read $Y[i] = (true, r)$ at statement $3[i]$ and assign $Y[i]$ at statement $4[i]$, either all are deflected left, or one, say p, stops at splitter i.

In the former case, at least one of the left-moving processes finds $Rnd[i][r]$ to be false at statement $20[i]$, and then reopens splitter i by executing statements $21[i]$ and $22[i]$, which establish $Y[i] = (true,\ r + 1) \ \wedge \ Y[i] = Reset[i]$. To see why at least one process executes statements $21[i]$ and $22[i]$, note that each process under consideration reads $Y[i] = (true, r)$ at statement $3[i]$, and thus its $y.rnd$ variable equals r while executing within statements $4[i]$-$9[i]$. Note also that $Rnd[i][r] = true$ is established only by statement $7[i]$. Moreover, each process deflected left at statement $9[i]$ first assigns $Rnd[i][r] := false$. Thus, at least one of the left-moving processes finds $Rnd[i][r]$ to be false at statement $20[i]$.

In the case that there is a winning process p that stops at splitter i during round r, we must argue that **(i)** p reopens splitter i upon leaving it, and **(ii)** no left- or right-moving process "prematurely" reopens splitter i before p has left it. Establishing (i) is straightforward. Process p will reopen the splitter by executing statements $18[i]$-$22[i]$ and 25, which establish $Y[i] = (true, r+1) \wedge Acquired[i] = false \ \wedge \ Y[i] = Reset[i]$. Note that the assignment to *Acquired* at statement 25 prevents the reopening of splitter i from actually taking effect until after p has finished executing its exit section.

To establish (ii), suppose, to the contrary, that some left- or right-moving process reopens splitter i by executing statement $22[i]$ while p is executing within statements $10[i]$-13 and 16-25. (Note that, because p stops at splitter i, it doesn't iterate again within the **repeat** loop.) Let q be the first left- or right-moving process to execute statement $22[i]$. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute statement $22[i]$ while p is executing within statements 16-22. Moreover, if p is executing within statements 12-13 or 23-25, then *Acquired* is true, and hence the splitter is closed. The remaining possibility is that p is enabled to execute statement $10[i]$ or 11. (Note that, in

this case, if q *were* to reopen splitter i then we could end up with two processes concurrently invoking $\texttt{ENTRY}_3(i, S)$ at statement 12, i.e., both processes use S as a "process identifier." The ENTRY calls obviously cannot be assumed to work correctly if such a scenario could happen.)

So, assume that q executes statement $22[i]$ while p is enabled to execute statement $10[i]$ or 11. For this to happen, q must have read $Rnd[i][r] = \mathit{false}$ at statement $20[i]$ before p assigned $Rnd[i][r] := \mathit{true}$ at statement $7[i]$. (Recall that all the processes under consideration read $Y[i] = (\mathit{true}, r)$ at statement $2[i]$. This is why p writes to $Rnd[i][r]$ instead of some other element of $Rnd[i]$. q reads from $Rnd[i][r]$ at statement $20[i]$ because it is the first process to attempt to reset splitter i, which implies that q reads $Reset[i] = (\mathit{true}, r)$ at statement $18[i]$.) Because q executes statement $20[i]$ before p executes statement $7[i]$, statement $19[i]$ is executed by q before statement $8[i]$ is executed by p. Thus, p must have found $Reset[i] \neq y$ at statement $7[i]$, i.e., it was deflected left at splitter i, which is a contradiction. It follows from the explanation given here that splitter i is eventually reset for round $r + 1$, i.e., we have the following property.

Property 1: Let $\mathcal{S}$ be the set of all processes that read $Y[i].rnd = r$ at statement $3[i]$. If $\mathcal{S}$ is nonempty, then $Y[i] = (\mathit{true},\ r+1) \wedge Y[i] = Reset[i] \wedge Acquired[i] = \mathit{false}$ is eventually established, and at all states after it is first established, no process in set $\mathcal{S}$ stops at splitter i. □

Because the splitters are always reset properly, it follows that the ENTRY and EXIT routines are always invoked properly. If these routines are implemented using Yang and Anderson's local-spin algorithm, then since that algorithm is starvation-free, Algorithm U is as well.

Having dispensed with basic correctness, we now informally argue that Algorithm U is contention sensitive. For a process p to descend to a splitter at level l in the renaming tree, it must have been deflected left or right at each prior splitter it accessed. Just as with the original grid-based long-lived renaming algorithm [12], this can only happen if the point contention experienced by p is $\Omega(l)$. Note that the time complexity per level of the renaming tree is constant. Moreover, with the ENTRY and EXIT calls implemented using Yang and Anderson's algorithm [14], the $\texttt{ENTRY}_2$, $\texttt{EXIT}_2$, $\texttt{ENTRY}_3$, and $\texttt{EXIT}_3$ calls take constant time, and the $\texttt{ENTRY}_N$ and $\texttt{ENTRY}_N$ calls take $\Theta(\log N)$ time. Note that the $\texttt{ENTRY}_N$ and $\texttt{ENTRY}_N$ routines are called by a process only if its point contention is $\Omega(\log N)$. Thus, we have the following.

Lemma 1: Algorithm U is a correct, starvation-free mutual exclusion algorithm with $O(min(k, \log N))$ time complexity and unbounded space complexity. □

Of course, the problem with Algorithm U is that the *rnd* field of $Y[i]$ that is used for assigning round numbers grows without bound. We now consider a variant of Algorithm U in which space is bounded.

Algorithm B. In Algorithm B (for bounded), which is shown in Fig. 5, modulo-N addition (denoted by $\oplus$) is used when incrementing $Y[i].rnd$. With this change,

the following potential problem arises. A process p may reach statement 8[i] in Fig. 5 with $y.rnd = r$ and then be delayed. While delayed, other processes may repeatedly increment $Y[i].rnd$ (statement 27[i]) until it "cycles back" to r. Another process q could then reach statement 8[i] with $y.rnd = r$. This is a problem because p and q may interfere with each other in updating $Rnd[i][r]$.

Algorithm B prevents such a scenario from happening by preventing $Y[i].rnd$ from cycling while a process p that stops at splitter i executes within statements 8[i]-31. Informally, cycling is prevented by requiring process p to erect an "obstacle" that prevents $Y[i].rnd$ from being incremented beyond the value p. More precisely, note that before reaching statement 8[i], process p must first assign $Obstacle[p] := i$ at statement 5[i]. Note further that before a process can increment $Y[i].rnd$ from r to $r \oplus 1$ (statement 27[i]), it must first read $Obstacle[r]$ (statement 25[i]) and find it to have a value different from i. This check prevents $Y[i].rnd$ from being incremented beyond the value p while p executes within statements 8[i]-31. Note that process p resets $Obstacle[p]$ to 0 at statement 18. This is done to ensure that p's own obstacle doesn't prevent it from incrementing a splitter's round number.

To this point, we have explained every difference between Algorithms U and B except one: in Fig. 5, there are added assignments to elements of Y and X (statements 20 and 21) after the critical section. The reason for these assignments is as follows. Suppose some process p is about to assign $Obstacle[p] := true$ at statement 5[i], but gets delayed. (In other words, p is "about to" erect an obstacle at splitter i.) We must ensure that if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p. Let r be the value read from $Y.rnd$ by p at statement 3[i]. For $Y.rnd$ to be incremented beyond p, some other process q that reads $Y.rnd = r$ must attempt to reopen splitter i.

So, suppose that process q reopens splitter i by executing statement 27[i] while p is delayed at statement 5[i]. If process q executes statement 21[i] after p executes statement 2[i], then p will find $X[i] \neq p$ at statement 6[i] and will be deflected left. So, assume that q executes statement 21[i] before p executes statement 2[i]. This implies that q establishes $Y[i].free = false$ by executing statement 20[i] before p reads $Y[i]$ at statement 3[i]. Note that $Y[i].free = true$ is only established within a critical section (statement 27[i]). Also, note that we have established the following sequence of statement executions (perhaps interleaved with statement executions of other processes): q executes statements 20[i] and 21[i]; p executes statements 2[i]-5[i]; q executes statement 27[i] (q's execution of statements 22[i]-26[i] may interleave arbitrarily with p's execution of statements 2[i]-5[i]). Because statements 17[i]-27[i] are executed as a critical section, this implies that p reads $Y[i].free = false$ at statement 3[i], and thus does not reach statement 5[i], which is a contradiction. We conclude from this reasoning that if p is delayed at statement 5[i], and if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p.

From the discussion above, we have the following property and lemma.

Property 2: If distinct processes p and q have executed statement 7[i] and have $nd = i$, then the value of p's private variable $y.rnd$ differs from that of q's. □

```
/* all variable declarations are as defined in Fig. 4 except as noted here */
type
    Ytype = record free: boolean; rnd: 0..N − 1 end        /* stored in one word */
shared variables
    X: array[1..T] of 0..N − 1;
    Rnd: array[1..T][0..N − 1] of boolean init false;
    Obstacle: array[0..N − 1] of 0..T init 0;
    Acquired: array[1..T] of boolean init false

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:      X[nd], dir := p, S;
3:      y := Y[nd];
        if ¬y.free then dir := R
        else
4:          Y[nd] := (false, 0);
5:          Obstacle[p] := nd;
6:          if X[nd] ≠ p ∨
7:                  Acquired[nd] then
                dir := L
            else
8:              Rnd[nd][y.rnd] := true;
9:              if Reset[nd] ≠ y then
10:                 Rnd[nd][y.rnd], dir := false, L
        fi fi fi;
11:     path[lvl] := (nd, dir);
        if dir ≠ S then
            lvl, nd := lvl + 1, 2 · nd;
            if dir = R then nd := nd + 1 fi
        fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
12:     Acquired[nd] := true;
        for j := lvl downto 0 do
13:         ENTRY3(path[j].nd, path[j].dir)
        od;
14:     ENTRY2(0)
    else     /* didn't get a name */
15:     ENTRY_N(p);
16:     ENTRY2(1)
    fi;
17: Critical Section;
18: Obstacle[p] := 0;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
        if path[j].dir ≠ R then
19:         n := path[j].nd;
20:         Y[n] := (false, 0);
21:         X[n] := p;
22:         y := Reset[n];
23:         Reset[n] := (false, y.rnd);
24:         if (j = lvl ∨
                    ¬Rnd[n][y.rnd]) ∧
25:                 Obstacle[y.rnd] ≠ n then
26:             Reset[n] := (true, y.rnd ⊕ 1);
27:             Y[n] := (true, y.rnd ⊕ 1)
            fi;
28:         if j = lvl then
                Rnd[y.rnd] := false
            fi
        fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
29:     EXIT2(0);
        for j := 0 to lvl do
30:         EXIT3(path[j].nd, path[j].dir)
        od;
31:     Acquired[nd] := false
    else
32:     EXIT2(1);
33:     EXIT_N(p)
    fi
od
```

Fig. 5. Algorithm B: adaptive algorithm with $\Theta(N^2)$ space complexity.

Lemma 2: Algorithm B is a correct, starvation-free mutual exclusion algorithm with $O(min(k, \log N))$ time complexity and $\Theta(N^2)$ space complexity. □

The $\Theta(N^2)$ space complexity of Algorithm B is due to the *Rnd* array. We now show that this $\Theta(N^2)$ array can be replaced by a $\Theta(N)$ linked list.

Algorithm L. In Algorithm L (for linear), which is depicted in Fig. 6, a common pool of round numbers ranging over $\{1, \ldots, U\}$ is used for all splitters in the renaming tree. As we shall see, $O(N)$ round numbers suffice. In Algorithm B, our key requirement for round numbers was that they not be reused "prematurely." With a common pool of round numbers, a process should not choose r as the next round number for some splitter if there is a process *anywhere* in the renaming tree that "thinks" that r is the current round number of some splitter.

Fortunately, since each process selects new round numbers within its critical section, it is fairly easy to ensure this requirement. All that is needed are a few extra data structures that track which round numbers are currently in use. These data structures replace the *Obstacle* array of Algorithm B. The main new data structure is a queue *Free* of round numbers. In addition, there is a new shared array *Inuse*, and a new shared variable *Check*. We assume that the *Free* queue can be manipulated by the usual *Enqueue* and *Dequeue* operations, and also by an operation *MoveToTail*(*Free*, i: 1..U), which moves i to the end of *Free*, if it is in *Free*. If *Free* is implemented as a doubly-linked list, then these operations can be performed in constant time. We stress that *Free* is accessed *only* within critical sections, so it is really a *sequential* data structure.

When comparing Algorithms B and L, the only difference before the critical section is statement 5[i]: instead of updating *Obstacle*[p], process p now marks the round number r it just read from $Y[i]$ as being "in use" by assigning *Inuse*[p] := r. The only other differences are in the code after the critical section (statements 18-33 in Fig. 6). Statements 24-27 are executed to ensure that no round number currently "in use" can propagate to the head of the *Free* queue. In particular, if a process p is delayed after having obtained r as the current round number for some splitter, then while it is delayed, r will be moved to the end of the *Free* queue by every N^{th} critical section execution. With $U = T + 2N$ round numbers, this is sufficient to prevent r from reaching the head of the queue while p is delayed. ($T + 2N$ round numbers are needed because the calls to *Dequeue* and *MoveToTail* can cause a round number to migrate toward the head of the *Free* queue by two positions per critical section execution.) Statement 28[i] enqueues the current round number for splitter i onto the *Free* queue. (Note that there may be other processes within the renaming tree that "think" that the just-enqueued round number is the current round number for splitter i; this is why we need a mechanism to prevent round numbers from prematurely reaching the head of the queue.) Statement 29[i] simply dequeues a new round number from *Free*. The rest of the algorithm is the same as before.

The space complexity of Algorithm L is clearly $\Theta(N)$, if we ignore the space required to implement the `ENTRY` and `EXIT` routines. (Each process has a $\Theta(\log N)$ *path* array. These arrays are actually unneeded, as simple calculations can be

```
/* all variable declarations are as defined in Fig. 5 except as noted here */
const  U = T + 2N                /* number of possible round numbers = O(N) */
type   Ytype = record free: boolean; rnd: 0..U end  /* stored in one word */

shared variables
  Y, Reset: array[1..T] of Ytype;
  Rnd: array[1..U] of boolean init false;
  Free: queue of integers;
  Inuse array[0..N − 1] of 0..U init 0;
  Check: 0..N − 1 init 0

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:    X[nd], dir := p, S;
3:    y := Y[nd];
      if ¬y.free then dir := R
      else
4:      Y[nd] := (false, 0);
5:      Inuse[p] := y.rnd;
6:      if X[nd] ≠ p ∨
7:           Acquired[nd] then
          dir := L
        else
8:        Rnd[y.rnd] := true;
9:        if Reset[nd] ≠ y then
10:         Rnd[y.rnd], dir := false, L
      fi fi fi;
11:   path[lvl] := (nd, dir);
      if dir ≠ S then
        lvl, nd := lvl + 1, 2 · nd;
        if dir = R then nd := nd + 1 fi
      fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
12:   Acquired[nd] := true;
      for j := lvl downto 0 do
13:     ENTRY3(path[j].nd, path[j].dir)
      od;
14:   ENTRY2(0)
    else     /* didn't get a name */
15:   ENTRY_N(p);
16:   ENTRY2(1)
    fi;

initially
  (∀i : 1 ≤ i ≤ T :: Y[i] = (true, i) ∧
     Reset[i] = (true, i)) ∧
     (Free = (T + 1) → ··· → U)

private variables
  ptr: 0..N − 1; nxtrd: 1..U; usdrd: 0..U

17: Critical Section;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
      if path[j].dir ≠ R then
18:     n := path[j].nd;
19:     Y[n] := (false, 0);
20:     X[n] := p;
21:     y := Reset[n];
22:     Reset[n] := (false, y.rnd);
23:     if j = lvl ∨ ¬Rnd[y.rnd] then
24:       ptr := Check;
25:       usdrd := Inuse[ptr];
26:       if usdrd ≠ 0 then
            MoveToTail(Free, usdrd)
          fi;
27:       Check := ptr ⊕ 1;
28:       Enqueue(Free, y.rnd);
29:       nxtrd := Dequeue(Free);
30:       Reset[n] := (true, nxtrd);
31:       Y[n] := (true, nxtrd)
        fi;
        if j = lvl then
32:       Rnd[y.rnd] := false;
33:       Inuse[p] := 0
      fi fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
34:   EXIT2(0);
      for j := 0 to lvl do
35:     EXIT3(path[j].nd, path[j].dir)
      od;
36:   Acquired[nd] := false
    else
37:   EXIT2(1);
38:   EXIT_N(p)
    fi
od
```

Fig. 6. Algorithm L: adaptive algorithm with $\Theta(N)$ space complexity.

used to determine the parent and children of a splitter.) If the `ENTRY/EXIT` routines are implemented using Yang and Anderson's arbitration-tree algorithm [14], then the overall space complexity is actually $\Theta(N \log N)$. This is because in Yang and Anderson's algorithm, each process needs a distinct spin location for each level of the arbitration tree. However, as we will show in the full paper, it is quite straightforward to modify the arbitration-tree algorithm so that each process uses the same spin location at each level of the tree. This modified algorithm has $\Theta(N)$ space complexity. We conclude by stating our main theorem.

Theorem 1. *N-process mutual exclusion can be implemented under read/write atomicity with time complexity $O(min(k, \log N))$ and space complexity $\Theta(N)$.* □

Acknowledgement: Gary Peterson recently conjectured to us that adaptivity under the remote-memory-references time measure must necessitate $\Omega(N^2)$ space complexity. His conjecture led us to develop Algorithm L.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 91–103, 1999.
2. J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning (full version of this paper). At `http://www.cs.unc.edu/~anderson/papers.html`.
3. J. Anderson and Y.-J. Kim. Fast *and* scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, September 1999. Full version to appear in *Distributed Computing*.
4. T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Sys.*, 1(1):6–16, 1990.
5. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. To appear in *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*.
6. M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
7. E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
8. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.
9. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Sys.*, 5(1):1–11, 1987.
10. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Sys.*, 9(1):21–65, 1991.
11. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.
12. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.
13. E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 159–168. 1992.
14. J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

Bounds for Mutual Exclusion with only Processor Consistency

Lisa Higham* and Jalal Kawash**

Department of Computer Science, The University of Calgary, Canada
{higham|kawash}@cpsc.ucalgary.ca

Abstract. Most weak memory consistency models are incapable of supporting a solution to mutual exclusion using only read and write operations to shared variables. Processor Consistency–Goodman's version (PC-G) is an exception. Ahamad et al.[1] showed that Peterson's mutual exclusion algorithm is correct for PC-G, but Lamport's bakery algorithm is not. In this paper, we derive a lower bound on the number and type (single- or multi-writer) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G. We show that any such solution for n processes must use at least one multi-writer and n single-writers. This lower bound is tight when $n = 2$, and is tight when $n > 2$ for solutions that do not provide fairness. We show that Burns' algorithm is an unfair solution for mutual exclusion in PC-G that achieves our bound. However, five other known algorithms that use the same number and type of variables do not guarantee mutual exclusion when the memory consistency model is only PC-G, as opposed to the Sequential Consistency model for which they were designed. A corollary of this investigation is that, in contrast to Sequential Consistency, multi-writers cannot be implemented from single-writers in PC-G.

1 Introduction

The Mutual Exclusion Problem is the most famous and well-studied problem in concurrency. Following Silberschatz et al.[14], we refer to this problem as the Critical Section Problem (CSP) to distinguish the problem from the Mutual Exclusion Property. In CSP, a set of processes coordinate to share a resource, while ensuring that no two access the resource concurrently. CSP solutions for memories that satisfy Sequential Consistency (SC) have been known since the 1960s; Raynal [13] provides an extensive survey. In fact, as shown by Lamport [10], even single-reader single-writer bits suffice to solve the critical section problem, as long as accesses to these objects are guaranteed to be SC.

Most weak memory consistency models, however, are incapable of supporting a solution to CSP using only read and write operations on shared variables [7]. Mutual exclusion on weak memory consistency models such as Java, Coherence, Pipelined-RAM, Total and Partial Store Ordering, Causal Memory, and several variants of Processor Consistency requires the use of expensive built-in synchronization primitives

* Supported in part by the Natural Sciences and Engineering Research Council of Canada grant OGP0041900.

** Supported in part by a Natural Sciences and Engineering Research Council of Canada doctoral scholarship and an Izaak Walton Killam Memorial scholarship.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 44–58, 2000.

such as locks, compare-and-swap, fetch-and-add and others [7]. A notable exception is Processor Consistency (abbreviated PC-G)[1] as proposed by Goodman and formalized by Ahamad et al.[1]. Though weaker than SC, this variant of processor consistency guarantees that processes have just enough agreement about the current state of shared memory to support a solution using only reads and writes of shared variables.

Ahamad et al. have shown that Peterson's mutual exclusion algorithm [12] is correct for PC-G, but that Lamport's bakery algorithm [8] fails for PC-G [1]. We are thus motivated to determine what is necessary and sufficient to solve CSP with only PC-G memory using only reads and writes to shared variables. For example, Peterson's algorithm makes use of multi-writers, variables that can be written my more than one process, while Lamport's bakery algorithm [8] uses only single-writers, variables that can be written by exactly one process. Are multi-writers essential?

In this paper, we derive tight bounds on the number and type (single- or multi-writer) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G. Specifically, any PC-G solution for n processes must use at least one multi-writer and n single-writers. We prove that Burns' algorithm [3], which uses one multi-writer and n single-writers, is an unfair solution for mutual exclusion in PC-G. Thus our bound is tight for unfair solutions to CSP. Since Peterson's 2-processor algorithm is fair and correct for PC-G, our bound is tight even for fair solutions when $n = 2$.

We further investigate properties that a solution, using one multi-writer and n single-writers, must satisfy in order to be correct for PC-G. Using these properties, we establish that five algorithms [13], Dekker's, Dijkstra's, Knuth's, De Bruijn's, Eisenberg and MacGuire's, do not guarantee mutual exclusion under only PC-G memory consistency. All of these have been developed for SC [9], and all use one multi-writer and n single-writers. However, most of these algorithms are fair solutions for CSP in SC. The only fair solution we have found for PC-G is Peterson's which uses $n-1$ multi-writers and n single-writers.

Since multi-writers are required to solve CSP in PC-G, a corollary of our investigation is that, in contrast to SC, multi-writers cannot be implemented from single-writers in PC-G.

Section 2 includes the definitions needed for this paper. Section 3 provides a template for our impossibility proofs, which is used to establish our lower bounds in Section 4. The major results in Section 4 have been automatically verified using the SPIN model checker [6].

2 Definitions

2.1 Multiprocess Systems and Memory Consistency Models

A multiprocess system can be modeled as a collection of processes operating on a collection of shared data objects. For this paper, the shared data objects are variables supporting only read and write operations, where $r(x)v$ and $w(x)v$ denote, respectively, a read operation of variable x returning v and a write operation to x of value v. An

[1] Several variants of Processor Consistency exist. The one referred to in this paper is due to Ahamad et al.'s[1] interpretation of Goodman's original work [5].

operation can be decomposed into invocation (performed by processes) and response (returned by variables) components.

It suffices to model a *process* as a sequence of read and write invocations, and a *multiprocess system* as a collection of processes together with the shared variables. Henceforth, we denote a multiprocess system by the pair (P,J) where P is a set of processes and J is a set of variables. A *process computation* is the sequence of reads and writes obtained by augmenting each read invocation in the process with its matching response. A *(multiprocess) system computation* is a collection of process computations, one for each process in the collection.

Let O be all the (read and write) operations in a computation of a system (P,J). Then, $O|p$ denotes all the operations that are in the process computation of process $p \in P$; $O|x$ denotes all the operations that are applied to variable $x \in J$, and $O|w$ denotes all the write operations. These notations are also combined to select those operations satisfying several restrictions at once. For example, $O|w|x|p$ is the set of all write operations by process p to variable x.

A sequence of read and write operations to variable x is *valid* if and only if each read in the sequence returns the value of the most recently preceding write. Given any collection of read and write operations O on a set of variables J, a *linearization of* O is a (strict) linear order[2] $(O, \xrightarrow{L})$ such that for each variable x in J, the subsequence $(O|x, \xrightarrow{L})$ of $(O, \xrightarrow{L})$ is valid. A linear order $(O, \xrightarrow{L})$ is also represented in this paper as the sequence $L = \langle o_1, o_2, \ldots \rangle$ where o_i precedes o_j in L if and only if $(o_i, o_j) \in (O, \xrightarrow{L})$.

Let O be a set of operations in a computation of a system (P,J). Define the *program order*, denoted $(O, \xrightarrow{prog})$, by $o_1 \xrightarrow{prog} o_2$ if and only if o_2 follows o_1 in the computation of p.

A *(memory) consistency model* is a set of constraints on system computations. A system (P,J) *satisfies* memory consistency D if every computation that can arise from it meets all the constraints in D.

We allow a *(sequential) program* to be any computer code containing control structures, local variables together with any computable operations on them, and reads and writes of global variables. (The global variables are so restricted because we are interested in what can be achieved with just reads and writes of variables for communication between processes.) Then a *(multiprocess) algorithm* is just a collection of such programs where all global variables are shared. (Often, but not necessarily, each program in the collection is the same.) The algorithm together with the memory consistency model can produce some set of system computations, where each program gives rise to a process as defined above.

Four memory consistency models are referred to in this paper. SC [9] is a strong memory consistency model that arises (for example) when the memory shared between processes is single-ported and thus all reads and writes to the memory are serialized. Thus, SC guarantees that the computation of the system is the result of some interleav-

[2] A *(strict) partial order* (simply, partial order) is an anti-reflexive, transitive relation. Denote a partial order by a pair (S,R). The notation $s_1 R s_2$ means $(s_1, s_2) \in R$. A *(strict) linear order* is a partial order (S,R) such that $\forall x, y \in S\ x \neq y$, either xRy or yRx.

ing of the processes. This model is typically assumed by algorithm designers, and a challenge for system designers is to build SC systems while exploiting the efficiencies of distributed shared memory.

Definition 1. *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* SC *if there is a linearization $(O, \xrightarrow{L})$ such that $(O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{L})$.*

If the single-ported globally shared memory is partitioned into several components each of which has separate single-ported access, then SC of the whole system is lost, but is maintained for each component individually. In the extreme, when each shared variable has its own access channel, the memory consistency model is called Coherence [4]. In a Coherent memory model, reads and writes of different variables can happen in time in the opposite order from program order. However, such a system still ensures that *for each shared variable* the outcome of the computation results from some interleaving of the process reads and writes to that variable.

Definition 2. *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* Coherence *if for each variable $x \in O$ there is a linearization $(O|x, \xrightarrow{L_x})$ such that $(O|x, \xrightarrow{prog}) \subseteq (O|x, \xrightarrow{L_x})$.*

Now consider a message-passing network of processes each of which stores a local copy of the shared memory. If the message channels are FIFO and form a complete network, reads are implemented by consulting the local memory, and writes are broadcast to every other process, then the memory consistency model that arises is the Pipelined-Random Access Machine (P-RAM)[11].

Definition 3. *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* P-RAM *if for each process $p \in P$ there is a linearization $(O|p \cup O|w, \xrightarrow{L_p})$ such that $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$.*

For a memory model to meet PC-G [1], there must be a set of linearizations that simultaneously satisfy both Coherence and P-RAM.

Definition 4. *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* PC-G *if for each process $p \in P$ there is a linearization $(O|p \cup O|w, \xrightarrow{L_p})$ such that*

1. $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$, *and*
2. $\forall q \in P$ *and* $\forall x \in J$, $(O|w|x, \xrightarrow{L_p}) = (O|w|x, \xrightarrow{L_q})$.

2.2 Critical Section Problem

We denote a CSP problem by CSP(n) where n is the number of processes in the system. Each process has the following structure:

repeat
<*remainder*>
<*entry*>
<*critical section*>
<*exit*>
until *false*

A solution to CSP(n), $n \geq 2$, must satisfy the following two properties[3]

- **Mutual Exclusion:** At any time there is at most one process in its <*critical section*>
- **Progress:** If at least one process is in <*entry*>, then eventually one will be in <*critical section*>.

CSP typically requires some notion of fairness as well. One possible fairness property is:

- **Fairness:** If a process p is in <*entry*>, then p will eventually be in <*critical section*>.

It is possible to consider stronger notions of fairness. We will see, however, that our impossibility and lower bound results apply even to unfair solutions of CSP, and therefore we make no fairness requirement in our definition.

Notice that time is used in the definition of CSP. However, we make no assumptions about agreement in rate or value between the clocks that are part of the multiprocess system, and, therefore, the memory consistency models considered here have been defined without reference to time. So we need to clarify how a system without a consistent notion of time can be tested for a property involving time. The multiprocess system exists in some environment that has its own meaningful time which we call *real time*. In the case of CSP, which is controlling access to some resource, real time can be taken to be the local clock time of that resource. For a system to satisfy the Mutual Exclusion property it is required that there is no computation of that system for which there are two or more processes in their critical sections at the same real time.

Let A and D be an algorithm and a memory consistency model, respectively. Then, *A solves CSP for D* if for every system S that satisfies D, every computation of S satisfies Mutual Exclusion and Progress.

3 Template for Impossibility and Lower Bound Proofs

We will use the partial computations 1, 2, and 3 defined below. First, assume for the sake of contradiction that there exists an algorithm A that solves CSP(n) for a given memory consistency model, D, for $n \geq 2$. This solution must work when exactly two processes, say p and q, are participating and the rest engaging in <*remainder*>. If A runs with p in <*entry*> while q stays in <*remainder*>, then by the Progress property, p must enter its <*critical section*> producing a partial computation of the form of Computation 1, where λ denotes the empty sequence and o_i^p denotes the i^{th} operation of p and k is a finite natural number.

[3] Other forms of defining solution properties are possible as is given by Attiya et al.[2].

Computation 1 $\begin{cases} p : o_1^p, o_2^p, \ldots, o_k^p \ \ (p \textit{ is in its } < \textit{critical section} >) \\ q : \lambda \end{cases}$

Similarly, if A runs with only q's participation, Progress guarantees that Computation 2 exists.

Computation 2 $\begin{cases} p : \lambda \\ q : o_1^q, o_2^q, \ldots, o_l^q \ \ (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

Now, consider Computation 3 where both p and q are participating, but both are in their $<$*critical section*$>$. By assumption, both computations 1 and 2 satisfy D. If we can show that Computation 3 also satisfies memory consistency condition D, the desired contradiction is achieved, since Mutual Exclusion is violated by Computation 3, but it is a possible outcome of algorithm A. This would imply that there is no algorithm that solves CSP(n) for memory consistency model D.

Computation 3 $\begin{cases} p : o_1^p, o_2^p, \ldots, o_k^p \ \ (p \textit{ is in its } < \textit{critical section} >) \\ q : o_1^q, o_2^q, \ldots, o_l^q \ \ (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

None of the arguments in the following theorems depends on the Fairness property, so the impossibilities apply to unfair solutions as well. Furthermore, none of these argument depends on the size of variables. So, these results apply to unbounded variables as well.

4 Bounds on CSP for PC-G

Ahamad et al.[1] proved that Peterson's algorithm [12], which was originally developed for SC systems, solves CSP(2) for PC-G. Given algorithm A_2 that solves CSP(2) for PC-G, an algorithm A_n that solves CSP(n) for PC-G, where $n \geq 2$, can be constructed from A_2 by building a tournament tree. Processes are partitioned into sets of size two each. For each set, A_2 is used to select a "winner". The winners are again partitioned into sets of size two, and A_2 can be used in this manner repeatedly until only one winner remains. Thus we conclude that there is an algorithm that solves CSP(n) for PC-G.

This section further investigates bounds and restrictions on these PC-G solutions.

4.1 Type of Variables

A *multi-writer* variable (simply, multi-writer) can be updated by any number of processes in the system, while a *single-writer* variable (simply, single-writer) can be updated by exactly one designated process.

We show that the use of multi-writers is crucial to solve CSP on PC-G. First we need the following lemma.

Lemma 1. *In a system* (P, J) *where J consists entirely of single-writers, PC-G is equivalent to P-RAM.*

Proof: Obviously, PC-G is at least as strong as P-RAM. We show that without the use of multi-writer variables, P-RAM is at least as strong as PC-G. Let $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ be linearizations for p and $q \in P$ that are guaranteed by P-RAM. Since, for any variable $x \in J$, there is only one process, say s, that writes to x, and both $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ have all these writes to x in the program order of s, the order of the writes to x in $(O|p \cup O|w, \xrightarrow{L_p})$ is the same as the order of the writes to x in $(O|q \cup O|w, \xrightarrow{L_q})$. Therefore, the definition of PC-G (Definition 4) is satisfied. ■

CSP, however, is impossible for P-RAM:

Theorem 1. *There does not exist an algorithm that solves CSP(n) for P-RAM, even if* $n = 2$.

Proof: Assume that there is an algorithm A that solves CSP(n) for P-RAM. Then computations 1 and 2 exist. Define the following sequences for p and q, respectively, for Computation 3.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle (o_1^p, \ldots, o_k^p), (o_1^q, \ldots, o_l^q)|w \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle (o_1^q, \ldots, o_l^q), (o_1^p, \ldots, o_k^p)|w \rangle$$

Clearly, each preserves $\xrightarrow{prog}$ as required by the definition of P-RAM. Also, each is a linearization because the first part (for instance, $(o_1^p, \ldots, o_k^p)$) corresponds to a possible computation, and the second part (for instance, $(o_1^q, \ldots, o_l^q)|w$) contains only writes. Thus, Computation 3 is P-RAM. Therefore, our assumption must have been in error and A does not exist. ■

Theorem 2. *There does not exist an algorithm that uses only single-writers and solves CSP(n) for PC-G, even if* $n = 2$.

Proof: This follows immediately from Lemma 1 and Theorem 1. ■

Ahamad et al.[1] also prove that Lamport's Bakery algorithm [8], which uses only single-writers, is incorrect for PC-G. The consequence of Theorem 2 is that any CSP solution for PC-G must use at least one multi-writer.

Vitanyi and Awerbuch [15] showed that multi-writer variables can be constructed in a waitfree manner from single-writer variables. In PC-G, there is no (even non-waitfree) construction of multi-writer variables from single-writer variables.

Corollary 1. *Multi-writers cannot be implemented from single-writers in PC-G memory systems.*

Proof: Peterson's algorithm solves CSP for PC-G using multi-writers, and there is no solution with only single writers by Theorem 2. Hence, multi-writers cannot be constructed from single-writers in PC-G. ■

4.2 Number of Variables

After showing that at least one multi-writer is required by a CSP solution for PC-G, a natural question is what is the minimum number of variables needed to solve CSP(n) for PC-G?

Theorem 3. *There does not exist an algorithm that uses fewer than n single-writers and one multi-writer and solves CSP(n) for PC-G, for any* $n \geq 2$.

Proof: Assume that there is an algorithm A that uses fewer than n single-writers and one multi-writer and solves CSP(n) for PC-G. Since there are n processes, the pigeonhole principle ensures that there is at least one process, say p, that does not write to any single-writer variable. Computations 1 and 2 must exist. We show that Computation 3 satisfies PC-G.

Let o_i^q be q's first write to the multi-writer. The following are the required PC-G linearizations for p and q.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle o_1^p, \cdots, o_k^p, (o_1^q, \cdots, o_l^q)|w \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle o_1^q, \cdots, o_{i-1}^q, (o_1^p, \cdots, o_k^p)|w, o_i^q, \cdots, o_l^q \rangle$$

Both sequences maintain program order. Moreover, p's sequence is valid because it consists of Computation 1 followed by only writes by q. Also, q's sequence is valid because the segment $o_1^q, \cdots, o_{i-1}^q$ does not contain any writes to the multi-writer. Since p does not write to the single-writer, the segment $(o_1^p, \cdots, o_k^p)|w$ contains only writes to the multi-writer. The segment $o_i^q, \cdots, o_l^q$ starts with a write to the multi-writer over-writing any changes the segment $(o_1^p, \cdots, o_k^p)|w$ caused. Therefore both are linearizations.

Also, each linearization lists p's writes to the multi-writer followed by q's. Since only q writes to any single-writers, the two linearizations also agree on the order of this variable. So, both linearizations agree on the order of writes for each variable (Condition 2 of Definition 4). ■

When $n = 2$, the bound of theorem 3 is tight, even if all variables are allowed to be multi-writers.

Theorem 4. *Two variables are insufficient to solve CSP(2) for PC-G.*

Proof: Assume that there is an algorithm A that uses exactly 2 variables, say x and y, (even multi-writers) and solves CSP(2) for PC-G. Then, computations 1 and 2 exist. We show that Computation 3 satisfies PC-G.

Partition p's computation of Computation 3 into subsequences $S_0^p, S_1^p, \ldots, S_u^p$ where each subsequence S_i^p is defined by:

1. S_0^p contains all operations from o_1^p up to but not including the first write by p, labeled $o_{\alpha_1}^p$.
2. S_i^p, $i \geq 1$, contains all operations from $o_{\alpha_i}^p$ up to but not including the first write, labeled $o_{\alpha_{i+1}}^p$, such that $o_{\alpha_i}^p$ and $o_{\alpha_{i+1}}^p$ are applied to different variables.

Partition q's computation of Computation 3 into subsequences $S_0^q, S_1^p, \ldots, S_r^q$ similarly.

The subsequence S_0^p is either empty or consists entirely of reads returning initial values. Each subsequence S_i^p ($i \geq 1$) starts with a write and all the writes in S_i^p are applied to the same variable. If the writes in S_i^p are applied to x, S_i^p is called x-gender; otherwise, it is called y-gender. Note that S_i^p (S_i^q) alternate in gender.

To show that Computation 3 satisfies PC-G, we consider two cases (the other two cases are symmetric).

S_1^p is an x-gender but S_1^q is a y-gender: Define $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ as follows.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle S_0^p,\ (S_0^q)|w,\ S_1^p,\ (S_1^q)|w,\ S_2^p,\ \cdots,\ (S_i^q)|w,\ S_{i+1}^p,\ \cdots \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle S_0^q,\ (S_0^p)|w,\ S_1^q,\ (S_1^p)|w,\ S_2^q,\ \cdots,\ (S_i^p)|w,\ S_{i+1}^q,\ \cdots \rangle$$

Clearly, $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ maintain program order. They are also valid because, for each $i \geq 1$, S_i^p (respectively, S_i^q) is of the same gender as S_{i+1}^q (respectively, S_{i+1}^p). Since S_i^q and S_{i+1}^p are of the same gender, adding $(S_i^q)|w$ immediately before S_{i+1}^p does not affect p's computation because S_{i+1}^p starts with a write that obliterates the changes caused by $(S_i^q)|w$; similarly for S_i^p and S_{i+1}^q.

The order on the writes to x in p's linearization is:

$$(S_1^p)|w,\ (S_2^q)|w,\ \cdots,\ (S_i^p)|w,\ (S_{i+1}^q)|w,\ \cdots, \text{ (where } i \text{ is odd)}$$

which is the same order maintained by q's linearization. The same applies to y. Therefore, Condition 2 of Definition 4 is also satisfied.

S_1^p and S_1^q are both x-gender: Define $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ as follows.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle (S_0^q)|w,\ S_0^p,\ (S_1^q)|w,\ S_1^p,\ \cdots,\ (S_i^q)|w,\ S_i^p,\ \cdots \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle S_0^q,\ S_1^q,\ (S_0^p)|w,\ S_2^q,\ (S_1^p)|w,\ S_3^q,\ \cdots,\ (S_i^p)|w,\ S_{i+2}^q,\ \cdots \rangle$$

Similar analysis to the previous case shows that these are PC-G linearizations.

Thus, in all cases, Computation 3 is PC-G, and our assumption must have been in error. ■

Since at least one multi-writer is necessary to solve CSP for PC-G, and since two multi-writers are insufficient to solve CSP(2) for PC-G, and since Peterson's Algorithm for CSP(2) uses exactly two single-writers and one multi-writer, we conclude the following.

Corollary 2. *Two single-writers and one multi-writer are the necessary and sufficient number and type of variables required to solve CSP(2) for PC-G.*

Algorithm	Year	$\|P\|$	Variables	`flag` Values	Fairness Delay
Dekker's	1965	$n = 2$	$n+1$	2	∞
Dijkstra's	1965	$n \geq 2$	$n+1$	3	∞
Knuth's	1966	$n \geq 2$	$n+1$	3	$2^{n-1}-1$
De Bruijn's	1967	$n \geq 2$	$n+1$	3	$(n^2-n)/2$
Eisenberg and MacGuire's	1972	$n \geq 2$	$n+1$	3	$n-1$
Burns'	1981	$n \geq 2$	$n+1$	2	∞
Peterson's	1981	$n \geq 2$	$2n-1$	2	$(n^2-n)/2$

Fig. 1. Well known solutions to CSP for Sequential Consistency

4.3 The General Case

By theorems 2 and 3, an algorithm that solves CSP(n) for PC-G must use at least n single-writers and one multi-writer. Most algorithms that solve CSP(n) for SC use exactly this number and type of variables. In particular, all the algorithms discussed in this section (except Peterson's which uses n single-writers and $n-1$ multi-writers) use the same number of variables: one multi-writer (`turn`) and n single-writers. Furthermore, each process writes and reads `turn`, and each process i is associated with the single-writer `flag[`i`]`. Every process $j \neq i$ reads `flag[`i`]`. These algorithms are quoted in Appendix A and listed in Figure 1, which characterizes each algorithm by four attributes: number of processes $|P| = n$, number of variables, number of values that a `flag` variable can be assigned, and fairness delay. This fairness delay is the maximum total number of times other processes can enter their critical sections before a certain process gets the opportunity to enter its critical section. When the there is no upper bound on the fairness delay (∞), the algorithm is prone to starvation, and is thus unfair.

Although this number of variables is a necessary requirement for a PC-G solution, we show next that most of these algorithms do not solve CSP(n) for PC-G. First, we provide some *rules-of-thumb* that allows us to nail down certain properties of correct solutions for PC-G. Then, these rules are used to show that Dekker's, Dijkstra's, Knuth's, De Bruijn's, and Eisenberg and MacGuire's fail to solve CSP(n) for PC-G.

Lemma 2. *Any algorithm that uses exactly n single-writers and one multi-writer and solves CSP(n) for PC-G must satisfy each of the following properties:*

1. *Each process writes one single-writer at least once in <entry>.*
2. *Each process must write the multi-writer at least once in <entry>, and this write cannot be the last operation in <entry>.*
3. *Each process must read every other single-writer in <entry>.*

Proof: We follow the proof template given in Section 3.

1. Assume it is not the case; then there is at least one process, say p, that does not write to any single-writer. The linearizations used in Theorem 3 apply.

2. Assume that a process p either does not write the multi-writer in <*entry*> or does write the multi-writer exactly once and this write operation is o_k^p. Under this assumption, Computation 3 satisfies PC-G as shown by the following linearizations.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle o_1^p, \ldots, o_{k-1}^p, (o_1^q, \ldots, o_l^q)|w, o_k^p \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle o_1^q, \ldots, o_l^q, (o_1^p, \ldots, o_k^p)|w \rangle$$

   Both maintain program order and are valid. They also maintain the same order on the writes to the multi-writer, which is simply q's writes then o_k^p. Note that this case is equivalent to the case where multi-writer is written in the <*critical section*> rather than in <*entry*>.
3. Assume, for the sake of contradiction, that there is a process, q, that does not read some single-writer of another process p. The linearizations of Theorem 3 apply.

■

Corollary 3. *The following CSP algorithms do not solve CSP(n) for PC-G, even if* $n = 2$*:*

1. *Dijkstra's Algorithm*
2. *Dekker's Algorithm*
3. *De Bruijn's Algorithm*
4. *Knuth's Algorithm*
5. *Eisenberg and MacGuire's Algorithm*

Proof: First, note that all these algorithms (reproduced in Appendix A) use n single-writers and one multi-writer.

In Dijkstra's Algorithm, if the multi-writer `turn` is initially p, p enters its <*critical section*> without writing to the multi-writer. In Dekker's and Bruijn's algorithms, the multi-writer is only written in <*exit*>. In Knuth's, and in Eisenberg and MacGuire's algorithms, the multi-writer is only written as the last step in <*entry*>. By Lemma 2(2), all of these algorithms are incorrect for PC-G.

■

Theorem 5. *Burns' Algorithm is an unfair CSP(n) solution for PC-G.*

Proof: *Mutual Exclusion:* Assume for the sake of contradiction that there exists some PC-G computation of Burns' Algorithm where two processes, say i and j, execute in their <*critical section*> concurrently. Then, i (respectively, j) must read `flag[`j`]` (respectively, `flag[`i`]`) to be *false* at line 11 before entering its <*critical section*> as shown by the following computation.

Computation 4 $\begin{cases} i: \ldots r(\texttt{flag}[j])\mathit{false} < \mathit{critical\ section} > \\ j: \ldots r(\texttt{flag}[i])\mathit{false} < \mathit{critical\ section} > \end{cases}$

```
        flag[0 .. n−1] in {true, false}
        turn in {0,···,n−1}

        <entry>
1       flag[i] ← true
2       turn ← i
3       repeat
4             while (turn ≠ i) do
5                   flag[i] ← false
6                   if (∀ j ≠ i, not flag[j]) then
7                     flag[i] ← true
8                     turn ← i
9                   end-if
10            end-while
11      until (∀ j ≠ i, not flag[j])

        <critical section>

        <exit>
12      flag[i] ← false
```

Processes have unique identifiers from the set $\{0,\cdots,n-1\}$, where n is the total number of processes. The algorithm is given by specifying the <*entry*> and <*exit*> sections of process i, $i \in \{0,\cdots,n-1\}$.

Fig. 2. Burns' CSP unfair solution

Note that when a process, say i, executes a $w(\texttt{flag}[i])$*true*, the next operation it executes is a $w(\texttt{turn})i$. Let $w(\texttt{turn})i$ be the last write operation to `turn` that i executes before entering its <*critical section*> (This write could be performed at line 2 or 8.) Similarly, let $w(\texttt{turn})j$ be the last write to `turn` that j did before entering its <*critical section*>.

Since Computation 4 satisfies PC-G, the two linearizations $(O|i \cup O|w, \xrightarrow{L_i})$ and $(O|j \cup O|w, \xrightarrow{L_j})$ must exist such that both agree on the order of writes to `turn`. Without loss of generality, suppose $w(\texttt{turn})i$ precedes $w(\texttt{turn})j$ in both linearizations. Since $w(\texttt{turn})j \xrightarrow{L_j} r(\texttt{flag}[i])$*false* (by program order), $w(\texttt{turn})i \xrightarrow{L_j} r(\texttt{flag}[i])$*false*. There must be some write $w(\texttt{flag}[i])$*true*, such that this write is the last write by i that precedes $w(\texttt{turn})i$ in j's view. Since $w(\texttt{turn})i$ is the last write by i before it enters its <*critical section*>, $w(\texttt{flag}[i])$*true* must be the last write to `flag[`i`]` before i enters its <*critical section*>. By transitivity, this write is the most recent write to `flag[`i`]` that precedes $r(\texttt{flag}[i])$*false* in j's view, contradicting the validity of $(O|j \cup O|w, \xrightarrow{L_j})$. Therefore, Burns' algorithm satisfies Mutual Exclusion for PC-G.

Progress: If only one process is participating, then it will enter the <*critical section*>. So assume m processes, $2 \leq m \leq n$, are participating in a computation of Burns' Algorithm such that none of them is able to progress to <*critical section*>. We show this is impossible. By PC-G, all processes must agree of the order of the writes to `turn`, and eventually $m-1$ of them will see `turn` different from their own identifiers; therefore, all $m-1$ processes enter the body of the while loop. At least one process will fail the test on line 4 skipping the while loop. This is because of the total order on the writes to `turn` that all processes agree on. Since there is at least one process, say j, that

does not engage in the while loop, we must have the following, where $i \neq j$:

$$w(\texttt{turn})i \xrightarrow{L_i} w(\texttt{turn})j \xrightarrow{L_i} r(\texttt{turn})j.$$

Since $w(\texttt{flag[}j\texttt{]})$*true* precedes $w(\texttt{turn})j$ in program order, we conclude:

$$w(\texttt{flag[}j\texttt{]})\mathit{true} \xrightarrow{L_i} r(\texttt{flag[}j\texttt{]})\mathit{true}.$$

Therefore, lines 7 and 8 are unreachable for i unless j makes progress to <*exit*>. So, i is repeatedly executing lines 4 and 5 and $w(\texttt{flag[}i\texttt{]})$*false* of line 5 must eventually appear in $(O|j \cup O|w, \xrightarrow{L_j})$, and consequently j enters its <*critical section*>.

Fairness: To see that Burns' algorithm is unfair for PC-G, we show it's unfair even for SC.[4] Consider the Computation 5 which represents a starvation scenario, where the segments enclosed by square brackets can be repeated indefinitely.

Computation 5

$$\begin{cases} i: [w(\texttt{flag[}i\texttt{]})\mathit{true}\ w(\texttt{turn})i\ r(\texttt{turn})i\ r(\texttt{flag[}j\texttt{]})\mathit{false} \\ \quad < \mathit{critical\ section} > w(\texttt{flag[}i\texttt{]})\mathit{false}] \\ j: w(\texttt{flag[}j\texttt{]})\mathit{true}\ w(\texttt{turn})j\ [r(\texttt{turn})i\ w(\texttt{flag[}j\texttt{]})\mathit{false} \\ \quad r(\texttt{flag[}i\texttt{]})\mathit{true}] \end{cases}$$

The following is an SC linearization. $(O, \xrightarrow{L}) = \langle w_j(\texttt{flag[}j\texttt{]})\mathit{true}\ w_j(\texttt{turn})j\ [w_i(\texttt{flag[}i\texttt{]})\mathit{true}\ w_i(\texttt{turn})i\ r_i(\texttt{turn})i\ r_j(\texttt{turn})i\ w_j(\texttt{flag[}j\texttt{]})\mathit{false}\ r_j(\texttt{flag[}i\texttt{]})\mathit{true}\ r_i(\texttt{flag[}j\texttt{]})\mathit{false} < \mathit{critical\ section} > w_i(\texttt{flag[}i\texttt{]})\mathit{false}]\rangle$. Operations are subscripted by the corresponding process id. The segment enclosed in square brackets is the part of the computation being repeated indefinitely.

■

5 Summary

PC-G is a consistency model that satisfies both Pipelined-RAM consistency and Coherence. Furthermore, for each process, there must be a single linearization that meets both requirements simultaneously. Even the slight relaxation to a consistency model that is the intersection of both Pipelined-RAM and Coherence (but which permits distinct linearizations for each requirement) is too weak to support a solution to CSP without using stronger objects than simple variables (even unbounded ones). This can be proved with techniques similar to ones used here [7]. Thus, PC-G appears to be the weakest memory consistency model in the literature that has a solution to CSP using only reads and writes to shared variables.

Any solution to CSP(n) for PC-G must use at least one multi-writer and n single-writers. Burns' algorithm, which uses one multi-writer and n single-writers and is correct for PC-G, establishes that this bound is tight. But Burns' algorithm is unfair. Peterson's algorithm for two processes, which uses one multi-writer and 2 single writers

[4] It is well known that Burns' algorithm is unfair even for SC.

and is correct and fair for PC-G, shows that this lower bound is tight even for fair solutions when $n = 2$. It is not clear to us yet whether a fair solution for n processes can be constructed using only one multi-writer and n single-writers. If not, then to tighten the lower bound in the general case, impossibility proofs will have to exploit fairness. Many other algorithms that use the same number and type of variables as Burns' have been shown to fail for PC-G. Finally, Peterson's algorithm, which uses $n-1$ multi-writers and n single-writers, is correct and fair for PC-G.

References

1. M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th Int'l Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.
2. H. Attiya, S. Chaudhuri, R. Friedman, and J. L. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal of Computing*, 27(1):65–89, February 1998.
3. J. E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Symp. on Foundations of Computer Science*, pages 169–174, 1981.
4. M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. 13th Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.
5. J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.
6. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):1–5, May 1997.
7. J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.
8. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communication of the ACM*, 17(8):453–455, August 1974.
9. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
10. L. Lamport. The mutual exclusion problem (parts I and II). *Journal of the ACM*, 33(2):313–326 and 327–348, April 1986.
11. R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.
12. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
13. M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.
14. A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., 1999.
15. P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th Symp. on Foundations of Computer Science*, 1986.

A CSP Algorithms

For each of the following CSP algorithms, processes have unique identifiers from the set $\{0,\cdots,n-1\}$, where n is the total number of processes. The algorithms are given by specifying the <*entry*> and <*exit*> sections of process i, $i \in \{0,\cdots,n-1\}$.

Peterson's Algorithm

```
flag[0 .. n−1] in {−1 .. n−2}
turn[0 .. n−2] in {0 .. n−1}

<entry>
for k = 0 to n−2 do
      flag[i] ← k
      turn[k] ← i
      while (∀ j ≠ i, flag[j] ≥ k and
            turn[k] = i) do nothing

<critical section>

<exit>
flag[i] ← −1
```

Dekker's Algorithm (2 processes)

```
flag[0 .. 1] in {true, false}
turn in {0,1}

<entry>
flag[i] ← true
while (flag[j]) do
      if (turn = j) then
         flag[i] ← false
         while (turn = j) do nothing
         flag[i] ← true
      end-if
end-while

<critical section>

<exit>
turn ← j
flag[i] ← false
```

Eisenberg and MacGuire's Algorithm

```
flag[0 .. n−1] in {idle, requesting, in-cs}
turn in {0,···,n−1}

<entry>
repeat
       flag[i] ← requesting
       j ← turn
       while (j ≠ i) do
             if (flag[j] ≠ idle) then
                j ← turn
             else j ← (j+1) mod n
       end-while
       flag[i] ← in-cs
until ((∀ j ≠ i, flag[j] ≠ in-cs) and
       (turn = i or flag[turn] = idle))
turn ← i

<critical section>

<exit>
j ← (turn+1) mod n
while (j ≠ turn and flag[j] = idle) do
      j ← (j+1) mod n
end-while
turn ← j
flag[i] ← idle
```

Dijkstra's Algorithm

```
flag[0 .. n−1] in {idle, requesting, in-cs}
turn in {0,···,n−1}

<entry>
repeat
       flag[i] ← requesting
       while (turn ≠ i) do
             if (flag[turn] = idle) then
                turn ← i
       end-while
       flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)

<critical section>

<exit>
flag[i] ← idle
```

De Bruijn's Algorithm

```
flag[0 .. n−1] in {idle, requesting, in-cs}
turn in {0,···,n−1}

<entry>
repeat
       flag[i] ← requesting
       j ← turn
       while (j ≠ i) do
             if (flag[j] ≠ idle) then
                j ← turn
             else j ← (j−1) mod n
       end-while
       flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)

<critical section>

<exit>
if (flag[turn] = idle and turn = i) then
   turn ← (turn−1) mod n
end-if
flag[i] ← idle
```

Knuth's Algorithm

```
flag[0 .. n−1] in {idle, requesting, in-cs}
turn in {0,···,n−1}

<entry>
repeat
       flag[i] ← requesting
       j ← turn
       while (j ≠ i) do
             if (flag[j] ≠ idle) then
                j ← turn
             else j ← (j−1) mod n
       end-while
       flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)
turn ← i

<critical section>

<exit>
turn ← (i−1) mod n
flag[i] ← idle
```

Even Better DCAS-Based Concurrent Deques

David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite,
Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr.

Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803 USA

Abstract. The computer industry is examining the use of strong synchronization operations such as double compare-and-swap (DCAS) as a means of supporting non-blocking synchronization on tomorrow's multiprocessor machines. However, before such a primitive will be incorporated into hardware design, its utility needs to be proven by developing a body of effective non-blocking data structures using DCAS.

In a previous paper, we presented two linearizable non-blocking implementations of concurrent deques (double-ended queues) using the DCAS operation. These improved on previous algorithms by nearly always allowing unimpeded concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation but also uses only a single DCAS per push or pop in the best case.

This paper answers that question in the affirmative. We present a new non-blocking implementation of concurrent deques using the DCAS operation. This algorithm provides the benefits of our previous techniques while overcoming drawbacks. Like our previous approaches, this implementation relies on automatic storage reclamation to ensure that a storage node is not reclaimed and reused until it can be proved that the node is not reachable from any thread of control. This algorithm uses a linked-list representation with dynamic node allocation and therefore does not impose a fixed maximum capacity on the deque. It does not require the use of a "spare bit" in pointers. In the best case (no interference), it requires only one DCAS per push and one DCAS per pop. We also sketch a proof of correctness.

1 Introduction

In academic circles and in industry, it is becoming evident that non-blocking algorithms can deliver significant performance benefits [3, 20, 17] and resiliency benefits [9] to parallel systems. Unfortunately, there is a growing realization that existing synchronization operations on single memory locations, such as compare-and-swap (CAS), are not expressive enough to support design of efficient non-blocking algorithms [9, 10, 12], and software emulations of stronger primitives from weaker ones are still too complex to be considered practical [1, 4, 7, 8, 21]. In response, industry is currently examining the idea of supporting

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 59–73, 2000.

	Array with centralized access (see [9])	Array used as circular buffer (see [2])	Linked list with tagged pointers (see [2])	Snark (with garbage collection) (this paper)
Left and right accesses interfere	yes	no	no	no
Fixed limit on size of deque	yes	yes	no	no
Tag bit needed in pointers	no	no	yes	no
DCAS ops per unimpeded pop	1	1	2	1
DCAS ops per unimpeded push	1	1	1	1
Number of reserved values	1	1	3	0
Storage allocator calls per push	0	0	1	1
Storage overhead per item	none	none	2 pointers	2 pointers

Table 1. Comparison of various DCAS-based deque algorithms

stronger synchronization operations in hardware. A leading candidate among such operations is double compare-and-swap (DCAS), a CAS performed atomically on *two* memory locations. However, before such a primitive can be incorporated into processor design, it is necessary to understand how much of an improvement it actually offers. One step in doing so is developing a body of efficient data structures and associated algorithms based on the DCAS operation.

There have recently been several proposed designs for non-blocking linearizable concurrent double-ended queues (*deques*) using the double compare-and-swap operation [9, 2]. Deques, as described in [15] and currently used in load balancing algorithms [3], are classic structures to examine, in that they involve all the intricacies of LIFO-stacks and FIFO-queues, with the added complexity of handling operations originating at both ends of the deque.

Massalin and Pu [16] were the first to present a collection of DCAS-based concurrent algorithms. They built a lock-free operating system kernel based on the DCAS operation (`CAS2`) offered by the Motorola 68040 processor, implementing structures such as stacks, FIFO-queues, and linked lists.

Greenwald, a strong advocate for using DCAS, built a collection of DCAS-based concurrent data structures improving on those of Massalin and Pu. In the best case (no interference from other threads), his array-based deque algorithms required one DCAS per push and one DCAS per pop. Unfortunately, these algorithms used DCAS in a restrictive way. The first ([9] pp. 196–197) used the two-word DCAS as if it were a three-word operation, keeping the two deque end pointers in the same memory word, and DCAS-ing on it and a second word containing a value; this prevents truly concurrent, noninterfering access to the two deque ends. The second algorithm ([9] pp. 219–220) assumed an array of unbounded size, and did not correctly detect when the deque is full in all cases.

Arora et al. [3] present an elegant CAS-based restricted deque with applications in job-stealing algorithms. This non-blocking implementation needs only a single CAS operation since it restricts one side of the deque to be accessed by only a single processor, and the other side to allow only pop operations.

In a recent paper [2], we presented two new linearizable non-blocking implementations of concurrent deques using the DCAS operation. One used an array representation, and improved on previous algorithms by allowing uninterrupted concurrent access to both ends of the deque while correctly handling the difficult boundary cases when the deque is empty or full. In the best case, this array technique required one DCAS per push and one DCAS per pop. A drawback of the array representation was that it imposed a fixed maximum capacity on the queue. The second implementation corrected this by using a dynamic linked-list representation, and was the first non-blocking unbounded-memory deque implementation. Drawbacks of this list-based implementation were that it required a "spare bit" in certain pointers to serve as a boolean flag and that it required at least two (amortized) DCAS operations per pop.

A remaining open question was whether, using DCAS, one can design a non-blocking implementation of concurrent deques that allows dynamic memory allocation, as in the linked-list algorithms of [2], but also uses only a single DCAS per push or pop in the best case, as in array-based algorithms [2, 9]. This paper answers that question in the affirmative. Table 1 outlines the characteristics of the various algorithms. The first six rows indicate that the algorithm presented in this paper avoids drawbacks of previous work.

2 Modeling DCAS and Deques

Our computation model follows [5, 6, 14] as well as our own previous paper [2]. A *concurrent system* is a collection of n *processors*, which communicate through shared data structures called *objects*. Each object provides a set of primitive *operations* that are the only means of manipulating that object. Each processor is a thread of control [14] that sequentially invokes object operations by issuing an invocation and then receiving the associated response before issuing the next invocation. A *thread behavior* is the entire set of invocations and associated responses associated with a single thread; this set is totally ordered in time according to the order in which the thread issued and received the invocations and responses. A *system behavior* is the (disjoint) union of the thread behaviors of all the threads in a concurrent system.

A *history* is a system behavior upon which a total order has been imposed on invocations and responses that is consistent with the orderings of the thread behaviors. Each history may be regarded as a "real-time" order of operations where an operation A is said to *precede* another operation B if A's response occurs before B's invocation. Two operations are *concurrent* if they are unrelated by the real-time order. When we reason about the possible behaviors of a system or a thread within that system, we typically try to characterize the set of possible histories of the system.

A *sequential history* is a history in which each invocation is followed immediately by its associated response. The *sequential specification* of an object is a set of permitted sequential histories. The basic correctness requirement for a concurrent implementation is *linearizability* [14]: for every history H that may be

realized by the system, there exists a sequential history that is in the intersection of the sequential specifications of all the objects in the system and whose total order of operations is consistent with the H's partial order of operations. In a linearizable implementation, each operation appears to take effect atomically at some point between its invocation and its associated response.

In our model, every shared memory location L of a multiprocessor machine's memory is a linearizable implementation of an object that provides every processor P_i with a set of sequentially specified machine operations (see [11, 13]):

$Read_i(\&L)$ reads location L and returns its value.
$Write_i(\&L, v)$ writes the value v to location L.
$DCAS_i(\&L1, \&L2, o1, o2, n1, n2)$ is a double-compare-and-swap operation with the semantics described below.

(The *address operator* `&` is used to pass the address of a location to an operation.) Because we assume a linearizable implementation, we can, in effect, assume that these operations are atomic when reasoning about programs that use them.

For the purposes of this paper, when we write code in a high-level language, we assume that each field of a high-level-language object and each global variable may be treated as a shared memory location. A simple reference to such a field or variable is a *Read* operation; a simple assignment to such a field or variable is a *Write* operation; and a method or subroutine called `DCAS` is used to perform the *DCAS* operation on two fields or variables.

The implementation we present is *non-blocking* (also called *lock-free*) [13]. Let us use the term *higher-level operations* to refer to operations of an object being implemented, and *lower-level operations* to refer to the (machine) operations in terms of which it is implemented. A *non-blocking* implementation is one for which any history that has invocations of some set O of higher-level operations but no associated responses may contain any number of responses for high-level operations concurrent with those in O. That is, even if some higher-level operations (each of which may be continuously taking steps, or not) never complete, other invoked operations may nevertheless continually complete. Thus the system as a whole can make progress; individual processors cannot be blocked, only delayed, by other processors continuously taking steps or failing to take steps. Using locks would violate the above condition, hence the alternate name *lock-free*.

Figure 1 contains code for the DCAS operation; for comparison, it also shows code for the simpler CAS operation (which is not used in the algorithms presented here). For either operation, the sequence of suboperations is assumed to be executed atomically, either through hardware support [12, 18, 19] or through a non-blocking software emulation [7, 21].

A CAS operation examines one memory location and compares its contents to an expected "old" value. If the contents match, then the contents are replaced with a specified "new" value and an indication of success is returned; otherwise the contents are unchanged and an indication of failure is returned.

A DCAS operation may be viewed as two yoked CAS operations: mismatch in either causes both to fail. (Note: the algorithms in this paper do not require the overloaded versions of DCAS that we used in our previous paper [2].)

```
boolean CAS(val *addr,
            val old,
            val new1) {
  atomically {
    if (*addr == old) {
      *addr = new;
      return true;
    } else return false;
  }
}
```

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1 == old1) &&
        (*addr2 == old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else return false;
  }
}
```

Fig. 1. Single and Double Compare-and-Swap Operations

We assume that a CAS operation is substantially more expensive than a simple read or write of a shared variable, and that a DCAS is rather more expensive than a CAS. We also assume that memory operations (Read, Write, DCAS) that operate on distinct locations can be carried out concurrently, but those that operate on the same location are carried out sequentially, so there is a potential performance advantage in, for example, avoiding having operations on one end of a deque touch variables associated with the other end of the deque.

A *deque* S is a concurrent shared object created by a `makeDeque(length)` operation that allows each processor to perform one of four types of operations on S: `pushRight`, `popRight`, `pushLeft`, and `popLeft`.

We require that a concurrent implementation of a deque object be one that is linearizable to a standard sequential deque of the type described in [15].

The state of a deque is a sequence of items $S = \langle v_0, \ldots, v_k \rangle$ having cardinality $|S|$ where $0 \leq |S| \leq$ `length`. A deque is initially empty, that is, has cardinality 0. A deque is said to be full when its cardinality is `length`. (For the purposes of this paper, the `length` of the deque is essentially the total amount of storage available for allocation as deque node objects.)

The four possible push and pop operations induce the following state transitions of the sequence $S = \langle v_0, \ldots, v_k \rangle$, with appropriate returned values:

- `pushRight`(v_{new}), if S is not full, changes S to be $\langle v_0, \ldots, v_k, v_{new} \rangle$ and returns "okay"; if S is full, it returns "full" and S is unchanged.
- `pushLeft`(v_{new}), if S is not full, changes S to be $\langle v_{new}, v_0, \ldots, v_k \rangle$ and returns "okay"; if S is full, it returns "full" and S is unchanged.
- `popRight`(), if S is not empty, changes S to be $\langle v_0, \ldots, v_{k-1} \rangle$ and returns v_k; if S is empty, it returns "empty" and S is unchanged.
- `popLeft`(), if S is not empty, changes S to be $\langle v_1, \ldots, v_k \rangle$ and returns v_0; if S is empty, it returns "empty" and S is unchanged.

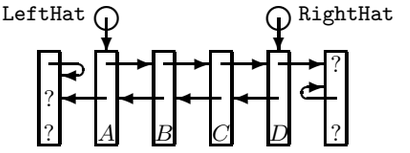
For example, starting with an empty deque $S = \langle\rangle$, `pushRight`(1) changes the state to $S = \langle 1 \rangle$; `pushLeft`(2) transitions to $S = \langle 2, 1 \rangle$; then `pushRight`(3) transitions to $S = \langle 2, 1, 3 \rangle$. A subsequent `popLeft`() transitions to $S = \langle 1, 3 \rangle$ and returns 2; then `popLeft`() transitions to $S = \langle 3 \rangle$ and returns 1 (which had been pushed from the right).

3 The "Snark" Linked-list Deque

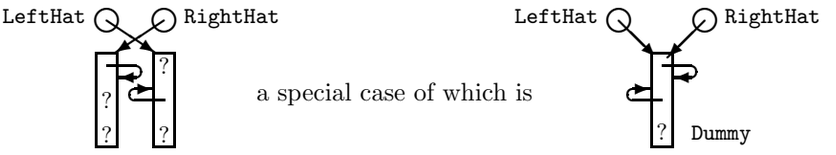
Our implementation (we have arbitrarily nicknamed it *Snark*) represents a deque as a doubly-linked list of nodes. Each node in the list contains two link pointers `R` and `L` and a value `V` (see Figure 2 below). There are two global "anchor" variables, arbitrarily called `LeftHat` and `RightHat` (lines 7–8), which generally point to the leftmost node and the rightmost node in the chain.

A node whose `L` field points to that same node is said to be *left-dead*; a node whose `R` field points to that same node is said to be *right-dead*. If `LeftHat` points to a node that is not left-dead, then the `L` field of that node points to a right-dead node; if `RightHat` points to a node that is not right-dead, then the `R` field of that node points to a left-dead node. As we will see, `LeftHat` points to a left-dead node if and only if `RightHat` points to a right-dead node; such a situation represents a deque with no items in it. The special node `Dummy` is both left-dead and right-dead (lines 6–7); as we will see, no other node is ever both left-dead and right-dead. In all cases, once a node becomes left-dead, it remains left-dead (until the node is determined to be inaccessible and therefore eligible to be reclaimed); once a node becomes right-dead, it remains right-dead. These rules may seem somewhat complicated, but they lead to a uniform implementation of pop operations.

A typical deque, with values A, B, C, and D in it, looks like this:



where ? indicates a "don't care" pointer or value. An empty deque looks like:



Figures 3 and 4 show non-blocking implementations of push and pop operations on the right-hand end of the deque. We describe these operations in detail. The left-hand-side operations shown in Figures 5 and 6 are symmetric.

The right-side push operation first obtains a fresh `Node` structure from the storage allocator (Figure 3, line 2). (Note that the problem of implementing a

```
1 structure Node {
2   Node *R;
3   Node *L;
4   val V;       }
```

```
5 Node Dummy = new Node();
6 Dummy.L = Dummy.R = Dummy;
7 Node *LeftHat = Dummy;
8 Node *RightHat = Dummy;
```

Fig. 2. The array-based deque—data structure and hats (anchors).

```
 1 val pushRight(val v) {
 2   nd = new Node();  /* Allocate new Node structure */
 3   if (nd == null) return "full";
 4   nd->R = Dummy;
 5   nd->V = v;
 6   while (true) {
 7     rh = RightHat;                                  /* Labels A, B,   */
 8     rhR = rh->R;                                    /* etc., are used */
 9     if (rhR == rh) {                                /* in the proof   */
10       nd->L = Dummy;                                /* of correctness */
11       lh = LeftHat;
12       if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))          /* A */
13         return "okay";
14     } else {
15       nd->L = rh;
16       if (DCAS(&RightHat, &rh->R, rh, rhR, nd, nd))           /* B */
17         return "okay";
18 } } }                              // Please forgive this brace style
```

Fig. 3. Simple linked-list deque—right-hand-side push.

non-blocking storage allocator is not addressed in this paper, but would need to be solved to produce a completely non-blocking deque implementation.) We assume that if allocatable storage has been completely exhausted (even after automatic reclamation has occurred), the `new` operation will yield a null pointer; the push operation treats this as sufficient cause to report that the deque is full (line 3). Otherwise, the `R` field of the new node is made to point to `Dummy` (line 4) and the value to be pushed is stored into the `V` field (line 5); all that remains is to splice this new node into the doubly-linked chain. But an attempt to splice might fail (because of an action by some other concurrent push or pop), so a "while true" loop (line 6) is used to iterate until a splice succeeds.

The `RightHat` is copied into local variable `rh` (line 7)—this is important. If `rh` points to a right-dead node (line 9), then the deque is empty. In this case, the new node should become the only node in the deque. Its `L` field is made to point to `Dummy` (line 10) and then a DCAS is used (line 12) to atomically make both `RightHat` and `LeftHat` point to the new node—but only if neither hat has changed. If this DCAS succeeds, then the push has succeeded (line 13); if the DCAS fails, then control will go around the "while true" loop to retry.

If the deque is not empty, then the new node must be added to the right-hand end of the doubly-linked chain. The copied content of the `RightHat` is stored into the `L` field of the new node (line 15) and then a DCAS is used (line 16) to make both the `RightHat` and the former right-end node point to the new node, which thus becomes the new right-end node. If this DCAS operation succeeds, then the push has succeeded (line 17); if the DCAS fails, then control will go around the "while true" loop to retry.

The right-side pop operation also uses a "while true" loop (line 2) to iterate until an attempt to pop succeeds. The `RightHat` is copied into local variable `rh`

```
 1 val popRight() {
 2   while (true) {
 3     rh = RightHat;                      // Delicate order of operations
 4     lh = LeftHat;                       // here (see proof of Theorem 4
 5     if (rh->R == rh) return "empty";    // and the Conclusions section)
 6     if (rh == lh) {
 7       if (DCAS(&RightHat, &LeftHat, rh, lh, Dummy, Dummy))     /* C */
 8         return rh->V;
 9     } else {
10       rhL = rh->L;
11       if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) {         /* D */
12         result = rh->V;
13         rh->R = Dummy;                                         /* E */
14         rh->V = null;  /* optional (see text) */
15         return result;
16 } } } }                          // Stacking braces this way saves space
```

Fig. 4. Simple linked-list deque—right-hand-side pop.

(line 7)—this is important. If `rh` points to a right-dead node, then the deque is empty and the pop operation reports that fact (line 4).

Otherwise, there are two cases, depending on whether there is exactly one item or more than one item in the deque. There is exactly one item in the deque if and only if the `LeftHat` and `RightHat` point to the same node (line 6). In that case, a DCAS operation is used to reset both hats to point to `Dummy` (line 7); if it succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node (line 8). (It is assumed that, after exit from the `popRight` routine, the node just popped will be reclaimed by the automatic storage allocator, through garbage collection or some such technique.)

If there is more than one item in the deque, then the rightmost node must be removed from the doubly-linked chain. A DCAS is used (line 11) to move the `RightHat` to the node to the immediate left of the rightmost node; at the same time, the `L` field of that rightmost node is changed to contain a self-pointer, thus making the rightmost node left-dead. If this DCAS operation fails, then control will go around the "while true" loop to retry; but if the DCAS succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node. Before this value is returned, the `R` field is cleared (line 13) so that *previously* popped nodes may be reclaimed. It may also be desirable to clear the `V` field immediately (line 14) so that the popped value will not be retained indefinitely by the queue structure. If the `V` field does not contain references to other data structures, then line 14 may be omitted.

The push and pop operations work together in a completely straightforward manner except in one odd case. If a `popRight` operation and a `popLeft` operation occur concurrently when there are exactly two nodes in the deque, then each operation may (correctly) discover that `LeftHat` and `RightHat` point to different nodes (line 6 in each of Figures 4 and 6) and therefore proceed to perform a DCAS for the multinode case (line 11 in each of Figures 4 and 6). Both of these

```
 1 val pushLeft(val v) {
 2   nd = new Node();  /* Allocate new Node structure */
 3   if (nd == null) return "full";
 4   nd->L = Dummy;
 5   nd->V = v;
 6   while (true) {
 7     lh = LeftHat;
 8     lhL = lh->L;
 9     if (lhL == lh) {
10       nd->R = Dummy;
11       rh = RightHat;
12       if (DCAS(&LeftHat, &RightHat, lh, rh, nd, nd))          /* A' */
13         return "okay";
14     } else {
15       nd->R = lh;
16       if (DCAS(&LeftHat, &lh->L, lh, lhL, nd, nd))            /* B' */
17         return "okay";
18 } } }                          // We were given a firm limit of 15 pages
```

Fig. 5. Simple linked-list deque—left-hand-side push.

DCAS operations may succeed, because they operate on disjoint pairs of memory locations. The result is that the hats pass each other:

But this works out just fine: there had been two nodes in the deque and both have been popped, but as they are popped they are made right-dead and left-dead, so that the deque is now correctly empty.

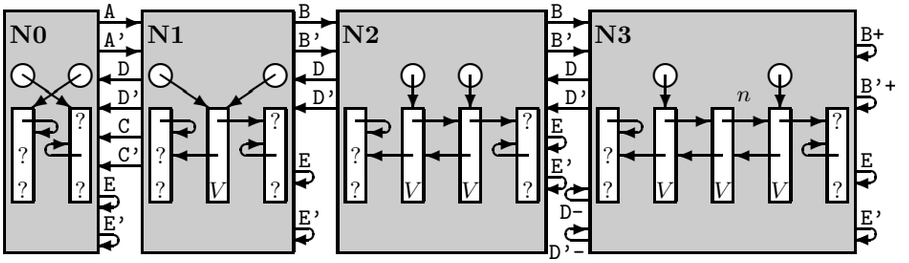
4 Sketch of Correctness Proof for the "Snark" Algorithm

We reason on a state transition diagram in which each node represents a class of possible states for the deque data structure and each transition arc corresponds to an operation in the code that can modify the data structure. For every node and every distinct operation in the code, there must be an arc from that node for that operation unless it can be proved that, when the deque is in the state represented by that node, either the operation must fail or the operation cannot be executed because flow of control cannot reach that operation with the deque in the prescribed state.

The possible states of a Snark deque are shown in the following state transition diagram:

```
 1 val popLeft() {
 2   while (true) {
 3     lh = LeftHat;                        // Delicate order of operations
 4     rh = RightHat;                       // here (see proof of Theorem 4
 5     if (lh->L == lh) return "empty";     // and the Conclusions section)
 6     if (lh == rh) {
 7       if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy))    /* C' */
 8         return lh->V;
 9     } else {
10       lhR = lh->R;
11       if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh)) {         /* D' */
12         result = lh->V;
13         lh->L = Dummy;                                        /* E' */
14         lh->V = null;  /* optional (see text) */
15         return result;
16 } } } }                 // Better to stack braces than to omit a lemma
```

Fig. 6. Simple linked-list deque—left-hand-side pop.



The rightmost node shown actually represents an infinite set of nodes, one for each integer n for $n \geq 1$, where there are $n+2$ items in the deque. The labels on the transition arcs correspond to the labels on operations that modify the linked-list data structure in Figures 3, 4, 5, and 6. The labels `B+` and `B'+` indicate a transition that increases n by 1; the labels `D-` and `D'-` indicate a transition that decreases n by 1. We will also use labels such as `A` and `A'` in the text that follows to refer to DCAS and assignment operations in those figures.

We say that a node is "in the deque from the left" if it is not left-dead and it is reachable from the node referred to by the `LeftHat` by zero or more steps of following pointers in the `L` field. We say that a node is "in the deque from the right" if it is not right-dead and it is reachable from the node referred to by the `RightHat` by zero or more steps of following pointers in the `R` field.

The Snark algorithm is proved correct largely by demonstrating that, for every DCAS operation and every possible state of the deque data structure, if the DCAS operation succeeds then a correct transition occurs as shown in the state diagram. In cases where there is no corresponding arc on the state diagram, it is necessary to prove either that the DCAS cannot succeed if the deque is in that state or that control cannot reach the DCAS with the deque in that state. Here we provide proofs only for these latter cases.

Lemma 1. *A node is in the deque from the left if and only if it is in the deque from the right (therefore from now on we may say simply "in the deque").*

Lemma 2. *If a node is in the deque and then is removed, thereafter that node is never in the deque again.*

Lemma 3. *No node except the* `Dummy` *node is ever both left-dead and right-dead.*

Proof. Initially, only the `Dummy` node exists. Inspection of the code for `pushRight` and `pushLeft` shows that newly created nodes are never made left-dead or right-dead. Only operation `D` ever makes an existing node right-dead, and only operation `D'` ever makes an existing node left-dead. But `D` and `D'` each operate on a node that is in the deque, and as it makes a node left-dead or right-dead, it removes it from the deque. By Lemma 2, a node cannot be removed twice. So the same node is never made right-dead by `D` and also made left-dead by `D'`. ∎

Lemma 4. *No node is ever made left-dead or right-dead after the node is removed from the deque.*

Proof. By Lemma 2, after a node is removed from the deque it is never in the deque again. Only operation `D` ever makes an existing node right-dead, and only operation `D'` ever makes a node left-dead. But each of these operations succeeds only on a node that is in the deque. ∎

Lemma 5. *Once a node is right-dead, it stays right-dead as long as it is reachable from any thread.*

Proof. Only operations `B`, `D'`, and `E` change the `R` field of a node. But `B` succeeds only if the node referred to by `rh` is not right-dead, and `D` always makes the node referred to by `rh` right-dead. Operation `E` always stores into the `R` field of a node that has been made left-dead as it was removed from the deque. By Lemma 3, the node was not right-dead when it was removed from the deque; by Lemma 4, the node cannot become right-dead after it was removed from the deque. Therefore when operation `E` changes the `R` field of a node, that node is not right-dead. ∎

Lemma 6. *Once a node is left-dead, it stays left-dead as long as it is reachable from any thread.*

Lemma 7. *The* `RightHat` *points to a right-dead node if and only if the deque is empty, and the* `LeftHat` *points to a left-dead node if and only if the deque is empty.*

Proof. Initially both `RightHat` and `LeftHat` point to the `Dummy` node, so this invariant is initially true. Operations `A` and `A'` make both `RightHat` and `LeftHat` point to a new node that is not left-dead or right-dead, so the deque is not empty. Operation `B` can succeed only if the `RightHat` points to a node that is not right-dead, and it changes `RightHat` to point to a new node that is not right-dead. A symmetric remark applies to `B'`. Operations `C` and `C'` make both `RightHat` and

LeftHat point to the Dummy node, which is both left-dead and right-dead, so the deque is empty. If operation D moves RightHat from a node that it not right-dead to a node that is right-dead, then the deque had only one item in it; then the LeftHat also points to the node just removed from the deque by operation D, and that operation, as it moved the RightHat and emptied the deque, also made the node left-dead. A symmetric remark applies to D'. Operations E and E' do not change whether a node is left-dead or right-dead (see proof of Lemma 5). ∎

Theorem 1. *Operation* A *fails if the deque is not empty.*

Proof. Operation A is executed only after the node referred to by rh has been found to be right-dead. By Lemma 5, once a node is right-dead, it remains right-dead. Therefore, if the deque is non-empty when A is executed, then RightHat must point to some other node than the one referred to by rh; therefore RightHat does not match rh and the DCAS must fail. ∎

Theorem 2. *Operation* B *fails if the deque is empty.*

Proof. Operation B is executed only after rhR has been found unequal to rh. If the deque is empty when B is executed, and RightHat equals rh, then the node referred to by rh must have become right-dead; but that means that rh->R equals rh, and therefore cannot match rhR, and so the DCAS must fail. ∎

Theorem 3. *Operation* C *fails unless there is exactly one item in the deque.*

Proof. When C is executed, rh equals lh, so C can succeed only when RightHat and LeftHat point to the same node. If the deque has two or more items in it, then RightHat and LeftHat contain different values, so the DCAS must fail.

If the deque is empty, and RightHat and LeftHat point to same node, then by Lemma 7 that node must be both left-dead and right-dead, and by Lemma 3 that node must be the Dummy node, which is created right-dead and (by Lemma 5) always remains right-dead. But then the test in line 5 of popRight would have prevented control from reaching operation C. Therefore, if C is executed with the deque empty, RightHat and LeftHat necessarily contain different values, so the DCAS must fail. ∎

Theorem 4. *Operation* D *fails if the deque is empty.*

Proof. This is the most difficult and delicate of our proofs. Suppose that some thread of control T is about to execute operation D. Then T, at line 3 of popRight, read a value from RightHat (now in T's local variable rh) that pointed to a node that was not right-dead when T executed line 5; therefore the deque was not empty at that time. Also, T must have read a value from LeftHat in line 4 that turned out not to be equal to rh when T executed line 6.

Now suppose, as T executes D in line 12, that the deque is empty. How might the deque have become empty since T executed line 5? Only through the execution of C or C' or D or D' by some other thread U. If U executed C or D, then it changed the value of RightHat; in this case T's execution of DCAS D must fail, because RightHat will not match rh.

So consider the case that U executed `C'` or `D'`. (Note that, for the execution of `D` by T to succeed, there cannot have been another thread U' that performed a `C'` or `D'` after U but before T's execution of DCAS `D'`, because that would require a preceding execution of `A` or of `A'`, either of which would have changed `RightHat`, causing T's execution of DCAS `D` to fail.)

Now, if U executed `C'`, then U changed the value of `RightHat` (to point to `Dummy`); therefore T's execution of DCAS `D` must fail.

If, on the other hand, U executed `D'` to make the deque empty, then the deque must have had one item in it when U executed DCAS `D'`. But thread U read values for `LeftHat` (in line 3 of `popLeft`) and `RightHat` (in line 4) that were found in line 6 not to be equal. Therefore, when U read `RightHat` in line 4, either the deque did not have exactly one item in it or the value of `LeftHat` had been changed since U read `LeftHat` in line 3. If `LeftHat` had been changed, then execution of `D'` by U would have to fail, contrary to our assumption. Therefore, if there is any hope left for execution of `D'` by U to succeed, the deque must *not* have had exactly one item in it when U read `RightHat` in line 4.

How, then, might the deque have come to hold exactly one item after U executed line 4? Only through some operation by a third thread. If that operation was `A'` or `B'` or `C'` or `D'`, that operation must have changed `LeftHat`; but that would cause the execution of DCAS `D'` by U to fail, contrary to our assumption. Therefore the operation by a third thread must have been `A` or `B` or `C` or `D`. Consider, then, the most recent execution (relative to the execution of `D` by T) of DCAS `A` or `B` or `C` or `D` that caused the deque to contain exactly one item, and let V be the thread that executed it. (It is well-defined which of these DCAS executions is most recent because DCAS operations `A`, `B`, `C`, and `D` all synchronize on a common variable, namely `RightHat`.)

If this DCAS operation by thread V occurred after thread T read `RightHat` in line 3, then it changed `RightHat` after T read `RightHat`, and the execution of DCAS `D` by T must fail. Therefore, if there is any hope left for execution of `D` by T to succeed, then execution of the most recent DCAS `A` or `B` or `C` or `D` (by V) must have occurred before T read `RightHat` in line 3.

To summarize the necessary order of events: (a) U reads `LeftHat` in line 3 of `popLeft`; (b) V executes `A` or `B` or `C` or `D`, resulting in the deque containing one item; (c) T executes lines 3, 4, 5, and 6 of `popRight`; (d) U executes `D'`; (e) T executes `D`. Moreover, there was no execution of `A` or `B` or `C` or `D` by any other thread after event (b) but before event (e), and there cannot have been any execution of `A'` or `B'` or `C'` or `D'` after event (a) but before event (d).

Therefore the deque contained exactly one item during the entire time that T executed lines 3 though 6 of `popRight`. But if so, the test in line 6 would have prevented control from reaching `D`.

Whew! We have exhausted all possible cases; therefore, if DCAS `D` is executed when the deque is empty, it must fail. ∎

Theorem 5. *Operation* `E` *always succeeds and does not change the number of items in the deque.*

Symmetric theorems apply to operations `A'`, `B'`, `C'`, and `D'`.

Space limitations prevent us from presenting a proof of linearizability and a proof that the algorithms are non-blocking—that is, if any subset of the processors invoke `push` or `pop` operations but fail to complete them (whether the thread be suspended, or simply unlucky enough never to execute a DCAS successfully), the other processors are in no way impeded in their use of the deque and can continue to make progress. However, we observe informally that a thread has not made any change to the deque data structure (and therefore has not made any progress visible to other threads) until it performs a successful DCAS, and once a thread has performed a single successful DCAS then, as observed by other threads, a push or pop operation on the deque has been completed. Moreover, each DCAS used to implement a `push` or `pop` operation has no reason to fail unless some other `push` or `pop` operation has succeeded since it was invoked.

5 Conclusions

We have presented non-blocking algorithms for concurrent access to a double-ended queue that supports the four operations `pushRight`, `popRight`, `pushLeft`, and `popLeft`. They depend on a multithreaded execution environment that supports automatic storage reclamation in such a way that a node is reclaimed only when no thread can possibly access it. Our technique improves on previous methods in requiring only one DCAS per push or pop (in the absence of interference) while allowing the use of dynamically allocated storage to hold queue items.

We have two remaining concerns about this algorithm and the style of programming that it represents. First, the implementation of the `pop` operations is not entirely satisfactory because a `popRight` operation, for example, necessarily reads `LeftHat` as well as `RightHat`, causing potential interference with `pushLeft` and `popLeft` operations even when there are many items in the queue, which in hardware implementations of interest could degrade performance.

Second, the proof of correctness is complex and delicate. While DCAS operations are certainly more expressive than CAS operations, and can serve as a useful building block for concurrent algorithms such as the one presented here that can be encapsulated as a library, after our experience we are not sure that we can wholeheartedly recommend DCAS as the synchronization primitive of choice for everyday concurrent applications programming. In an early draft of this paper, we had transposed lines 4 and 5 of Figure 4 (and similarly lines 4 and 5 of Figure 6); we thought there was no need for `popRight` to look at the `LeftHat` until the case of an empty deque had been disposed of. We were wrong. As we discovered when the proof of Theorem 4 would not go through, that version of the code was faulty, and it was not too difficult to construct a scenario in which the same node (and therefore the same value) could be popped twice from the queue. As so many (including ourselves) have discovered in the past, when it comes to concurrent programming, intuition can be extremely unreliable and is no substitute for careful proof. While we believe that non-blocking algorithms are an important strategy for building robust concurrent systems, we also believe it is desirable to build them upon concurrency primitives that keep the necessary proofs of correctness as simple as possible.

References

1. Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *Proc. 16th ACM Symp. Principles of Dist. Computing*, pages 111–120, August 1997. Santa Barbara, CA.
2. O. Agesen, D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. DCAS-based concurrent deques. In *Proc. 12th ACM Symp. Parallel Algorithms and Architectures (to appear)*, July 2000.
3. N. S. Arora, R. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures*, 1998.
4. H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th ACM Symp. Principles of Dist. Computing*, May 23–26 1996. Philadelphia, PA.
5. Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(4):725–763, July 1994.
6. Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, March 1998.
7. G. Barnes. A method for implementing lock-free shared data structures. In *Proc. 5th ACM Symp. Parallel Algorithms and Architectures*, pages 261–270, June 1993.
8. B. N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proc. 13th IEEE International Conf. on Distributed Computing Systems*, pages 264–273. IEEE Computer Society Press, May 25–28 1993. Los Alamitos, CA.
9. M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 1999.
10. M. B. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symp. Operating Systems Design and Implementation*, pages 123–136, October 28–31 1996. Seattle, WA.
11. M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Trans. Programming Languages and Systems*, 15(5):745–770, November 1993.
12. M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992.
13. M. P. Herlihy. Wait-free synchronization. *ACM Trans. Programming Languages and Systems*, 13(1):123–149, January 1991.
14. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems*, 12(3):463–492, July 1990.
15. D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison-Wesley, 2nd edition, 1968.
16. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report TR CUCS–005–9, Columbia University, New York, NY, 1991.1¡
17. M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR 599, Computer Science Department, University of Rochester, 1995.
18. Motorola. *MC68020 32-Bit Microprocessor User's Manual*. Prentice-Hall, 1986.
19. Motorola. *MC68030 User's Manual*. Prentice-Hall, 1989.
20. Martin C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Trans. Computer Systems*, 17(4):337–371, November 1999.
21. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

Distributed Algorithms for English Auctions
(Extended abstract)

Yedidia Atzmony and David Peleg *

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science
Rehovot, 76100 Israel
{didi,peleg}@wisdom.weizmann.ac.il

Abstract. This paper deals with the implementation of an English auction on a distributed system. We assume that all messages are restricted to bids and resignations (referred to as the *limited communication* assumption) and that all participants are trying to maximize there gains (referred to as the *prudence* assumption). We also assume that bidders are risk-neutral, and that the underlying communication network is complete, asynchronous and failure-free. Under these assumptions, we show that the time and communication requirements of any auction process are $\Omega(M_2)$ and $\Omega(M_2 + n)$ respectively, where M_2 denotes the second largest valuation of a participant in the auction.
We then develop a number of distributed algorithmic implementations for English auction, analyze their time and communication requirements, and propose an algorithm achieving optimal time and communication, i.e., meeting the above lower bounds. Finally we discuss extensions to the case of dynamically joining participants.

1 Introduction

1.1 Background

The theory of auctions is a well-researched area in the field of economics (cf. [5, 6, 8, 9] and the references therein). While auctions come in many different forms (such as Dutch Auction, first price sealed bids, Vickrey auction, double auction and many other variations), in this paper we focus on the one of the most commonly used methods, namely, the English auction.

An English auction proceeds as follows. The auctioneer (be it the seller or a third party, e.g., an auction house) displays an article to be sold and announces the *reserved price*, namely, the minimal price the auction begins with (hereafter assumed w.l.o.g. to be 0). The auction process consists of bidders making increasingly higher bids, until no one is willing to pay more. The highest bidder buys the article at the proposed price. There may be a minimal increase for each bid; for simplicity we assume (w.l.o.g. again) that this minimum increase is 1.

* Supported in part by a grant from the Israel Ministry of Science and Art.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 74–88, 2000.

The recent advent of Internet based *electronic commerce* has led to fast development in the use of *online auctions* (cf. [1, 4, 3] and the references therein). The main advantage of online auctioning is that the bidders do not have to attend personally; an agent (including an electronic agent) would suffice. This means that the sale is not confined to any physical place, and bidders from any part of the world are allowed to join the auction. The most popular type of auction chosen by the different Internet auction houses is by far the English auction.

For simplicity, we ignore a number of complicating factors which arise in actual auction systems. For instance, the system is assumed to take care of security issues. (For a treatment of these issues see [7].) Also, we do not address the issue of fault-tolerance, i.e., it is assumed that the auction system is reliable and fault-free. Finally, an aspect unique to auctions which will not be dealt with directly in this paper concerns potential attempts by the auctioneer or the participants to influence the outcome of the auction (in lawful and/or unlawful ways). For example, the auctioneer may try to raise the offer to the maximum using a shill (namely, a covert collaborator) in the auction. In the case of computerized English auctioning, that shill does not really have to exist; it can be an imaginary bidder imitated by the auctioneer. But, as in live auctioning, the auctioneer may end up selling the article to itself.

1.2 The model

Let us next describe a model for auctions in a distributed network-based system. The basic components of this model include the underlying communication network, the auction system, and the auction protocol.

The communication network: The underlying *communication network* is represented by a complete n-vertex graph, where the vertices $V = \{v_1, \ldots, v_n\}$ represent the network processors and every two vertices are connected by an undirected edges, representing a bidirectional communication channel.

Communication is asynchronous, and it is assumed that at most one message can occupy a communication channel at any given time. Hence, the channel becomes available for the next transmission only after the receiving processor reads the previous message.

The auction system: The *auction system* is formally represented as a pair $\mathcal{A} = \langle \beta, n \rangle$, where n is the number of nodes of V hosting the auction participants, and β is the function assigning valuations to the bidders. Without loss of generality we assume that the auctioneer resides in the node $A = v_1$. For simplicity, it is assumed that each node hosts a single bidder, thus in an n-vertex network there may be at most n bidders. In real life situations, a single processor (or network node) may host any number of prospective bidders. Our algorithmic approach can be easily altered to accommodate such situations.

The *valuation* $\beta(v_i)$ assigned to each participant is a natural number representing the maximal offer the participant is willing to bid. These valuations may depend on a number of parameters, and in principle may change as the bidding

progresses. This might be the case, for instance, if some participants are *risk-averse*. In the current paper we ignore the issue, simply assuming *risk-neutral* participants whose valuation $\beta(v_i)$ remains constant during the entire bidding process.

The auction process: The behavior of any auction is determined by the auction system $\mathcal{A}$, which can be thought of as the input, and the set of protocols ALG used by the participants. An *auction process* $\eta^{\text{ALG}}(\mathcal{A})$ is the execution of a given algorithm ALG over a given auction system $\mathcal{A}$. The algorithm ALG consists of a set of bidder protocols ALG_i invoked by the participating nodes v_i, and a special protocol ALG_A used by the node A hosting the auctioneer. Note that the bidder protocols are not required to be identical. This imitates real auctioning, since in reality, each participant has its own bidding policy. The behavior of a protocol ALG_i, executed by v_i relies solely on $\beta(v_i)$ and the inputs received by v_i during the auction process.

English auctions: As explained earlier, in an English auction the auctioneer declares an initial price and the bidders start bidding up until no one offers a new bid or the auctioneer decides to stop the auction at a given price. To make our model concrete, the following assumptions are postulated on the network and the participants.

The auctioneer will sell the article to the highest bidder. Let M denote the set of all the integers that occur as valuations in a given auction system $\mathcal{A}$. The members of M are denoted in decreasing order by $M_1, M_2, \ldots$ and so on. Hence assuming the auction is carried to the end, the participant with valuation M_1 will win the auction. In case there is more then one participant with the highest valuation, the first one to bid the maximal offer is the winner. In case of a simultaneous bid, some sort of a tiebreaker will determine the winner.

It is assumed that during the bidding, each offer is final and obligating. Also, once a participant has resigned (namely, failed to offer a bid upon request), it cannot rejoin the bidding process. Hence the auction starts with a group of possible bidders P, and as the auction process progresses, the set P is partitioned into two disjoint sets, namely the set of *active participants* at the beginning of round t, denoted AP_t, and the set of *resigned participants*, denoted RP_t. At any time t, $AP_t \cap RP_t = \emptyset$ and $AP_t \cup RP_t = P$. Of course, both sets change as the auction progresses and participants move from AP_t to RP_t.

The current (highest) offer at the end of round t is denoted by B_t (later on, t is sometimes omitted). Initially $B_0 = 0$. On each round t of the execution, the auctioneer addresses a set of participants, henceforth referred to as the *query set* of round t. It presents the members of this set with the current bid B_{t-1}, and requests a new, higher bid. Upon receiving the bidding request, each addressed participant v_i decides, according to its protocol Alg_i, whether to commit to a new offer $B(v_i, t)$ or resign. The auctioneer waits until receiving a reply (in the form of a bid $B(v, t)$ or a resignation) from each addressed participant. According to these answers, the auctioneer updates B_t and the sets AP_t and RP_t, and decides on its next step. In particular, in case there were one or more bidders, the

auctioneer appoints one of them as the current winner, denoted W_t. In contrast, if all approached participants have resigned, then the auctioneer will approach a new query set in the next round. The process continues until all participants but the current winner resign, upon which the auction terminates.

At any given moment during the bidding process, the current configuration of the auction system is described as a tuple $\mathcal{C}_t = \langle B_t, W_t, AP_t \rangle$, where B_t denotes the current bid, W_t is the current winner (namely, the participant committed to B_t) and AP_t is the set of currently active participants. The initial auction configuration is $\mathcal{C}_0 = \langle 0, null, V \rangle$.

Communications and time complexities: It is assumed that sending a message from v_i to a neighbor v_j takes one time unit. The minimal data unit that needs to be transferred in an auction is the ID of a participant, which requires $O(\log n)$ bits, and the offer itself, $b = B(v,t)$, which takes $\log b$ bits. Henceforth, we assume that the allowable message size is some value m large enough to hold the offer, i.e., $m = \Omega(\log n + \log b)$. (Our results can be readily extended to models allowing only fixed-size messages, in the natural way.)

The *time complexity* of a given algorithm ALG on a given auction system $\mathcal{A}$, denoted $T_{\mathrm{ALG}}(\mathcal{A})$, is the number of time units incurred by an execution $\eta^{\mathrm{ALG}}(\mathcal{A})$ of ALG on $\mathcal{A}$ from beginning to completion in the worst case. The *communication complexity* of ALG on $\mathcal{A}$, denoted $C_{\mathrm{ALG}}(\mathcal{A})$, is the number of messages of size m incurred by the execution $\eta^{\mathrm{ALG}}(\mathcal{A})$ in the worst case.

1.3 Our results

In this paper we initiate the study of distributed implementations for English auctions, and their time and communication complexities. Section 2 discusses our assumptions and their basic implications. We assume that all messages are restricted to bids and resignations. This is referred to as the *limited communication* assumption. We also assume that all participants are trying to maximize there gains. This is referred to as the *prudence* assumption. Under these assumptions, it is shown that the time and communication requirements of any auction protocol are $\Omega(M_2)$ and $\Omega(M_2 + n)$ respectively. Our main result is presented in Section 3, in which we develop a number of distributed algorithmic implementations for English auction, analyze their time and communication requirements, and propose an algorithm achieving optimal time and communication, i.e., meeting the above lower bounds. Finally we discuss an extension of our algorithm to the case of dynamically joining participants.

2 Basic properties

2.1 Assumptions on computerized English auctions

The assumptions on English auctions stated in Section 1.2 follow the behavior of live English auctions, and serve as basic guidelines in any implementation of

English auction. In contrast, the set of assumptions stated next is optional; these assumptions are not always essential but they are often natural, and are taken in order to facilitate handling an auction in a distributed setting.

The first assumption is that the auction is *limited*, meaning that the data sent over the network is limited to participant ID's accompanied by bids or resignations. This restriction adheres to the behavior of live English auctions, where bids are the only type of communication allowed between the auctioneer and attendees. (Clearly, in reality it is not feasible to enforce this requirement since an auction house or participant on the Internet cannot monitor the behavior and private communications between any members participating in the auction. As mentioned earlier, in the current paper we do not address enforcement issues.)

It is assumed that none of the participants will exceed its valuation. Conversely, it is assumed that a participant v_i will not resign until the current highest bid exceeds its valuation $\beta(v_i)$. In other words, the bidders are *risk-neutral*. This leads to a natural bidder protocol, by which the participant v_i responds to each request with a higher bid whenever the current highest offer B satisfies $B \leq \beta(v_i)$, and a resignation otherwise.

Note that this is not the only risk-neutral protocol. In case the current offer B satisfies $B \leq \beta(v_i)$, the bidder v_i may raise the bid to any value between B and $\beta(v_i)$ and still maintain risk-neutrality. We now introduce another reasonable assumption (called *prudence*), and show that under this assumption, a unit increment becomes mandatory. The *prudence* assumption concerns the rational behavior of the auctioneer and the bidders. We say that the auction is *prudent* if all participants try to maximize their success. For the bidders this means paying the minimal price possible. For the auctioneer it means receiving the highest offer possible.

Definition 1. *A protocol* ALG_i *is* bidder-prudent *if it ensures that the bidder that wins the auction pays* at most $M_2 + 1$ *(where* M_2 *is the second highest valuation among all participants). When more than one participant has valuation* M_1*, the price is at most* M_1*.*
A protocol ALG_A *is* auctioneer-prudent *if it always results with the highest possible offer.*
An auction protocol is prudent *if it is both auctioneer-prudent and bidder-prudent.*

Consequently, when an auction is prudent, the final offer would be exactly $M_2 + 1$ (it may be M_1 in case of a tie, or M_2 if the second highest valuation was offered by the participant with the highest valuation).

As mentioned earlier, we do not deal with the question of how prudence can be enforced against possible attempts of cheating, by the auctioneer or some of the participants.

Hereafter, we make the following assumptions, for simplicity of presentation. First, there is only one participant with the maximal valuation M_1. Secondly, M_2 is always offered by someone other than the participant with the maximal valuation. Notice that these assumptions are not necessary, and all our results may be easily modified to handle the omitted extreme cases.

2.2 Properties of the English auction

We next analyze the special properties implied by the above assumptions.

Lemma 1. *Any bidder-prudent protocol* ALG_v *increases the bid by at most 1 in each step.*

Proof. Suppose, for the sake of contradiction, that there exists a bidder-prudent protocol ALG_v which allows v to raise some bid by more than 1. Then, at some point t in some execution $\eta^{\text{ALG}}(\mathcal{A})$, the last bid offered by some participant w was $B_t = B(w,t)$ and at time $t+1$, v offers $B(v,t+1) > B(w,t)+1$.

Let Z be the set of participants other than v with valuation at least $B(w,t)$,

$$Z = \{v' \neq v \mid \beta(v') \geq B(w,t)\}.$$

Consider now a different scenario, over an auction system $\mathcal{A}' = \langle \beta', n \rangle$, where $\beta'(u) = B(w,t)$ for every $u \in Z$ and $\beta'(u) = \beta(u)$ for every $u \notin Z$. Let us also specify all protocols ALG_u for all $u \in Z$ to act the same as in $\eta^{\text{ALG}}(\mathcal{A})$ until round $t-1$. At time t all new protocols are forced to resign. The executions remain the same until round $t+1$, when v raises the bid to $B(v,t+1)$. After this point, no participant raises the bid again, and v ends up winning the auction with $B(v,t+1)$ instead of the current $M_2'+1 = B(w,t)+1$. This contradicts the prudentiality of ALG_v, since v could have won the auction by offering exactly $B(w,t)+1$. ∎

Corollary 1. *In a limited and prudent auction protocol* ALG *on an auction system* $\mathcal{A}$*, the auctioneer must receive a bid* $B(v,t) = x$ *by some* v*, at some time during the execution, for every* $1 \leq x \leq M_2 + 1$. ∎

Lemma 2. *In a limited and prudent auction protocol* ALG *on an auction system* $\mathcal{A}$*, there cannot be a round* t *in which the auctioneer receives simultaneously two different new bids higher than* B_{t-1}.

Proof. Suppose, for the sake of contradiction, that there exists an execution $\eta^{\text{ALG}}(\mathcal{A})$ in which on round t in the auction the highest published offer is B_{t-1} and the auctioneer received $B(w,t)$ and $B(v,t)$ where $B_{t-1} < B(w,t), B(v,t)$ and also $B(w,t) \neq B(v,t)$. Without loss of generality, assume that $B(w,t) > B(v,t)$. But this implies that w's protocol is not prudent by Lemma 1; contradiction. ∎

Corollary 2. *For any prudent and limited protocol* ALG *for a given auction system* $\mathcal{A}$*,* $T_{\text{ALG}}(\mathcal{A}) = \Omega(M_2)$ *and* $C_{\text{ALG}}(\mathcal{A}) = \Omega(M_2 + n)$.

Proof. Since there must be a separate bid, at a separate time, for each possible value between 1 and M_2 (Lemma 1, Corollary 1 and Lemma 2), $T_{\text{ALG}}(\mathcal{A}) = \Omega(M_2)$ and $C_{\text{ALG}}(\mathcal{A}) \geq \Omega(M_2)$. The bidding may stop only after all but one of the participants have resigned, hence $C_{\text{ALG}}(\mathcal{A}) \geq \Omega(n)$. Combined together, $C_{\text{ALG}}(\mathcal{A}) = \Omega(M_2 + n)$. ∎

Remark: The lower bound on communication complexity does not hold if the underlying communication network is synchronous, as employing the well-known idea of using *silence* to convey information, it is possible to devise an algorithm based on interpreting clock-ticks as active bids, with time complexity $O(M_2)$ and communication complexity $O(n)$ (see [2]).

Next, it is proven that under a prudent auction-protocol, the same participant cannot bid twice in a row unless someone else bid in between.

Lemma 3. *Under a limited and prudent auction protocol* ALG *on a given system* $\mathcal{A}$*, if the auctioneer receives two different consecutive bids* $B(w,t) = B$ *and* $B(w,t+1) = B+1$ *from the same participant* w*, then it must have received a bid of* B *from another participant on round* t*.*

Proof. Suppose that there is an execution in which participant w offers the bids $B(w,t) = B$ followed by $B(w,t+1) = B+1$, with no one else bidding in round t. Let Z be the set of participants whose valuation exceeds or equals $B(w,t)$, i.e.,

$$Z = \{v \neq w \mid \beta(v) \geq B(w,t)\}.$$

Consider a different auction system $\mathcal{A}' = \langle \beta', n \rangle$, where $\beta'(u) = B(w,t) - 1$ for every $u \in Z$ and $\beta'(u) = \beta(u)$ for every $u \notin Z$. Note that in $\mathcal{A}'$ the new valuations satisfy $M_1 \geq B(w,t)+1$ and $M_2 = B(w,t)-1$. However, the execution $\eta^{\text{ALG}}(\mathcal{A}')$ and the bidding stay the same as in $\eta^{\text{ALG}}(\mathcal{A})$ up until time t, when w raised the bid to $B(w,t)$ followed by $B(w,t)+1$. Following this point, no participant will raise the bid, and w will end up winning this auction with $B(w,t)+1$. That contradicts the bidder-prudentiality of the auction protocol, since w could have won it with the first offer of $B(w,t)$ made at time t. ∎

Remark: The algorithmic implementations of English auction discussed in the next section enforce the requirement implied by the last lamma by ensuring that the designated winner of round t, W_t, is not approached again by the auctioneer before getting a higher bid from someone else.

Finally, we point out that the global lower bounds of Cor. 2 are easily achieved by an offline algorithm OPT.

Lemma 4. *For any auction system* $\mathcal{A} = \langle \beta, n \rangle$*, an offline algorithm* OPT *can perform an auction optimally in both communication and time, i.e.,* $T_{\text{OPT}}(\mathcal{A}) = O(M_2)$ *and* $C_{\text{OPT}}(\mathcal{A}) = O(n + M_2)$*.*

Proof. Assume that v_1 and v_2 are the participants with valuations $\beta(v_1) = M_1$ and $\beta(v_2) = M_2$. Then the optimal algorithm OPT runs an auction between these two participants until v_2 resigns when the bid reaches $B = M_2 + 1$. This takes $T_{\text{OPT}}(\mathcal{A}) = C_{\text{OPT}}(\mathcal{A}) = O(M_2)$. Now OPT addresses all other participants in a round of broadcast and convergecast, receiving simultaneously the remaining $n-2$ resignation messages. This takes two additional time units and $2(n-2)$ messages. Overall, $T_{\text{OPT}}(\mathcal{A}) = O(M_2)$ and $C_{\text{OPT}}(\mathcal{A}) = O(n + M_2)$. ∎

3 Auctioning algorithms

3.1 Set algorithms

All our algorithms belong to a class termed *set algorithms*, which can be cast in a uniform framework as follows. On each round t of the execution, the query set selected by the auctioneer is an *arbitrary* subset of active participants, $\sigma_t \subseteq AP_t \setminus \{W_t\}$, of size ρ_t. The only difference between the various set algorithms is in the *size* ρ_t fixed for the query set σ_t on each round t.

We develop our optimal algorithm through a sequence of improvements. Our first two simple algorithms represent two extremes with opposing properties; the first is communication optimal, the other - time optimal.

The Singleton algorithm (SINGL) Algorithm *singleton* (SINGL) is simply the *sequential* set algorithm with $\rho_t = 1$. Specifically, at every time $t \geq 1$, the adversary chooses a query set $\sigma_t = \{v\}$ for some $v \in AP_t \setminus \{W_t\}$. The participant v chosen for being queried on each round can be selected in round-robin fashion, although this is immaterial, and an arbitrary choice will do just as well.

For estimating the communication and time complexities of Algorithm SINGL, note that the auctioneer needs to receive resignations from all participants but one, and also raise the bid up to $M_2 + 1$, getting $C_{\text{SINGL}}(\mathcal{A}) = T_{\text{SINGL}}(\mathcal{A}) = O(M_2 + n)$.

Algorithm FULL At the other extreme, Algorithm FULL requires the auctioneer to address, on each round t, *all* active participants (except for the node W_t designated as the winner of the previous bidding round). I.e., it selects $\rho_t = |AP_t| - 1$ and hence $\sigma_t = AP_t \setminus \{W_t\}$.

Lemma 5. *For any auction system* $\mathcal{A} = \langle \beta, n \rangle$, $T_{\text{FULL}}(\mathcal{A}) = O(M_2)$ *and* $C_{\text{FULL}}(\mathcal{A}) = O(nM_2)$.

Proof. For analyzing the time complexity of the algorithm, assume that v_1 and v_2 hold valuations $\beta(v_1) = M_1$ and $\beta(v_2) = M_2$ respectively. Since Algorithm FULL addresses either v_1 or v_2 (and sometimes both) on each round, the auction will end after exactly $M_2 + 1$ rounds. Thus $T_{\text{FULL}}(\mathcal{A}) = O(M_2)$.

As for the communication complexity, on each round t the auctioneer communicates with all participants in AP_t. Hence over all $M_2 + 1$ rounds, the algorithm incurs $C_{\text{FULL}}(\mathcal{A}) = \sum_{t=1}^{M_2+1} |AP_t|$ messages. As $|AP_t| \leq n - 1$ for every t, we have $C_{\text{FULL}}(\mathcal{A}) = O(nM_2)$. ∎

We note that both bounds are tight for Algorithm FULL, as can be seen by considering an auction system $\mathcal{A} = \langle \beta, n \rangle$ where $\beta(v) = M_1$ for every node v, in which all participants are active on every round.

The following subsections are devoted to the development of successively improved set algorithms based on some intermediate versions between Algorithms SINGL and FULL.

3.2 The fixed size scheme (FSS)

Algorithm FIXED SIZE SCHEME (FSS) is based on an intermediate point between Algorithms SINGL and FULL, represented by a fixed integer parameter $\rho \geq$. In each round t, the auctioneer addresses an arbitrary set of $\rho_t = \rho$ active participants. (Once the number of remaining active participants falls below ρ, it addresses all of them.) Namely, the query set on round t is some arbitrary $\sigma_t \subseteq AP_t \setminus \{W_t\}$ of size $\rho_t = \min\{\rho, |AP_t| - 1\}$.

Lemma 6. *For any fixed integer parameter $\rho \geq 1$ and auction system $\mathcal{A} = \langle \beta, n \rangle$, $T_{\mathrm{FSS}}(\mathcal{A}) = O(M_2 + n/\rho)$ and $C_{\mathrm{FSS}}(\mathcal{A}) = O(M_2 \cdot \rho + n)$.*

Proof. The algorithm requires exactly $M_2 + 1$ bid-increase rounds to reach the final bid. In addition, there may be at most n/ρ rounds t in which the auctioneer receives resignations from all the participants of the query set σ_t, hence gaining no bid increase. Overall, this yields a time complexity of $T_{\mathrm{FSS}}(\mathcal{A}) = O(M_2 + n/\rho)$.

The communication complexity is bounded by noting that in each time step, the algorithm incurs (at most) ρ messages, hence $C_{\mathrm{FSS}}(\mathcal{A}) \leq T_{\mathrm{FSS}}(\mathcal{A}) \cdot \rho = O(M_2 \cdot \rho + n)$. ∎

Again, the analysis is tight, as evidenced by an auction system $\mathcal{A} = \langle \beta, n \rangle$ where $\beta(v) = M_1$ for every node v.

3.3 The increasing size scheme (ISS)

Examining the communication and time performance of Algorithm FSS reveals that using a large ρ value is better when $M_2 < n$, and on the other hand, if $M_2 \geq n$ then a small ρ value is preferable. The break point between the two approaches is when $M_2 = n$.

We next devise a set algorithm called INCREASING SIZE SCHEME (ISS), which exploits this behavior by using a decreasing value of the parameter ρ, inversely proportional to M_2. Since the value of M_2 is unknown, it is estimated by the only estimate known to us, namely, the current bid B. For simplicity, let us ignore rounding issues by assuming w.l.o.g. that n is a power of 2. The algorithm begins with $\rho = n$, and divides ρ by 2 whenever the current bid B_t doubles. I.e., ρ is set to $2^{\log n - i}$ once B_t reaches 2^i. Once $B = n$, Algorithm ISS continues as the sequential Algorithm SINGL until the auction is over.

Lemma 7. *For any auction system $\mathcal{A} = \langle \beta, n \rangle$, $T_{\mathrm{ISS}}(\mathcal{A}) = O(M_2)$.*

Proof. Each phase i of Algorithm ISS starts with a bid value of $\hat{B}_{i-1} = 2^{i-1}$ and ends when either all participants have resigned or Algorithm ISS reaches a bid of $\hat{B}_i = 2^i$, whichever comes first. Phase i is therefore similar to a run of Algorithm FSS with sets of size $\rho^i = n/2^i$, an initial bid of $\hat{B}_{i-1} = 2^{i-1}$ and a maximal bidding value of $x_i = \min\{M_2, 2^i\}$, or equivalently, an initial bid of 1

and a maximal bidding value of $M_2^i = x_i - 2^{i-1} \leq 2^{i-1}$. Hence each such phase i takes time $T_i = 2(M_2^i + n/\rho^i) \leq 2(2^{i-1} + 2^i) < 2^{i+2}$.

Let us first consider the case of an auction system $\mathcal{A}$ with $M_2 < n$. Then Algorithm ISS reaches at most phase $I_f = \lceil (\log M_2 + 1) \rceil$, where it will reach the final bid of $\hat{B}_i = M_2 + 1$. The total time for all phases is therefore

$$T_{\mathrm{ISS}}(\mathcal{A}) \;\leq\; \sum_{i=0}^{I_f} T_i \;\leq\; \sum_{i=0}^{\lceil \log(M_2+1) \rceil} 2^{i+2} \leq 4 \cdot 2^{\lceil \log(M_2+1) \rceil} = O(M_2).$$

Now assume that $M_2 > n$. Then the execution of Algorithm ISS has $\log n$ phases. The first $\log n - 1$ phases, as well as the steps of the last phase up to the point when $B_t = n$, take $O(2^{\log n}) = O(n)$ just as in the previous case. The remaining steps are performed in the sequential fashion of Algorithm SINGL, starting at $B = n$ and ending at M_2, thus including (at most) $M_2 - n$ bidding rounds and n resignation rounds. The total time complexity is again $T_{\mathrm{ISS}}(\mathcal{A}) = O(M_2)$. ∎

Lemma 8. *For any auction system $\mathcal{A} = \langle \beta, n \rangle$, $C_{\mathrm{ISS}}(\mathcal{A}) = O(M_2 + n \log \mu)$ where $\mu = \min\{M_2, n\}$.*

Proof. As in the proof of Lemma 7, every phase $i \leq \log n$ in the execution of Algorithm ISS can be regarded as a complete execution of Algorithm FSS starting from some appropriate initial state with maximal valuation M_2^i. Hence, as shown in the proof of Lemma 6, the bound on the bidding communication in a single phase i of Algorithm ISS is $C_i^{bids} = O(\rho^i M_2^i) \leq O(2^{i-1} \frac{n}{2^i}) = O(n)$. The resignations throughout the auction require $O(n)$ additional messages.

Again the analysis is divided into two cases. When $M_2 < n$, Algorithm ISS stops at phase $I_f = \lceil \log(M_2 + 1) \rceil$. Thus over all I_f phases, the number of messages incured by Algorithm ISS is

$$C_{\mathrm{ISS}}(\mathcal{A}) = \sum_{i=1}^{I_f} C_i = \sum_{i=1}^{\lceil \log(M_2+1) \rceil} O(n) = O(n \log M_2).$$

On the other hand, when $M_2 > n$, Algorithm ISS performs $\log n$ phases at a cost of $O(n)$ messages each, as shown above, summing up to $O(n \log n)$ messages. The remaining steps are as in Algorithm SINGL, starting at an initial bid of $B_t = n$ and ending at $B + t = M_2 + 1$, at a cost of $O(M_2 - n)$ messages. In total, $C_{\mathrm{ISS}}(\mathcal{A}) = O(M_2 + n \log n)$.

Thus the communication complexity of Algorithm ISS in the general case is $C_{\mathrm{ISS}}(\mathcal{A}) = O(M_2 + n \log \mu)$. ∎

3.4 The varying size scheme (VSS)

The ensuing discussion reveals the following property for set algorithms. Whenever many participants are willing to raise the bid, it is preferable to address

a small set σ_t. On the other hand, if there are many resignations then it is advantageous if the approached set σ_t is large.

The VARYING SIZE SCHEME (VSS) algorithm attempts to exploit this property by varying the size of the addressed set *dynamically*. As in Algorithm ISS, phase k of the algorithm tries to double the bid B from 2^{k-1} to 2^k. However, the exact size of the set σ_t fluctuates dynamically during the phase. The initial set size used for the first round of phase k is set[1] to $\rho^k = \frac{n}{2^k}$. After each round t, the set size used, denoted $\rho_t = |\sigma_t|$, is either *doubled* or *halved*, according to the following rules.

1. If the bid was raised in round t, and $\rho_t > 1$, then ρ_t is decreased by half, i.e., $\rho_{t+1} \leftarrow \max\{\rho_t/2, 1\}$.
2. If all the participants of σ_t have resigned, and $\rho_t < |AP_{t+1}|$, then ρ_t is doubled, setting $\rho_{t+1} \leftarrow \min\{2\rho_t, |AP_{t+1}|\}$.

When the bid B reaches n, Algorithm VSS continues as in the sequential Algorithm SINGL.

We now proceed with an analysis of the time and communication complexities of Algorithm VSS. The set of rounds in phase k, denoted $\mathcal{T}_k$, can be divided into $\mathcal{T}_k = U_k \cup D_k$. The set D_k consists of the *down steps*, which are rounds t in which the bid was increased, resulting in halving ρ for the next round (or leaving it at the minimal size of 1). The set U_k contains the *up steps*, which are rounds t in which all members of σ_t resigned, resulting in doubling ρ for round $t+1$ (or leaving it at the maximal available size at that round, which is $|AP_{t+1}|$).

The set U_k may be split further into two kinds of up steps.

1. U_k^s is the set of steps t where $\rho_t < \rho^k = \frac{n}{2^k}$.
2. U_k^l is the set of steps t where $\rho_t \geq \rho^k$.

Likewise, the set D_k can be divided further into two subsets of down steps. The first set, denoted by D_k^f, is the set of all down steps t which address a set of size $\rho_t = \frac{n}{2^j}$ where $j \geq k$ *for the first time* during the phase (i.e., there was *no* prior up step from the same set size). Formally,

$$D_k^f = \{t \in D_k \mid \rho_t = \frac{n}{2^j} \text{ for some } j \geq k, \text{ and } \rho_{t'} \neq \rho_t \text{ for every } t' \in \mathcal{T}_k \text{ s.t. } t' < t\}.$$

The remaining down steps are denoted $D_k^p = D_k \setminus D_k^f$.

Lemma 9. *Each phase k takes $T_k = O(2^k)$ steps.*

Proof. As $|D_k|$ equals the number of bid raises in phase k, which is at most 2^k, we have

$$|D_k| \leq 2^k. \tag{1}$$

[1] again assuming n to be a power of 2

Also, each step $t \in U_k^s$ increases the set size from ρ_t to $2\rho_t \leq \rho^k$. Thus, there must have been a corresponding prior down step t' that decreased a set of size $2\rho_{t'} = 2\rho_t$ to the size of $\rho_{t'} = \rho_t$. Hence

$$|U_k^s| \leq |D_k|,$$

so also

$$|U_k^s| \leq 2^k. \tag{2}$$

Finally, U_k^l consists of resignation steps that occurred on sets σ_t of size $\rho_t \geq \frac{n}{2^k}$, hence there can be at most 2^k steps of that sort before removing all possible n participants, implying that

$$|U_k^l| \leq 2^k. \tag{3}$$

Combining Inequalities (1), (3) and (2), we get that

$$T_k = 2(|U_k^l| + |U_k^s| + |D_k|) \leq 6 \cdot 2^k. \qquad \blacksquare$$

Let us now estimate the time complexity of Algorithm VSS.

Lemma 10. *For any auction system $\mathcal{A} = \langle \beta, n \rangle$, $T_{\mathrm{VSS}}(\mathcal{A}) = O(M_2)$.*

Proof. First suppose $M_2 \leq n$. Then Algorithm VSS is run for $\lceil \log(M_2 + 1) \rceil$ phases, and by Lemma 9 the total time is

$$T_{\mathrm{VSS}}(\mathcal{A}) \;=\; \sum_{k=0}^{\lceil \log(M_2+1) \rceil} T_k \;=\; \sum_{k=0}^{\lceil \log(M_2+1) \rceil} O(2^k) \;=\; O(M_2).$$

In case $M_2 > n$, the $\log n$ phases take $O(n)$ time by the same calculation. Afterwards, Algorithm VSS operates as Algorithm SINGL for the remaining $M_2 - n$ bids. The time required for this stage is $M_2 - n$ steps, plus (at most) n additional steps for resignations. Thus, overall, $T_{\mathrm{VSS}}(\mathcal{A}) = O(M_2)$. $\blacksquare$

Next we deal with the communication complexity of Algorithm VSS. Note that the communication cost of round t is $2\rho_t$, and therefore a set of rounds X costs $2\sum_{t \in X} \rho_t$. Let C_k denote the total communication complexity of phase k, and let n_k^R denote the number of participants which resigned during phase k. Let $C^{all} = \sum_{k=1}^{\log n} C_k$ and $n^R = \sum_{k=1}^{\log n} n_k^R$. Let C^S denote the communication cost of the final sequential stage of the algorithm.

Lemma 11. $C^{all} \leq 4n + 6n^R$.

Proof. Let C_k^u, C_k^f and C_k^p denote the total amount of communication due to rounds t in U_k, D_k^f and D_k^p, respectively, hence $C_k = C_k^u + C_k^f + C_k^p$. We analyze each of the three terms separately.

Clearly, $C_k^u \leq 2n_k^R$. Turning to the communication cost of rounds in D_k^f, note that any round $t \in D_k^f$ in phase k involves a set σ_t of size $\rho_t = \frac{n}{2^j}$ where $j \geq k$. Moreover, D_k^f contains at most one round t such that $\rho_t = \frac{n}{2^j}$ for every

$\log n \geq j \geq k$. Thus, the number of rounds t in which $\rho_t = \frac{n}{2^j}$ throughout the entire execution of Algorithm VSS is bounded by j (as after reaching phase j, $\rho^k < \frac{n}{2^j}$, hence sets of size $\frac{n}{2^j}$ will not be included in D_k^f). Thus the total communication cost due to D_k^f steps is

$$C_k^f \;=\; 2\sum_{t\in D_k^f}\rho_t \;\leq\; \sum_{j=k}^{\log n}\frac{n}{2^j} \;<\; \frac{n}{2^{k-2}}\sum_{j=1}^{\infty}\frac{1}{2^j} \;=\; \frac{n}{2^{k-2}}\,.$$

Finally consider C_k^p. We argue that each round $t \in D_k^p$ can be matched with a *distinct* prior resignation step in the same phase, $r(t) \in U_k$. Specifically, $r(t)$ is the largest round $t' < t$ satisfying $t' \in U_k$ and $\rho_{t'} = \frac{\rho_t}{2}$. Note that U_k must contain such a round t', since by definition of D_k^p, there exists some round $t'' < t'$, $t'' \in D_k^p$, with $\rho_{t''} = \rho_t$. On round $t''+1$, $\rho_{t''+1} = \frac{\rho_{t''}}{2}$. Hence, the fact that by time t the size of σ_t went back to $\rho_{t''}$ implies that there must have been an intermediate step $t'' < t' < t$ in which the algorithm performs an up step bringing σ_t back to size $\rho_{t''}$. Note also that by the definition, $r(t)$ is unique for every t. It follows that

$$C_k^p \;=\; 2\sum_{t\in D_k^p}\rho_t \;=\; 2\sum_{t\in D_k^p}2\rho_{r(t)} \;\leq\; 4\sum_{t'\in U_k}\rho_{t'} \;=\; 2C_k^u \;=\; 4n_k^R.$$

It follows from the preceding three bounds that $C_k < n/2^{k-2} + 6n_k^R$. Subsequently,

$$C^{all} \;=\; \sum_{k=0}^{\log n} C_k \;<\; \sum_{k=0}^{\log n}\left(\frac{n}{2^{k-2}} + 6n_k^R\right) \;<\; \sum_{k=0}^{\infty}\frac{n}{2^{k-2}} + 6n^R \;=\; 4n + 6n^R\,. \qquad \blacksquare$$

Lemma 12. *When $M^2 > n$, $C^S \leq 2(M_2 - n^R)$.*

Proof. The communication incurred by the final SINGL stage for increasing the bid from n to $M_2 + 1$ is $O(M_2 - n)$. As each participant resigns at most once, Algorithm VSS also incurs (at most) $n - n^R - 1$ resignations during that stage, one for each of the $n - n^R$ remaining active participants except for the winner. $\blacksquare$

Corollary 3. *For any auction system $\mathcal{A} = \langle \beta, n\rangle$, $C_{\mathrm{VSS}}(\mathcal{A}) \leq 2M_2 + 8n$.*

Proof. For $M_2 \leq n$, $C_{\mathrm{VSS}}(\mathcal{A}) = C^{all} = 4n + 6n^R$ by Lemma 11. For the case of $M_2 > n$, we have in addition to that also a cost of $C^S = 2M_2 - 2n^R$. In total, $C_{\mathrm{VSS}}(\mathcal{A}) \leq 2M_2 + 4n + 4n^R \leq 2M_2 + 8n$. $\blacksquare$

Theorem 1. *For any auction system $\mathcal{A} = \langle \beta, n\rangle$, Algorithm* VSS *achieves asymptotically optimal complexities $T_{\mathrm{VSS}}(\mathcal{A}) = O(M_2)$ and $C_{\mathrm{VSS}}(\mathcal{A}) = O(n + M_2)$.* $\blacksquare$

3.5 Allowing dynamic participants

One of the limitations of Algorithm VSS is that it requires all the participants to register in advance, since the beginning of the auction (as the algorithm takes the number of participants into account when deciding its querying policy). In actual computerized auctions, it is desirable to allow new participants to join in, so long as the auction has not terminated. In this section we extend our framework by allowing newcomers to join the auction. Assume there is some sort of *bulletin board* on which the process announces the auction and the current bid, and also that it has a mailbox in which it may receive requests to join the auction.

For simplicity, the process will grant these requests only at the end of an auction (where only one participant left). Thus the entire auction process is composed of a sequence of auctions, viewed as sub-auctions of the entire auction, each starting upon termination of the previous one, until no further requests to join arrive.

More specifically, the auctioneer acts as follows. It starts by running a first sub-auction on the initial set of participants, V^1. This sub-auction is run until all participants but one have resigned, and the current bid is $B_1 = M_2^1 + 1$, made by participant v_{i_1}. The auctioneer now opens the mailbox and reviews the requests to join. Let V^2 denote the set of newcomers asking to join the auction. The auctioneer now starts a sub-auction on $V^2 \cup \{v_{i_1}\}$, starting from the initial bid $B_1 + 1$, rather than 1. (This may well be transparent to v_{i_1} itself, i.e., there is no need for it to know that the first sub-auction has finished and a new sub-auction has begun.) This second sub-auction terminates with some participant v_{i_2} making a bid of $B_2 = M_2^2 + 1$, where M_2^2 is defined as the second highest valuation on $V^2 \cup \{v_{i_1}\}$. Once this sub-auction has terminated, the auctioneer again inspects its mailbox, and if there are additional newcomers then it repeats the process. Hence the entire auction process finalizes only once a sub-auction ends and no additional requests to join have arrived, i.e., the mailbox is empty. Evidently, the execution may be quite long (or even infinite, assuming the value of the sold item keeps raising!) In practice, however, the number of rounds in the auction will be bounded by the value of the sold item, which is presumably stable over short periods of time.

Formally, the extended auction system can be represented by an initial auction system $\mathcal{A}_1 = \langle \beta_1, n_1 \rangle$ and a sequence of *extensions*, $\mathcal{A}_2, \mathcal{A}_3, \ldots, \mathcal{A}_p$. An *extension* represents an entry point of new participants, described by a pair $\mathcal{A}_i = \langle \beta_i, n_i \rangle$, where n_i is the number of newcomers at the entry point, denoted $V^i = \{v_1^i, \ldots, v_{n_i}^i\}$, and β_i is their valuation function. We also denote by M_2^i the second highest valuation among the newcomers in the ith extension and the winner of the previous sub-auction. Note that the initial bid for the $(i+1)$st sub-auction is $B_{i+1} = M_2^i + 1$. Note also that M_2^i is the second highest valuation over $\bigcup_{j=1}^{i} V^j$.

To handle this type of extended auction system we present Algorithm EXTENDED VARYING SIZE SCHEME (EVSS), which acts as follows. The initial sub-auction is handled the same as in Algorithm VSS. Each of the following sub-auctions is again managed by a variant of Algorithm VSS. As explained above,

this variant starts the bidding process at B, the highest bid of the previous sub-auction. However, the algorithm must shift the bids by B for the purpose of computing the size ρ_t of the query set in each round t. That is, phase k of the algorithm must now be defined as the phase during which increases the bid from $B + 2^{k-1}$ to $B + 2^k$ (rather than as the phase during which the bid is *doubled* from 2^{k-1} to 2^k).

It is straightforward to verify that the analysis of Algorithm VSS in the previous section goes through, and guarantees the following.

Lemma 13. *For $i \geq 1$, the ith sub-auction requires $T_{\mathrm{EVSS}}(\mathcal{A}_i) = O(M_2^i - M_2^{i-1})$ (denoting $M_2^0 = 0$) and $C_{\mathrm{EVSS}}(\mathcal{A}_i) = O(M_2^i - M_2^{i-1} + n_i)$.*

For an extended auction system $\bar{\mathcal{A}} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_p \rangle$, let $n = \sum_{i=1}^{p} n_i$ and define M_2^i as above. Recall that M_2^p is the second highest valuation over the set of all participants throughout the entire auction process, $\bigcup_{j=1}^{p} V^j$. We have the following.

Theorem 2. *For any extended auction system $\bar{\mathcal{A}} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_p \rangle$, $T_{\mathrm{EVSS}}(\bar{\mathcal{A}}) = O(M_2^p)$ and $C_{\mathrm{EVSS}}(\bar{\mathcal{A}}) = O(n + M_2^p)$.*

Let us remark that a more flexible variant of Algorithm EVSS, allowing various entry points in the middle of a sub-auction, rather than only at the end of each sub-auction, is described in [2], and is shown to enjoy the same asymptotic complexities.

References

1. Going... Going... Gone! A survey of auction types. *Agorics Inc.*, 1996. Can be found at http://www.agorics.com/
2. Yedidia Atzmony. Distributed Algorithms for English Auctions. M.Sc. Thesis, The Weizmann Institute of Science, Mar. 2000.
3. C. Beam and A. Segev. *Auctions on the Internet: A Field Study.* Univ. of California, Berkeley; Can be found at http://haas.berkeley.edu/ citm/research/ under *publications*
4. C. Beam, A. Segev and G.J. Shanthikumar. *Electronic Negotiation through Internet-Based Auctions.* CMIT Working Paper 96-WP-1019, Dec-96; Can be found at http://haas.berkeley.edu/ citm/research/ under *publications*
5. R. Cassady. *Auctions and Auctioneering.* Univ. of California Press, 1967.
6. R.P. McAfee and J. McMillan. Auctions and Bidding. *J. Economic Literature* (1987), 699–738.
7. M. Naor, B. Pinkas and R. Sumner. Privacy Preserving Auctions and Mechanism Design. *Proc. 1st ACM conf. on Electronic Commerce*, Denver, Colorado, Nov. 1999.
8. M. Shubik. Auctions, Bidding and Markets: An Historical Sketch. in R. Engelbrecht-Wiggans, M. Shubik and R.M. Stark (Eds), *Auctions, Bidding, and Contracting: Uses and Theory*, New York university press, New York, 1983, pp. 165–191.
9. E. Wolfstetter. Auctions: An introduction. *J. Economic Surveys* **10**, (1996), 367–417.

A Probabilistically Correct Leader Election Protocol for Large Groups

Indranil Gupta, Robbert van Renesse, and Kenneth P. Birman

Cornell University, Ithaca, NY 14850, USA
{gupta, rvr, ken}@cs.cornell.edu

Abstract. This paper presents a scalable leader election protocol for large process groups with a weak membership requirement. The underlying network is assumed to be unreliable but characterized by probabilistic failure rates of processes and message deliveries. The protocol trades correctness for scale, that is, it provides very good probabilistic guarantees on correct termination in the sense of the classical specification of the election problem, and of generating a constant number of messages, both independent of group size. After formally specifying the probabilistic properties, we describe the protocol in detail. Our subsequent mathematical analysis provides probabilistic bounds on the complexity of the protocol. Finally, the results of simulation show that the performance of the protocol is satisfactory.

Keywords: Asynchronous networks, process groups, leader election, fault-tolerance, scalable protocols, randomized protocols.

1 Introduction

Computer networks are plagued by crashing machines, message loss, network partitioning, etc., and these problems are aggravated with increasing size of the network. As such, several protocol specifications are difficult, if not impossible, to solve over large-scale networks. The specifications of these protocols, which include reliable multicast, leader election, mutual exclusion, and virtual synchrony, require giving strong deterministic correctness guarantees to applications. However, in results stemming from the famous Impossibility of Consensus proof by Fischer-Lynch-Paterson [8], most of these problems have been proved to be unsolvable in failure-prone asynchronous networks. Probabilistic and randomized methodologies are increasingly being used to counter this unreliability by reducing strict correctness guarantees to probabilistic ones, and gaining scalability in return. A good example of such a protocol is the Bimodal Multicast protocol [1], an epidemic protocol that provides only a high probability of multicast delivery to group members. In exchange, the protocol gains scalability, delivering messages at a steady rate even for large group sizes.

* This work was funded by DARPA/RADC grant F30602-99-1-6532 and in part by the NSF grant No. EIA 97-03470..

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 89–103, 2000.

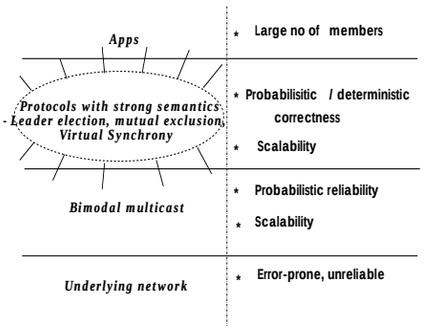


Fig. 1. Target Setting.

Our current work is targeted toward realizing similar goals for the important class of protocols that classically have been formulated over reliable multicast message delivery. We envision a world where applications would run over a new class of probabilistic protocols (Figure 1) and receive probabilistically guaranteed services from the layer below. By virtue of the proposed approach, these applications would scale arbitrarily, while guaranteeing correctness with a certain minimal probability even in the face of an unreliable network. For example, these protocols could be used to build a replicated file system with probabilistic guarantees on consistency.

As a step towards this goal, this paper presents a probabilistic leader election protocol. Leader election arises in settings ranging from locking and synchronization to load balancing [12] and maintaining membership in virtually synchronous executions [13]. The classical specification of the leader election problem for a process group states that at the termination of the protocol, exactly one non-faulty group member is elected as the leader, and every other non-faulty member in the group knows about this choice. In this paper, we show that, given probability guarantees on point-to-point (unicast) and multicast message delivery, process failure rates, and multicast group view content, our protocol gives a very high probability of correct termination. In return, it gains on the scalability: with very high probability, the protocol involves only a constant number of messages *regardless of group size*. We also show how to augment our protocol to adapt to changing failure probabilities of the network (w.r.t. processes and messages).

Sabel and Marzullo [20] proved that leader election over a failure-prone asynchronous network is impossible. This and a variety of other impossibility results all stem from the FLP result [8], which proves that there is no protocol by which an asynchronous system of processes can agree on a binary value, even with only one faulty process.

To provide a taxonomy of the complexity of the class of consensus protocols, Chandra and Toueg [4] proposed extending the network with *failure detectors*.

However, the leader election problem can be solved if and only if a perfect failure detector is available - one that suspects no alive processes, and eventually suspects every faulty one [20]. [6] discusses several weakened system models and what types of consensus are possible in these models, while [7] presents a weakened asynchronous model which assumes that message deliveries are always time-bounded. Since "real" systems lack such guarantees, these results have been valuable mostly in a theoretical rather than a practical sense.

Non-randomized leader election algorithms for a failure-prone asynchronous network model broadly fall into the following flavors. 1) Gallager-Humblet-Spira-type algorithms [9, 17] that work by constructing several spanning trees in the network, with a prospective leader at the root of each of these, and recursively reduce the number of these spanning trees to one. The correctness guarantees of these algorithms are violated in the face of pathological process and message failures. 2) Models that create logical partitions in the network when communication becomes unreliable, each logical partition electing one leader [7]. This approach does not solve the scalability problem but circumvents it. 3) Models that involve strong assumptions such as, for example, that all (process) failures occur before the election protocol starts [22], or that all messages are delivered reliably [3].

Probabilistic solutions to leader election in general networks are usually classified as randomized solutions to the consensus problem [5], but these focus on improving either the correctness guarantee [19], or the bound on the number of tolerated failures [23]. The (expected or worst case) message complexities in these algorithms are typically at least linear in the group size, and fault tolerance is usually guaranteed by tolerating process failures up to some fraction of the group size. Further, most of these protocols involve several rounds of $O(N)$ simultaneous multicasts to the group (where N is the group size), and this can cause deterioration of the delivery performance of the underlying network.

Our take on the leader election problem is in a more practical setting than any of the above cited works. We are motivated by practical considerations of scaling in a real network where failures can be characterized by probabilities. The spirit of our approach is close to that of [1] and [24]. Our protocol's probabilistic guarantees are similar to those of leader election algorithms for the perfect information model [14, 25], while our guarantee on the number of messages resembles that of [11], which presents an election protocol for anonymous rings. To the best of our knowledge, ours is the first protocol that trades correctness of the leader election problem for better scalability.

The analysis and simulation of our protocol will assume a network model where process failures, and message delivery latencies and statistics have identical, independent and uniform distributions. Before doing so, however, we suggest that the leadership election algorithm proposed here belongs to a class of gossip protocols, such as Bimodal Multicast [1], where such a simplified approach leads to results applicable in the real world. Although the model used in [1], like the one presented here, seems simplified and unlikely to hold for more than some percentage of messages in the network, one finds that in real-world scenarios,

even with tremendous rates of injected loss, delay, and long periods of correlated disruption, the protocol degrades gracefully in its probabilistic guarantees.

The rest of the paper is organized as follows. Section 2 describes the assumed model and statement of the election problem we solve. Section 3 describes the protocol in detail. Section 4 analyses the protocol mathematically, while Section 5 presents simulations results. In Section 6, we present our conclusions.

2 The Model and Problem

2.1 Model

In our model, all processes have unique identifiers (*e.g.*, consisting of their host address and local process identifier). All processes that might be involved in the election are part of a *group*, which can have an arbitrarily large number of members. Each process has a possibly incomplete list of other members in the group, called the process' *view*. A process can communicate to another process in its view by `ucast` (unicast, point-to-point) messages, as well as to the entire group by `mcast` (multicast) messages.

Processes and message deliveries are both unreliable. Processes can undergo only fail-stop failures, that is, a process halts and executes no further steps. Messages (either `ucast` or `mcast`) may not be delivered at some or all of the recipients. This is modeled by assuming that processes can crash with some probability during a protocol round and a `ucast` (`mcast`) message may not reach its recipient(s) with some probability. Probabilistically reliable multicast can be provided using an epidemic protocol such as Bimodal Multicast [1]. The Bimodal multicast protocol guarantees a high probability of multicast message delivery to all group members in spite of failures by having each member periodically gossip undelivered multicasts messages to a random subset of group members in its view.

A few words on the weak group model are in order. As we define them, views do not need to be consistent across processes, hence a pessimistic yet scalable failure detection service such as the gossip heartbeat mechanism of [24] suffices. New processes can join the group by multicasting a message to it, and receiving a reply/state transfer from at least one member that included it in its view.

Our analysis later in this paper assumes a uniform distribution for process failure probabilities (p_{fail}), `ucast`/`mcast` message delivery failure probabilities (p_{ucastl}/p_{mcastl}), as well as the probability that a random member has another random member in its view, which we call the view probability (*view_prob*).

2.2 Problem Statement

An election is initiated by an `mcast` message. This might originate from, say, a client who wants to access a database managed by the group, or one or more member(s) detecting a failure of a service or even the previous leader. In our discussion, we will assume only one initiating message, but the extension of our protocol to several initiating messages is not too difficult.

In classical leader election, after termination there is exactly one non-faulty process that has been elected leader, and all non-faulty processes know this choice. In probabilistic leader election, with *known high probability*,

- (*Uniqueness*) there is exactly one non-faulty process that considers itself the leader;
- (*Agreement*) all non-faulty group members know this leader; and
- (*Scale*) a round of the protocol involves a total number of messages that can be bounded independent of the group size.

3 Probabilistic Leader Election

This section describes the proposed leader election protocol. The protocol consists of several rounds, while each round consist of three phases. In Section 3.1, we present the phases in a round. In Section 3.2, we describe the full protocol and present its pseudo-code.

3.1 Phases in a Round of the Protocol

Filter Phase We assume that the initiating `mcast` I is uniquely identified by a bit string A_I. For example, A_I could be the *(source address, sequence number)* pair of message I, or the *(election #, round #)* pair for this election round. Each group member M_i that receives this message computes a *hash* of the concatenation of A_I and M_i's address, using a hash function H that deterministically maps bit strings to the interval $[0, 1]$. Next, M_i calculates the *filter value* $H(M_i A_I) \times N_i$ for the initiating message, where N_i is the size of (number of members in) M_i's current view. M_i participates in the next phase of this round, called the *Relay Phase*, if and only if this filter value is less than a constant K; otherwise it waits until the completion of the Relay phase. We require that H and K be the same for all members. We show in Section 4 that for a good (or fair) hash function H, large total number of group members N, the probability that the number of members throughout the Relay phase lies in an interval near K, quickly goes to unity at small values of K. This convergence is independent of N and is dependent only on the process failure, message delivery failure and view probabilities.

If the N_i's are the same for all members, each member M_i can calculate the set of members $\{M_j\}_i$ in its view that will satisfy the filter condition. It does so by checking if $H(M_j A_I) \times N_i < K$ for each member M_j in its view. In practice, the N_i's may differ, but this will not cause the calculated set $\{M_j\}_i$ to differ much from the actual one. (A more practical approach is to use an approximation of the total number of group members for N_i. This can be achieved by gossiping this number throughout the group. Thus, N_i's of different members will be close and the above filter value calculation will be approximately consistent.)

Figure 2 shows an illustration of one protocol round. The initiating multicast I is multicast to the entire group, but some group members may not receive it since `mcast` delivery is unreliable. The ones who do receive it evaluate the filter condition in the next step. The members labeled with solid circles (2, 3, N) find this condition to be true and hence participate in the Relay phase.

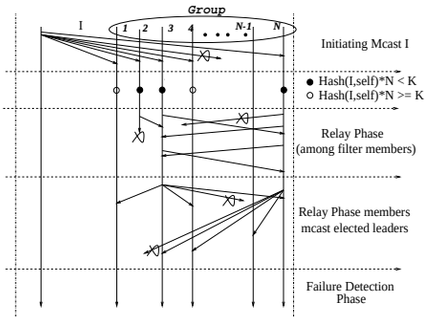


Fig. 2. One (Successful) Protocol Round.

Relay Phase As explained earlier, a member M_i that has passed the filter and is participating in the Relay phase can calculate the subset of members $\{M_j\}_i$ in its view that would have passed the filter condition if they received I. In the Relay phase, M_i first sends `ucast` messages to all such members in the set $\{M_j\}_i$ specifying M_i's preferred choice for a leader from among its view members. This choice is determined by the ordering generated by a *choice function* which evaluates the advantages of a particular member being elected leader. We require no restriction on the particular choice functions used, although all members need to use the same choice function in evaluating their preferred leaders, breaking ties by choosing the process with a lower identity. A good choice function would account for load, location, network topology, etc. [21].

Second, whenever M_i is contacted by another member M_k in a similar manner, it includes M_k in its view (and adds it to $\{M_j\}_i$), and compares M_k's choice with its own. If M_k's choice is "better" than its own according to the choice function, M_i *relays* this new choice to all the members in the set $\{M_j\}_i$ by `ucast` messages, and replaces its current best choice for leader. Otherwise, M_i replies back to M_k specifying its current best leader choice.

In the example of Figure 2, the 2^{nd}, 3^{rd} and N^{th} group members enter the Relay phase, but the 2^{nd} member subsequently fails. If either of the 3^{rd} and the N^{th} members has the other in its view, they will be able to exchange relay messages regarding the best leader.

Consider the undirected graph with nodes defined by the set of members participating in the Relay phase (relay members), and an edge between two members if and only if at least one of them has the other in its view throughout the phase. We call this the *relay graph*. Assuming timely message deliveries and no process failures, each connected component of this graph will elect exactly one leader, with a number of (`ucast`) messages dependent only on the size of the component. In Sections 4 and 5, we show that for a good hash function, the likelihood of the relay graph having exactly one component (and thus electing exactly one leader in the Relay Phase), approaches unity quickly at low values of K. Further, this convergence is independent of N and is dependent only on the

process failure, message delivery failure and view probabilities. In Section 5, for an example choice function that is widely used in many distributed systems, we show that message delivery and process failures do not affect this convergence. Note that the number of `ucast` messages exchanged in a Relay phase with m members is $O(m^3)$, since each relay member's best choice might be communicated to every other relay member.

Finally, at the end of the Relay phase, when each component has decided on one leader, each member M_i participating in the Relay phase multicasts the identifier of the leader selected by M_i's component (M_i's current best choice) to the entire group—this is the set of *final multicasts* of this election round. The total number of multicast messages in the Relay phase is thus $O(m)$. Since it is likely that m lies in an interval near the protocol parameter K which is chosen regardless of N (analysis of Section 4), this implies only a constant number of `ucast` and `mcast` messages in the Relay Phase with high probability, regardless of the value of N.

In the example of Figure 2, once the 3^{rd} and N^{th} members have agreed on a leader, each of them multicasts this information to the group. Some of the group members may not receive both multicasts, but it is unlikely that every non-faulty member will receive neither.

Failure Detection Phase Consider a situation in which there is more than one connected component in the relay graph. Each of these components may select and multicast different leaders in the Relay phase. Having each Relay phase member broadcast its component's selected leader to the entire group using a probabilistic multicast mechanism (such as Bimodal Multicast [1]) would give us a high probability that this inconsistency is detected by some group member (which need not have participated in the Relay phase). If a member detects an inconsistency such as two leaders elected in the same round, it immediately sends out a multicast to the entire group re-initiating the next election round. If no member detects any such inconsistency, the election protocol round would satisfy the Uniqueness and Agreement conditions of Section 2.2 if and only if there was exactly one component in the Relay phase, this component selected exactly one leader, every other non-faulty group member received at least one of the multicast messages specifying this selected leader, and this elected leader did not fail during the election round.

To reduce the probability of many group members sending out a (re-)initiating multicast message at the same time, we could have each member M_i calculate the hash (using H) of its own id concatenated with the message identifier of one of the resulting messages, and send out a re-initiating multicast only if this is lower than K/N_i. This would again give an expected constant number of re-initiating multicasts. Alternatively, we could use a randomized delay before sending the request: if a process receives a re-initiation request, it need not send one of its own.

Member M_i**::Election** $(Sequence, RoundNum)$**:**

1. On receiving "Init election" message I specifying $(Sequence, RoundNum)$,
   select K from $RoundNum$ using strategy
   if $H(M_i A_I) \times N_i < K$, go to step 2
   else wait for timeout period $Time\_Out\_1$ (time for step 2 to complete) and jump to step 3
2. Find the set of members $\{M_j\}_i$ in my view such that $H(M_j A_I) \times N_i < K$
   find best preferred leader in my view and send this using `ucast` messages to members in $\{M_j\}_i$
   do until $Time\_Out\_2$
      receive similar preferred leader messages for this $(Sequence, RoundNum)$ from
         other members M_k
      include M_k in $\{M_j\}_i$ and M_i's view
      compare current best leader choice with M_k's preference (using choice function)
      if M_k's preference better,
         update current best leader choice and send `ucast` messages to all members in $\{M_j\}_i$
            specifying this
      else
         inform M_k using a `ucast` of M_i's current best choice
   wait $Time\_Out\_3$ to receive everyone's final leader choice.

3. if received none or more than one leader as final choice,
      choose one of the final choice messages F
      if $H(M_i A_F) \times N_i < K$,
         multicast an initiating message I' specifying $(Sequence, RoundNum + 1)$
      wait for $Time\_Out\_3$, increment $RoundNum$ and jump to step 1
   if no re-initiating `mcast` received within another $Time\_Out\_3$,
      declare received choice as elected leader and include it in M_i's view
   else increment $RoundNum$ and jump to step 1.

Fig. 3. The Complete Election Protocol.

3.2 General Protocol Strategy

Figure 3 contains the pseudo-code for the steps executed by a group member M_i during a complete election protocol, each distinct election specified by a unique sequence number $SequenceNum$. Our complete election protocol strategy is to use the election round described in the previous section as a building block in constructing a protocol with several rounds. A complete protocol strategy specifies 1) the value of K to be used (by each member) in the first round of each election, 2) the value of K to be used in round number $l+1$ as a function of the value of K used in round l, and 3) a maximum number of rounds after which the protocol is aborted. Note that this strategy is deterministic and known to all members, and is not decided dynamically. In Figure 3, $RoundNum$ refers to the current round number in this election. $M_i :: Election(SequenceNum, 1)$ is called by M_i on receipt of the initiating message for that election protocol.

As we will see in Section 4, the initial value of K can be calculated from the required protocol round success probability, view probabilities, process and message delivery failure probabilities for the network in which the group members are based, and the total maximum number of group members. Unfortunately, in practice, failure probabilities may vary over time. Since a higher value of K leads to a higher probability of success in a round (Section 4), we conclude that round $l+1$ must use a higher value of K than round l. For example, one could use twice the value of K in round l, for round $l+1$. This class of strategies make our leader

election protocol adaptive to the unpredictability of the network. Note that a low maximum number of protocol rounds implies fewer expected messages while a higher value results in a better probability of correct termination.

The pseudo-code of Figure 3 has the members using time-outs (the $Time\_Out\_*$ values) to detect (or, rather, estimate) completion of a particular part of the protocol round in an asynchronous network.[1] $Time\_Out\_2$ is the expected time for termination of the Relay phase (before the final multicasts at the end). This is just the worst case propagation and processing delay needed for a message containing a relay member's initial preferred leader to reach all other relay members (if it is not lost by a process or link failure). Although the number of relay members is not known a priori, we show in Section 4 that with known high probability, the number of relay members who do not fail until the end of the Relay phase is at most (3K/2). Thus $Time\_Out\_2$ can be taken to be the product of $(3K/2)$ (the maximum length of any path in a relay graph with $3K/2$ members) and the maximum propagation and processing delay for a `ucast` packet in the underlying network. $Time\_Out\_3$ is just the worst case time for delivery of a `mcast` message. In the Bimodal multicast protocol [1], this would be the maximum time a message is buffered anywhere in the group. $Time\_Out\_1$ is the sum of the maximum time needed at member M_i to calculate the set $\{M_j\}_i$, and the values of $Time\_Out\_2$ and $Time\_Out\_3$. Also, a member ignores any messages from previous protocol rounds or phases, and "jumps ahead" on receiving a message from a future protocol round or phase.

4 Analysis - Properties of the Protocol

In this section, we summarize the analysis of the probability of success, detection on incorrect termination and message and time complexity of a round of our protocol. Detailed discussions and proofs of the results are available in [10].

Let N be the number of group members at the start of the election round - we will assume that this value is approximately known to all group members so that the filter value calculation is consistent across members. Let *view_prob* be the probability of any member M_i having any other member M_k in its view throughout the election round. Let p_{fail}, p_{ucastl}, p_{mcastl} be the failure probabilities of a process during an election round, a `ucast` message delivery, and a `mcast` message delivery, respectively. The protocol round analyzed uses the parameter K as in Figure 2. We denote the terms $((1-p_{mcastl})\cdot K)$ and $((1-p_{fail})\cdot(1-p_{mcastl})\cdot K)$ as K_1 and K_2 respectively.

For simplicity, we assume that the probabilities of deliveries of a `mcast` message at different receivers are independent, as well as that the $Time\_Out\_*$ values in the protocol of Figure 2 are large enough. Our analysis can be modified to omit the latter assumption by estimating the $Time\_Out\_*$ values from K and the worst-case propagation and processing delays for `ucast` and `mcast` messages (as

[1] Although an asynchronous network model does not admit real time, in practice timers are readily available, and we do not assume any synchronization nor much in the way of accuracy in measuring intervals.

described in Section 3.2), and redefining $p_{ucastl}(p_{mcastl})$ to be the failure probabilities of a `ucast` (`mcast`) message delivery within the corresponding worst-case delays, as well as calculating *view_prob*, p_{fail} for a round duration. We also assume that the hash function H used is fair, that is, it distributes its outputs uniformly in the interval $[0, 1]$. For a particular hash function (*e.g.*, the one described in [15]), we would need to know its distribution function and plug it into a similar analysis.

Consider the following events in an election round with parameter K:

E1: between $K_1/2$ and $3K_1/2$ members are chosen to participate in the Relay phase, and between $K_2/2$ and $3K_2/2$ relay members do not fail before sending out the final multicast;
E2: the set of relay members who do not fail before their final multicasts form a connected component in the relay graph throughout the Relay phase;
E3: at the end of the Relay phase, each non-faulty relay member has selected the same leader;
E4: by the end of the election round, each group member either fails or receives at least one of the final multicast messages (specifying the selected leader) from each component in the relay graph at the end of the Relay phase;
E5: the elected leader does not fail.

Theorem 1 (Round success probabilities):
(a) The event [E3, E4, E5] in an election round in the protocol of Figure 2 implies that it is successful, that is, the election satisfies the Uniqueness and Agreement properties of Section 2.2.

(b) From [2, 16], the probability of success in an election round with parameter K can be lower bounded by

$$\begin{aligned} & \mathtt{Pr}[\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}, \mathsf{E5}] \\ = \ & \mathtt{Pr}[\mathsf{E1}] \cdot \mathtt{Pr}[\mathsf{E2}|\mathsf{E1}] \cdot \mathtt{Pr}[\mathsf{E3}|\mathsf{E1}, \mathsf{E2}] \cdot \mathtt{Pr}[\mathsf{E4}|\mathsf{E1}, \mathsf{E2}, \mathsf{E3}] \cdot \mathtt{Pr}[\mathsf{E5}|\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}]. \\ \geq \ & (1 - 2\sqrt{\tfrac{2}{\pi K_1}} e^{-K_1/8} - 2\sqrt{\tfrac{2}{\pi K_2}} e^{-K_2/8}) \\ & \cdot (e^{-\frac{K_2}{2}(1-view\_prob)^{\frac{K_2}{2}-1}}) \cdot ((1 - p_{fail}) \cdot (1 - p_{ucastl})^{3K_2/2-1}) \\ & \cdot ((p_{fail} + (1 - p_{fail})(1 - p_{mcastl}^{K_2/2}))^N) \cdot (1 - p_{fail}) \end{aligned}$$

□

Figure 4 shows the typical variation of the lower bounds (subscript `lb` stands for "lower bound") of the first four product terms and $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}, \mathsf{E5}]$, for values of K up to 65, with $(view\_prob, p_{mcastl}, p_{ucastl}, p_{fail}, N)$ $= (0.4, 0.01, 0.01, 0.01, 10000)$. The quick convergence of $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}]$ and $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E2}|\mathsf{E1}]$ to unity at small K (here $\simeq 40$) is independent of the value of N. In fact, $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E4}|\mathsf{E1}, \mathsf{E2}, \mathsf{E3}]$ is the only one among the five factors of $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}, \mathsf{E5}]$ that seems to depend on N. However, its value remains close to unity for

$N \cdot p_{mcastl}^{K_2/2} \ll 1$, or $N \ll p_{mcastl}^{-K_2/2}$, which, for $K = 40$, turns out as 10^{39}, a number beyond the size of most practical process groups.

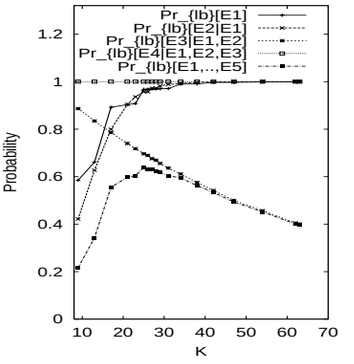


Fig. 4. Pessimistic Analysis of Success Probability of one round of our Leader Election Protocol.

Thus, for all practical values of initial group size N, the minimum probability that an election round of the protocol of Figure 2 satisfies the Uniqueness and Agreement conditions is dependent only on the failure and view probabilities in the group, but is independent of N.

From Figure 4, this minimal protocol round success probability appears to peak at 0.6 for the above parameters. This is because our estimate for $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E3}|\mathsf{E1}, \mathsf{E2}]$ is very pessimistic in assuming a weak global view knowledge, and thus including the possibility that all the initially preferred leaders in the Relay phase might be distinct. In a practical setting however, a fair number of the m (non-faulty) relay members would have the same initial leader choices (eg., if the choice function preferred a candidate leader with lower identity), so the probability $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E3}|\mathsf{E1}, \mathsf{E2}]$ (and hence $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}, \mathsf{E2}, \mathsf{E3}, \mathsf{E4}, \mathsf{E5}]$) would be much higher than the curve shows. The simulation results in Section 5 confirm this for the choice function mentioned above.

Theorem 2 (Detection of incorrect termination in a round): Pr[a re-initiating `mcast` is sent out to the group or all group members fail by the end of the round | election round with parameter K does not succeed] is bounded below by $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}] \cdot (1 - (1 - (1 - p_{fail})(1 - p_{mcastl})^{3K_2/2})^N)$.

Note that, with K fixed so that the term $\mathtt{Pr}_{\mathtt{lb}}[\mathsf{E1}]$ is arbitrarily close to unity, the probability of detection of incorrect election in a round of the presented protocol goes to unity as N tends to infinity. □

Theorem 3 (Round message complexity): With (high) probability $\texttt{Pr}_{\texttt{lb}}[\texttt{E1}]$ $((\texttt{Pr}_{\texttt{lb}}[\texttt{E1}])^2)$, the number of `ucast` (`mcast`) messages in an election round is $O(K^3)$ $(O(K))$ (since K_1, K_2 are both $O(K)$). Also, with (high) probability $(\texttt{Pr}_{\texttt{lb}}[\texttt{E1}])^2$, the number of simultaneous multicasts in the network anytime during the round is $O(K)$. □

Theorem 4 (Round message complexity): Further, the *expected* number of `ucast` (`mcast`) messages in a round of the protocol is $O(K^3)$ $(O(K))$. This is $O(1)$ when K is fixed independent of N. The suggested election protocol round thus achieves the optimal expected message complexity for any global agreement protocol on a group of size N. □

Theorem 5 (Round time complexity): With (high) probability $(\texttt{Pr}_{\texttt{lb}}[\texttt{E1}])^2$, the time complexity of an election round is $O(\mathcal{N}K + N)$ for a group of size N over a network with $\mathcal{N}$ nodes. This is $O(\mathcal{N} + N)$ for K fixed independent of N, which is the optimal time complexity for any global agreement protocol. □

5 Simulation Results

In this section, we analyze, through simulation, the performance of an election protocol strategy from the class described in Section 3.2. The correctness, scalability and fault tolerance of the proposed protocol are more evident here than from the pessimistic analysis of Section 4. The strategy we analyze is specified by 1) an initial (first round) parameter $K_{init} = 7$; 2) for $l \leq 4$, the value of K in round l is twice the value used in round $l-1$; and at $l = 5$, $K = N$; and finally 3) the election protocol aborts after 5 rounds. The protocol is initiated by one `mcast` to the group, which initially has N members.

The unreliability of the underlying network and process group mechanism is characterized by the parameters $p_{ucastl}, p_{mcastl}, p_{fail}, view\_prob$ as defined in Section 4. The hash function is assumed to be a fair one. The choice function used in the simulation is the simple one that prefers candidates with lower identities.

The metrics used to measure the performance of the protocol are the following. *P(Success)* evaluates the final success probability of the protocol, and appears in two forms. "Strong" success probability refers to the (average) probability that a protocol run satisfies the Uniqueness and Agreement conditions. "Weak" success probability is in fact the (average) majority fraction of the non-faulty group members that agree on one leader at the end of the protocol. This is a useful metric for situations where electing more than one leader may be allowed, such as [18]. *# Rounds* refers to the average number of rounds after which the protocol terminates, either successfully, or without detecting an inconsistent election, or because the maximum number of rounds specified by the strategy has been reached. *# Messages* refers to the average number of `ucast` and `mcast` messages generated in the network during the protocol.

Figure 5 shows the results from the simulations. This figure is organized with each column of graphs indicating the variation of a particular performance metric

as a function of each of the system parameters, and each row of graphs showing the effect of varying a system parameter on each of the performance metrics. Each point on these plots is the average of results obtained from 1000 runs of the protocol with the specified parameters. In Figures 5(a-c), $p_{ucastl} = p_{mcastl}$ is varied in the range $[0, 0.5]$ for fixed $N = 2000, p_{fail} = 0.001, view\_prob = 0.5$. The graphs for varying p_{fail} are very similar and not included here. In Figures 5(d-f), N is varied in the range $[1000, 5000]$ for fixed $p_{fail} = 0.001, view\_prob = 0.5, p_{ucastl} = p_{mcastl} = 0.001$. In Figures 5(g-i), $view\_prob$ is varied in the range $[0.2, 0.5]$ for $N = 5000, p_{fail} = 0.001, p_{ucastl} = p_{mcastl} = 0.001$.

Figures 5(a,d,g) show the very high success probability (strong) guaranteed by the above strategy even in the face of high message loss rates (up to $p_{ucastl} = p_{mcastl} = 0.4$, up to and beyond $N = 6000$ and $view\_prob = 0.2$). Notice that even the "weak" success ratio is close to 1 for these ranges, and as expected, is higher than the strong success probability. Figures 5(b,e,h) show the time scalability of the protocol for the same ranges of parameters that produced high success probabilities. Note Figure 5(e), which shows termination within 1 expected round for values of N up to 6000 (!) group members. Figures 5(c,f,i) show the message scalability for the same variation of parameters. Note again the lack of variation in the expected number of messages exchanged (Figure 5(f)) as N is varied up to 6000 members.

Figures 5(a-c) display the level of *fault tolerance* the protocol possesses with respect to message failures. Figures 5(d-f) show how much our protocol *scales* even as the number of group members is increased into the thousands. Finally, Figures 5(g-i) show that our protocol performs well even in the presence of only *partial membership* information at each member.

6 Conclusions

This paper described a novel leader election protocol that is scalable, but provides only a probabilistic guarantee on correct termination. Mathematical analysis and simulation results show that the protocol gives very good probabilities of correct termination, in the classical sense of the specification of leader election, even as the group size is increased into the tens of thousands. The protocol also (probabilistically) guarantees a low and almost constant message complexity independent of this group size. Finally, all these guarantees are offered in the face of process and link failure probabilities in the underlying network, and with only a weak membership view requirement.

The trade-off among the above guarantees is determined by one crucial protocol parameter–the value of K in an election round. From the simulation results, it is clear that choosing K to be a small number (although not very small) suffices to provide acceptable guarantees for the specified parameters. Increasing the value of K would enable the protocol to tolerate higher failure probabilities, but would increase its message complexity. Varying K thus yields a trade-off between increasing the fault tolerance and correctness probability guarantee on one hand and lowering the message complexity on the other.

Acknowledgements We wish to thank the anonymous referees for their suggestions in improving the quality of the paper and its presentation.

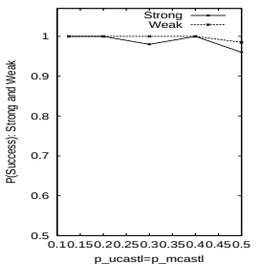


Fig (a). P(Success) vs message loss probability.

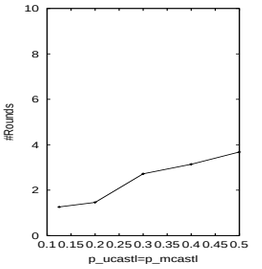


Fig (b). Avg. #Rounds vs message loss probability.

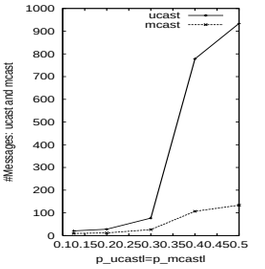


Fig (c). Avg. #Messages vs message loss probability.

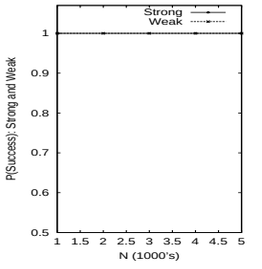


Fig (d). P(Success) vs # Group members.

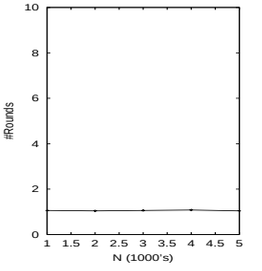


Fig (e). Avg. #Rounds vs # Group members.

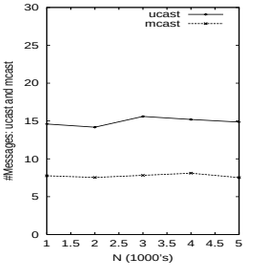


Fig (f). Avg. #Messages vs # Group members.

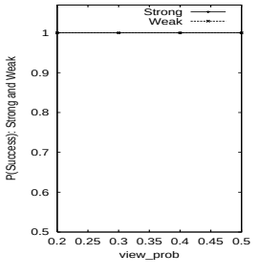


Fig (g). P(Success) vs view probability.

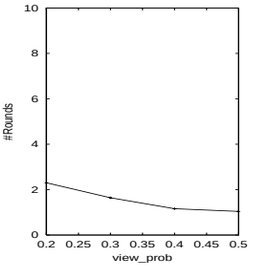


Fig (h). Avg. #Rounds vs view probability.

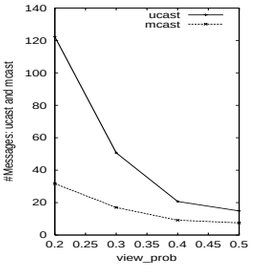


Fig (i). Avg. #Messages vs view probability.

Fig. 5. Performance characteristics of our Leader Election Protocol

References

1. K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky, "Bimodal multicast", *ACM Trans. Computer Systems*, vol. 17, no. 2, May 1999, pp. 41-88.
2. B. Bollobas, A. Thomason, "Random graphs of small order", *Annals of Discrete Mathematics, Random Graphs '83*, vol. 8, 1983, pp. 47-97.
3. J. Brunekreef, J.-P. Katoen, R. Koymans, S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks", *Distributed Computing*, vol. 9, no. 4, Mar 1997, pp. 157-171

4. T.D. Chandra, S. Toueg, "Unreliable failure detectors for asynchronous systems", *Proc. 10th Annual ACM Symp. Principles of Distributed Computing*, 1991, pp. 325-340.
5. B. Chor, C. Dwork, "Randomization in Byzantine agreement", *Advances in Computing Research*, vol. 5, 1989, pp. 443-498.
6. D. Dolev, C. Dwork, L. Stockmeyer, "On the minimal synchronism needed for distributed consensus", *JACM*, vol. 34, no. 1, Jan 1987, pp. 77-97.
7. C. Fetzer, F. Cristian, "A highly available local leader election service", *IEEE Trans. Software Engineering*, vol. 25, no. 5, Sep-Oct 1999, pp. 603-618.
8. M.J. Fischer, N.A. Lynch, M.S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journ. of the ACM*, vol. 32, no. 2, Apr 1985, pp. 374-382.
9. R. Gallager, P. Humblet, P. Spira, "A distributed algorithm for minimum weight spanning trees", *ACM Trans. Programming Languages nd Systems*, vol. 4, no. 1, Jan 1983, pp. 66-77.
10. I. Gupta, R. van Renesse, K.P. Birman, "A probabilistically correct leader election protocol for large groups", Computer Science Technical Report ncstrl.cornell/TR2000-1794, Cornell University, U.S.A., Apr. 2000.
11. A. Itai, "On the computational power needed to elect a leader", *Lecture Notes in Computer Science*, vol. 486, 1991, pp. 29-40.
12. C.-T. King, T.B. Gendreau, L.M. Ni, "Reliable election in broadcast networks", *Journ. Parallel and Distributed Computing*, vol. 7, 1989, pp. 521-540.
13. C. Malloth, A. Schiper, "View synchronous communication in large scale networks", *Proc. 2nd Open Workshop of the ESPRIT project BROADCAST*, Jul 1995.
14. R. Ostrovsky, S. Rajagopalan, U. Vazirani, "Simple and efficient leader election in the full information model", *Proc. 26th Annual ACM Symp. Theory of Computing*, 1994, pp. 234-242.
15. O. Ozkasap, R. van Renesse, K.P. Birman, Z. Xiao, "Efficient buffering in reliable multicast protocols", *Proc. 1st Intnl. Workshop on Networked Group Communication*, Nov. 1999, Lecture Notes in Computer Science, vol. 1736.
16. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill International Edition, 3^{rd} edition, 1991.
17. D. Peleg, "Time optimal leader election in general networks", *Journ. Parallel and Distributed Computing*, vol. 8, no. 1, Jan, 1990, pp. 96-99.
18. R. De Prisco, B. Lampson, N. Lynch, "Revisiting the Paxos algorithm", *Proc.* 11^{th} *Intnl. Workshop on Distributed Algorithms*, 1997, Lecture Notes in Computer Science, vol. 1320, pp. 111-125.
19. M.O. Rabin, "Randomized Byzantine generals", *Proc. 24th Annual Symp. Foundations of Computer Science*, Nov. 1983, pp. 403-409.
20. L.S. Sabel, K. Marzullo, "Election vs. consensus in asynchronous systems", Computer Science Technical Report ncstrl.cornell/TR95-1488, Cornell University, U.S.A., 1995.
21. S. Singh, J.F. Kurose, "Electing good leaders", *Journ. Parallel and Distributed Computing*, vol. 21, no. 2, May 1994, pp. 184-201.
22. G. Taubenfeld, "Leader election in the presence of n-1 initial failures", *Information Processing Letters*, vol. 33, no. 1, Oct 1989, pp. 25-28.
23. S. Toueg, "Randomized Byzantine agreements", *Proc. 3rd Annual ACM Symp. Principles of Distributed Computing*, 1984, pp. 163-178.
24. R. van Renesse, Y. Minsky, M. Hayden, "A gossip-style failure detection service", *Proc. Middleware '98 (IFIP)*, Sept 1998, pp. 55-70.
25. D. Zuckerman, "Randomness-optimal sampling, extractors, and constructive leader election", *Proc. 28th Annual ACM Symp. Theory of Computing*, 1996, pp. 286-295.

Approximation Algorithms for Survivable Optical Networks

(Extended Abstract)

Tamar Eilam[1,2] Shlomo Moran[1] Shmuel Zaks[1]

[1] Department of Computer Science
The Technion, Haifa 32000, Israel
{eilam,moran,zaks}@cs.technion.ac.il

[2] IBM T.J. Watson Research Center
Yorktown Heights, N.Y. 10598

We are motivated by the developments in all-optical networks – a new technology that supports high bandwidth demands. These networks provide a set of lightpaths which can be seen as high-bandwidth pipes on which communication is performed. Since the capacity enabled by this technology substantially exceeds the one provided by conventional networks, its ability to recover from failures within the optical layer is important. In this paper we study the design of a survivable optical layer. We assume that an initial set of lightpaths (designed according to the expected communication pattern) is given, and we are targeted at augmenting this initial set with additional lightpaths such that the result will guarantee survivability. For this purpose, we define and motivate a *ring partition survivability condition* that the solution must satisfy. Generally speaking, this condition states that lightpaths must be arranged in rings. The cost of the solution found is the number of lightpaths in it. This cost function reflects the *switching cost* of the entire network. We present some negative results regarding the tractability and approximability of this problem, and an approximation algorithm for it. We analyze the performance of the algorithm for the general case (arbitrary topology) as well as for some special cases.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 104–118, 2000.

1 Introduction

1.1 Background

Optical networks play a key role in providing high bandwidth and connectivity in today's communication world, and are currently the prefered medium for the transmission of data. While first generation optical networks simply served as a transmission medium, second generation optical networks perform some switching and routing functions in the optical domain. In these networks (also termed, *all-optical*) routing is performed by using *lightpaths*. A lightpath is an end-to-end connection established across the optical network. Every lightpath corresponds to a certain route in the network, and it uses a wavelength in each link in its route. (Two lightpaths which use a same link are assigned different wavelengths.) Routing of messages is performed on top of the set of lightpaths where the route of every message is a sequence of complete lightpaths. At least in the near term the optical layer provides a static (fixed) set of lightpaths which is set up at the time the network is deployed.

Since the capacity enabled by this technology substantially exceeds the one provided by conventional networks, it is important to incorporate the ability to recover from failures into the optical layer. *Survivability* is the ability of the network to recover from failures of hardware components. In this paper we study the design of a survivable optical layer. Our goal is the construction of a low-cost survivable set of lightpaths in a given topology. We assume that an initial set of lightpaths (designed according to the expected communication pattern) is given, and we are targeted at augmenting this initial set with additional lightpaths such that the resulting set will guarantee survivability. For this purpose, we define a *survivability condition* that the solution must satisfy and a *cost function* according to which we evaluate the cost of the solution found.

We focus on the *ring partition survivability condition*. Informally, this condition states that lightpaths are partitioned to rings, and that all lightpaths in a ring traverse disjoint routes in the underlying topology. The motivation for the ring partition survivability condition is two folded. First, it supports a simple and fast protection mechanism. In the case of a failure, the data is simply re-routed around the impaired lightpath, on the alternate path of lightpaths in its ring. The demand that all lightpaths in one ring traverse disjoint routes guarantees that this protection mechanism is always applicable in the case of one failure. Second, a partition of the lightpaths to rings is necessary in order to support a higher layer in the form of SONET/SDH self healing rings which is anticipated to be the most common architecture at least in the near term future ([GLS98]).

Another issue is determining the cost of the design. We assume that a *uniform cost* is charged for every lightpath, namely, the cost of the design is the number of lightpaths in it. This cost measure is justified for two reasons. First, in regional area networks it is reasonable to assume that the same cost will be charged for all the lightpaths ([RS98]). Second, every lightpath is terminated by a pair of line terminals (LTs, in short). The *switching cost* of the entire network is dominated by the number of LTs which is proportional to the number

of lightpaths ([GLS98]).

We assume that the network topology is given in the form of a simple graph. A lightpath is modeled as a pair (ID, P) where ID is a unique identifier and P is a simple path in the graph. A *design* D for a set of lightpaths C is a set of lightpaths which subsumes C (i.e., $C \subseteq D$). A design is termed *ring partition* if it satisfies the ring partition condition. The *cost* of a design is the number of lightpath in it (namely, $cost(D) = |D|$). We end up with the following optimization problem which we term the *minimum cost ring partition design* (MCRPD in short) problem. The input is a graph G and an initial set C of lightpaths in G. The goal is to find a ring partition design D for C with minimum cost.

1.2 Results

We prove that the MCRPD problem is NP-hard for every family of topologies that contains cycles with unbounded length, e.g., rings (see formal definition in the sequel). Moreover, we prove that there is no polynomial time approximation algorithm A that constructs a design D which satisfies $Cost(D) \leq OPT + n^{\alpha}$, for any constant $\alpha < 1$, where n is the number of lightpaths in the initial set, and OPT is the cost of an optimal solution for this instance (unless $P = NP$). For $\alpha = 1$, a trivial approximation algorithm constructs a solution within this bound.

We present a *ring partition algorithm* (RPA, in short) which finds in polynomial time a ring partition design for every given instance of MCRPD (if it exists). We analyze the performance of RPA and show that for the general case (arbitrary topology) RPA guarantees
$Cost(D) \leq \min(OPT + \frac{3}{5} \cdot n, 2n)$, where n and OPT are as defined above. We analyze the performance of RPA also for some interesting special cases in which better results are achieved.

The structure of the paper follows. We first present the model (Section 2), followed by a description of the MCRPD problem (Section 3). We then discuss the results (Section 4), followed by a summary and future research directions (Section 5). Some of the proofs in this extended abstract are only briefly sketched or omitted.

1.3 Related Works

The paper [GLS98] studies ring partition designs for the special case where the physical topology is a ring. In fact, the MCRPD problem is a generalization of this problem for arbitrary topologies. This paper also motivates the focus on the number of lightpaths rather than the total number of wavelengths in the design. Some heuristics to construct ring partition designs in rings are given and some lower and upper bounds on the cost (as a function of the load) are proved. The paper also considers *lightpath splitting* – a lightpath might be partitioned to two or more lightpath. It is shown that better results can be achieved by splitting lightpaths.

Other works in this field refer to different models than what we considered. [GRS97] presents methods for recovering from channel, link and node failures in first generation WDM ring networks with limited wavelength conversion.

Other works refer to second generation optical networks, where traffic is carried on a set of lightpaths. The paper [RS97] assumes that lightpaths are dynamic and focuses on management protocols for setting them up and taking them down.

When the set of lightpaths is static, the survivability is achieved by providing disjoint routes to be used in the case of a failure. [HNS94] and [AA98] studies this problem but the objective is the minimization of the total number of wavelengths and not the number of lightpaths.

The paper [ACB97] offers some heuristics and empirical results for the following problem. Given the physical topology and a set of connections requests (i.e., requests for lightpaths in the form of pairs of nodes), find routes for the requests so as to minimize the number of pairs (l, e) consisting of a routed request (i.e., a lightpath) l and a physical link e, for which there is no alternative path of lightpaths between the endpoints of l in the case that e fails. Note that this survivability condition is less restrictive than the ring partition condition that we consider in this paper.

2 Model and Definitions

For our purposes, lightpaths are modeled as *connections*, where every connection c has a unique identifier $ID(c)$ and is associated with a simple path $\mathcal{R}(c)$ in the network. $\mathcal{R}$ is termed the *routing function*. Note that two different connections might have the same route. We assume that routes of connections are always simple (i.e., they do not contain loops). We say that two connections are *disjoint* if their routes are disjoint, namely, they do not share any edge and any node which is not an end node of both connections. We use the terms connections and lightpaths interchangeably.

A *virtual path* P is a sequence $\langle v_1, c_1, v_2, c_2, \cdots, c_k, v_{k+1} \rangle$, where c_i is a connection with endpoints v_i and v_{i+1} (for $i = 1, \cdots, k$). P is termed a *virtual cycle* if $v_1 = v_{k+1}$. We denote by $S(P)$ the set $\{c_1, c_2, \cdots, c_k\}$ of connections in P. The routing function $\mathcal{R}$ is naturally generalized to apply to virtual paths (and cycles) by concatenating the corresponding paths of connections. A virtual path (or cycle) P is termed *plain* if $\mathcal{R}(P)$ is a simple path (or cycle) in the network.

A design D for a set of connections C in a network G is a set of connections which subsumes C (i.e., $C \subseteq D$). A *ring partition design* D for a set of connections C satisfies $D = \cup_{t \in T} S(P_t)$, where every P_t, $t \in T$, is a plain virtual cycle, and $S(P_{t_1}) \cap S(P_{t_2}) = \emptyset$, for every $t_1, t_2 \in T$. The partition $\{P_t\}_{t \in T}$ is termed *the ring partition of the design* D. For a design D, $cost(D) = |D|$, i.e., the number of lightpaths in the design.

The *minimum cost ring partition design* (MCRPD, in short) problem is formally defined as follows. The input is a graph G and a set of connections C in

G. The goal is to find a ring partition design D for C that minimizes $cost(D)$. The corresponding decision problem is to decide for a set of connections C in G and a positive integer s whether there is a ring partition design D for C such that $cost(D) \leq s$.

MCRPD$_{\mathcal{G}}$ denotes the version of the problem in which the input is restricted to a family $\mathcal{G}$ of networks (e.g., the family $\mathcal{R}$ of rings).

Figure 1 is an example of the MCRPD problem, where **(*a*)** shows an instance with an initial set of size 4, and **(*b*)** shows a solution which consists of 2 rings and 3 new connections. The cost of the solution is thus 7.

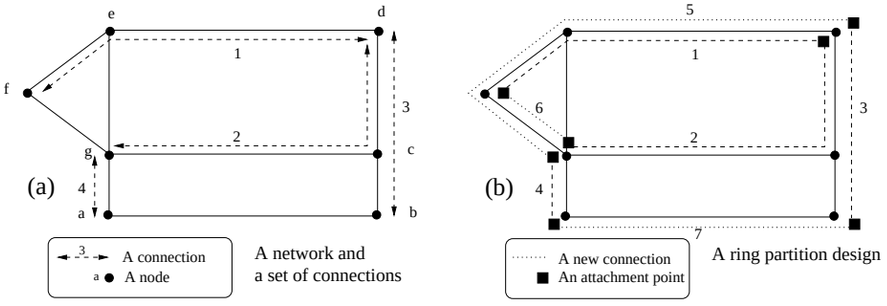


Fig. 1. The MCRPD problem.

3 The MCRPD Problem

In this section we start our study of the MCRPD problem by providing some negative results regarding the tractability and approximability of the problem.

We say that a family of topologies $\mathcal{G} = G_1, G_2, \cdots$ has the *unbounded cycle* (UBC) property if there exists a constant k, such that for every n, there exists a graph $G_{i_n} \in \mathcal{G}$, with size $O(n^k)$, that contains a cycle of length n. Examples for families of topologies having the UBC property are the family $\mathcal{R}$ of ring topologies, and the family of complete graphs.

Theorem 1. *The MCRPD$_{\mathcal{G}}$ problem is NP-hard for every family of topologies $\mathcal{G}$ having the UBC property.*

Proof. See [EMZ00].

We continue by studying approximation algorithms for the MCRPD problem. A trivial approximation algorithm is achieved by adding for every connection c, a new disjoint connection between c's endpoints. Note that if there is no such route then there is no ring partition design for this instance. The resulting ring partition design will include virtual cycles, each with two connections, one of which belongs to the initial set C. For an algorithm A, we denote by $A(I)$ the

value of a solution found by A for an instance I, and by $OPT(I)$ the value of an optimal solution. Clearly, $TRIV(I) = 2n \leq OPT(I) + n$, for every instance $I = (G, C)$ of MCRPD, where $|C| = n$. A question which arises naturally is whether there exists an approximation algorithm A for the MCRPD problem that guarantees, $A(I) \leq OPT(I) + n^{\alpha}$, for some constant $\alpha < 1$. We give a negative answer for this questions (for every constant $\alpha < 1$).

Theorem 2. *Let $\mathcal{G}$ be any family of topologies having the UBC property. Then for any constant $\alpha < 1$, $MCRPD_{\mathcal{G}}$ has no polynomial-time approximation algorithm A that guarantees $A(I) \leq OPT(I) + n^{\alpha}$ (unless $P = NP$).*

Proof. See [EMZ00].

The next question is whether there is an approximation algorithm A for MCRPD which guarantees $A(I) \leq OPT(I) + k \cdot n$, where $k < 1$ is a constant (clearly, the trivial algorithm $TRIV$ satisfies this bound for $k = 1$). In the sequel we answer this question positively for $k = \frac{3}{5}$.

4 A Ring Partition Approximation Algorithm

In this section we provide an approximation algorithm, termed *ring partition algorithm* (RPA, in short), for the MCRPD problem. We analyze RPA and show that it guarantees $RPA(I) \leq \min(OPT(I) + \frac{3}{5} \cdot n, 2n)$ for every instance I (where n is the number of connections in the initial set). We also study some special cases in which better results are achieved.

Unless stated otherwise we assume an arbitrary network topology $G = (V, E)$, where $V = \{v_1, \cdots, v_m\}$, and an initial set of connections C in G, where $|C| = n$. We assume that the route $\mathcal{R}(c)$ of every connection c in C is a sub-path in some simple cycle in G (observe that this assumption can be verified in polynomial time, and without it there is no ring partition design D for C).

4.1 Preliminary Constructions

We define some preliminary constructions that are used later for the definition of RPA. Recall that a virtual path P is a sequence $\langle v_1, c_1, v_2, c_2, \cdots, c_k, v_{k+1} \rangle$, where c_i is a connection with endpoints v_i and v_{i+1} (for $i = 1, \cdots, k$). P is termed a *virtual cycle* if $v_1 = v_{k+1}$. The pair of connections c_i and c_{i+1} are termed *attached at node* v_{i+1} in P (or simply, *attached in* P). If P is a virtual cycle then the pair c_1 and c_k are also considered *attached (at node* v_{k+1}*)* in P.

Let C be a set of connections in G, and let v be a node in G. We denote by $C(v) \subseteq C$ the set of connections for which v is an endpoint. Let Q be the symmetric binary relation over the set C of connections that is defined as follows. $(c_1, c_2) \in Q$ iff c_1 and c_2 are disjoint and there exists a simple cycle in G which contains both routes $\mathcal{R}(c_1)$ and $\mathcal{R}(c_2)$. Then Q defines an *end-node graph* $NG_v = (NV_v, NE_v)$ for every node v, where the set of nodes NV_v is $C(v)$, and NE_v is the set of edges, as follows. For every pair of connections $c_i, c_j \in C(v)$, $\{c_i, c_j\} \in NE_v$

iff $(c_i, c_j) \in Q$. A *matching* for a graph $G = (V, E)$ is a set $E' \subseteq E$ such that no two edges in E' share a common endpoint. A *maximum matching* is a matching of maximum size. We denote by $match(G)$ the size of a maximum matching for G. A matching in an end-node graph NG_v, for a node v describes a set of attachments of pairs of connections (which satisfy Q) in v.

Consider a graph $G = (V, E)$, where $V = \{v_1, v_2, \cdots, v_m\}$, and a set of connections C in G. A *matching-set* for G and C is a set of matchings $\mathcal{E} = \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$, where $NE'_{v_i} \subseteq NE_{v_i}$ is a matching in the end-node graph NG_{v_i} (see Figure 2 as an example).

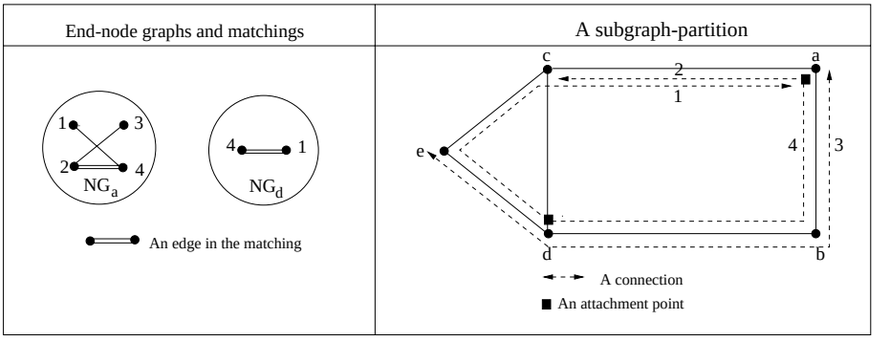


Fig. 2. A graph, a set of connections, a matching-set (where only matchings in non-trivial end-node graphs are shown), and the equivalent subgraph-partition.

A *subgraph-partition* $\mathcal{G} = \mathcal{G}_p \cup \mathcal{G}_c$, for a set of connections C, is a partition of the connections in C into virtual paths and cycles (which are also termed *subgraphs*) as follows. Recall that $S(g)$ is the set of connections that are included in a virtual path (or cycle) g. $\mathcal{G}_p$ is a set of virtual paths, $\mathcal{G}_c$ is a set of virtual cycles, $C = \cup_{g \in \mathcal{G}} S(g)$, and $S(g_1) \cap S(g_2) = \emptyset$ for every $g_1, g_2 \in \mathcal{G}$. Note that the ring partition $\{P_t\}_{t \in T}$ of a ring partition design $D = \cup_{t \in T} S(P_t)$ is actually a subgraph-partition for D (where, $\mathcal{G} = \mathcal{G}_c$, $\mathcal{G}_p = \emptyset$). In general the virtual paths and cycles in a subgraph-partition might not be plain.

Note that there is a one-to-one correspondence between matching-sets and subgraph-partitions, as follows. Consider a matching-set $\mathcal{E} = \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$ and a subgraph-partition $\mathcal{G} = \mathcal{G}_p \cup \mathcal{G}_c$ for a set of connections C in G. $\mathcal{E}$ and $\mathcal{G}$ are termed *equivalent* if the following condition is satisfied. For every pair of connections $c_1, c_2 \in C$, there exists a subgraph g, $g \in \mathcal{G}$, such that c_1 and c_2 are attached at node v_i in g, iff $\{c_1, c_2\} \in NE'_{v_i}$.

For a matching-set $\mathcal{E}$ we denote by $\mathcal{G}_\mathcal{E}$ the (unique) equivalent subgraph-partition. Similarly, $\mathcal{E}_\mathcal{G}$ is the (unique) equivalent matching-set for a given subgraph-partition $\mathcal{G}$. Clearly, for a matching-set $\mathcal{E}$, $\mathcal{E}_{\mathcal{G}_\mathcal{E}} = \mathcal{E}$. As an example see Figure 2.

4.2 Ring Partition Algorithm (RPA)

We present a ring partition algorithm, called RPA, which finds a ring partition design for a set of connections C in G in four main stages. First, the end-node graph NG_{v_i} is constructed and a maximum matching in it is found for every node v_i, $i = 1, \cdots, m$. This defines a maximum matching-set $\mathcal{E}$. Then, the equivalent subgraph-partition $\mathcal{G} = \mathcal{G}_{\mathcal{E}}$ is constructed. Next, we partition every non-plain virtual path or virtual cycle in $\mathcal{G}$ to plain virtual paths. In addition, we make sure that for every virtual path $P \in \mathcal{G}$, there is a simple cycle in G in which $\mathcal{R}(P)$ is a sub-path. Last, the subgraph-partition is completed to a ring partition, by adding for every virtual path $P \in \mathcal{G}$, a connection which completes it to a plain virtual cycle. Following is the description of RPA followed by an informal description of the operations taken by its main functions.

$RPA(G, C)$
$(\mathcal{G}_p, \mathcal{G}_c) := ConstructPartition(G, C)$
$(\mathcal{G}_p, \mathcal{G}_c) := AdjustPartition(\mathcal{G}_p, \mathcal{G}_c, G)$
$D := C \cup CompletePartition(\mathcal{G}_p, \mathcal{G}_c, G)$
return D

$ConstructPartition(G, C)$
for every $i \in 1, \cdots, m$
    construct $NG_{v_i} = (NV_{v_i}, NE_{v_i})$
    find maximum matching $NE'_{v_i} \subseteq NE_{v_i}$
$\mathcal{E} := \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$
construct the equivalent subgraph-partition $\mathcal{G}_{\mathcal{E}} = (\mathcal{G}_p, \mathcal{G}_c)$
return $(\mathcal{G}_p, \mathcal{G}_c)$

$AdjustPartition(\mathcal{G}_p, \mathcal{G}_c, G)$
for every $P \in \mathcal{G}_p \cup \mathcal{G}_c$
    $\mathcal{G}_c := \mathcal{G}_c \setminus \{P\}$ /* in case P is a cycle */
    $C_p := Partition(P)$
    $\mathcal{G}_p := (\mathcal{G}_p \setminus \{P\}) \cup C_P$
for every $P \in \mathcal{G}_P$
    if $(cycle(P))$ then
        $\mathcal{G}_P := \mathcal{G}_P \setminus \{P\}$
        $\mathcal{G}_C := \mathcal{G}_C \cup \{P\}$
return $(\mathcal{G}_p, \mathcal{G}_c)$

$CompletePartition(\mathcal{G}_p, \mathcal{G}_c, G)$
$D' := \emptyset$
for every $P \in \mathcal{G}_P$
    $P^c := findDisjoint(P)$
    $D' := D' \cup \{P^c\}$
return D'

$Partition(P)$
Assume that $P := \langle v_1, c_1, v_2, c_2, \cdots, v_l, c_l, v_{l+1} \rangle$
$C_P := \emptyset$; $first := 1$

32: for $i := 1$ to l
33: $\quad P' := \langle v_{first}, c_{first}, \cdots, c_i, v_{i+1} \rangle$
34: $\quad$ if ($\neg(plain(P') \wedge cycleExists(P'))$) then
35: $\quad\quad C_P := C_P \cup \{\langle v_{first}, c_{first}, \cdots, c_{i-1}, v_i \rangle\}$
36: $\quad\quad first := i$
37: **return** $C_P \cup \{\langle v_{first}, c_{first}, \cdots, c_l, v_{l+1} \rangle\}$

The function *ConstructPartition* first constructs the end-node graphs. The algorithm to construct the end-node graphs is straightforward and is not elaborated. It consists of determining, for every pair of connections with a common endpoint, whether they are disjoint, and whether the path that is formed by concatenating them can be completed to a simple cycle in G. This could be done using standard BFS techniques (see, e.g., [Eve79]). *ConstructPartition* then finds maximum-matchings in the end-node graphs. Efficient algorithms for finding maximum matchings in graphs can be found in, e.g., [MV80] (for a survey see [vL90], pages 580–588)). Last, the construction of the equivalent subgraph-partition is straightforward.

The function *AdjustPartition* partitions every virtual path and virtual cycle in the subgraph-partition using the function *Partition*. After the partition, every virtual path is plain and can be completed to a simple cycle in G. Every virtual path is then checked and if it is actually a cycle (i.e., its endpoints are equal) then it is inserted into $\mathcal{G}_c$.

The task of *Partition* is to partition a virtual path (or cycle) to a set $\{P_1, \cdots, P_l\}$ of plain virtual paths, s.t. for every P_i, $\mathcal{R}(P_i)$ is a sub-path in some simple cycle in G. The function $cycleExists(P)$ returns *true* if there is a disjoint path in G between P's endpoints. The function $cycle(P)$ returns *true* if the endpoints of a given virtual path are equal.

Last, the function *CompletePartition* completes every virtual path in $\mathcal{G}_p$ to a virtual cycle by adding a new disjoint connection P^c between P's endpoints.

4.3 Correctness and Analysis

We first present four observations that are used for the proof of the main theorem (Theorem 3). Observation 1 shows a connection between the sizes of matching-sets and the equivalent subgraph-partitions.

Observation 1 *Let $\mathcal{E} = \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$ be a matching-set for a set of connections C in $G = (V, E)$, where $|C| = n$, and $V = \{v_1, \cdots, v_m\}$. Let $\mathcal{G}_\mathcal{E} = \mathcal{G}_p \cup \mathcal{G}_c$ be the equivalent subgraph-partition. Then $|\mathcal{G}_p| = n - \sum_{i=1}^{m} |NE'_{v_i}|$.*

Proof. Let an *attachment point* in $\mathcal{G}_\mathcal{E}$ be an ordered pair $(\{c_1, c_2\}, v)$, where the connections c_1 and c_2 are attached at node v in some subgraph $g \in \mathcal{G}_\mathcal{E}$. Clearly the number of unique attachment points in a virtual path $P_x \in \mathcal{G}_p$ is one less than the number of connections in P_x, i.e., $|S(P_x)| - 1$. The number of unique attachment points is equal to $|S(P_x)|$ if $P_x \in \mathcal{G}_c$ is a virtual cycle. It follows that the number of unique attachment points is equal to $(\sum_{g \in \mathcal{G}_\mathcal{E}} |S(g)|) - |\mathcal{G}_p| = n - |\mathcal{G}_p|$. Now by the definitions there is a one-to-one correspondence between attachment points and edges in the matchings. It follows that the number of attachment points is equal to the number of edges in the matching set, i.e, $n - |\mathcal{G}_p| = \sum_{i=1}^{m} |NE'_{v_i}|$.

Let $\mathcal{G}(D)$ be a subgraph-partition for a set of connections D. The *projection* $\mathcal{G}(D)|_C$ of $\mathcal{G}(D)$ on a set of connections $C \subseteq D$ is a subgraph-partition for C which is obtained from $\mathcal{G}(D)$ by deleting all the connections that are not in C (i.e., all the connections in $D \setminus C$). Note that a virtual path (or cycle) in $\mathcal{G}(D)$ might be cut by this process into few virtual paths. Similarly, let $\mathcal{E}(D)$ be a matching-set for D. Then the *projection* $\mathcal{E}(D)|_C$ of $\mathcal{E}(D)$ on a set of connections $C \subseteq D$, is a matching-set for C which is obtained from $\mathcal{E}(D)$ by deleting from the end-node graphs (and the matchings) nodes which correspond to connections in $D \setminus C$ and the edges that meet them. Clearly, if $\mathcal{G}(D)$ and $\mathcal{E}(D)$ are equivalent then so are $\mathcal{G}(D)|_C$ and $\mathcal{E}(D)|_C$.

Consider a ring partition design $D = \cup_{t \in T} S(P_t)$ for a set of connections C. We denote by $\mathcal{G}(D)$ the ring partition $\{P_t\}_{t \in T}$ of D, and by $\mathcal{E}(D)$ the equivalent matching-set for D (i.e., $\mathcal{E}(D) = \mathcal{E}_{\mathcal{G}(D)}$). The subgraph-partition $\mathcal{G}(D)|_C$ and the matching-set $\mathcal{E}(D)|_C$ for the initial set of connections C are termed *the induced subgraph-partition* and *the induced matching-set*, respectively (note that they are equivalent). Observation 2 associates the cost of ring partition designs, with the sizes of the induced matching-sets and subgraph-partitions.

Observation 2 *Let $D = \cup_{t \in T} S(P_t)$ be a ring partition design for a set of connections C in a physical topology $G = (V, E)$, where $|C| = n$, and $|V| = m$. Let $\mathcal{E}(D)|_C = \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$ and $\mathcal{G}(D)|_C = \mathcal{G}_p \cup \mathcal{G}_c$ be the induced matching-set and subgraph-partition for C. Then $cost(D) \geq n + |\mathcal{G}_p| = 2n - \sum_{i=1}^{m} |NE'_{v_i}|$.*

Proof. By the definitions, $Cost(D) = \sum_{t \in T} S(P_t)$. Let $new(P_t)$ be the number of new connections in the virtual cycle P_t, i.e., $new(P_t) = S(P_t) \cap (D \setminus C)$. Clearly, $Cost(D) = n + \sum_{t \in T} new(P_t)$. Consider now the induced subgraph partition $\mathcal{G}(D)|_C = \mathcal{G}_p \cup \mathcal{G}_c$. Recall that it is obtained from D by deleting all the new connections. In this process a virtual cycle in the ring-partition might be cut into few virtual paths. Clearly the number of such virtual paths for each virtual cycle, is at most the number of new connections in it. It follows that $|\mathcal{G}_p| \leq \sum_{t \in T} new(P_t)$, thus $Cost(D) \geq n + |\mathcal{G}_P|$. by Observation 1, $n + |\mathcal{G}_p| = 2n - \sum_{i=1}^{m} |NE'_{v_i}|$. Note that strict inequality occurs when two new connections are attached in one of the virtual cycles.

A *maximum-matching-set*, is a matching set $\mathcal{E} = \{NE'_{v_1}, \cdots, NE'_{v_m}\}$ for a set of connections C, s.t. the matching NE'_{v_i} is a maximum matching for the end-node graph NG_{v_i}, for every $i = 1, \cdots, m$. Recall that $match(G)$ is the size of a maximum matching for G. Observation 3 is a lower bound on the value of an optimal solution.

Observation 3 *Every ring partition design D for C satisfies $cost(D) \geq 2n - \sum_{i=1}^{m} match(NG_{v_i})$ (where, n and m are defined as above).*

Proof. Let $D = \cup_{t \in T} S(P_t)$ be a ring-partition design for C. Note that every two connections that are attached in a virtual cycle P_t, $t \in T$, in the design satisfy the relation Q, i.e., they are disjoint and there is a simple cycle that contains both routes. Clearly, the same holds also for the induced subgraph-partition $\mathcal{G}(D)|_C$ and matching-set (since we only delete connections). Consider the equivalent matching set $\mathcal{E}(D)|_C = \{NE'_{v_1}, NE'_{v_2}, \cdots, NE'_{v_m}\}$. In follows that NE'_{v_i} is actually a matching in the end-node graph NG_{v_i}, for $i = 1, \cdots, m$, and thus $|NE'_{v_i}| \geq match(NE_{v_i})$. It follows, by Observation 2, that $Cost(D) \geq 2n - \sum_{i=1}^{m} |NE'_{v_i}| \geq 2n - \sum_{i=1}^{m} match(NG_{v_i})$.

Consider a ring partition design $D = \cup_{t \in T} S(P_t)$ for a set of connections C in G. Let $new(P_t)$ be the number of new connections in $S(P_t)$ (i.e., connections in $S(P_t) \cap (D \setminus C)$).

A *canonical* ring partition design satisfies that $new(P_t) \leq 1$ for every $t \in T$. Note that it is always possible to construct from a given ring partition design D, a canonical ring partition design D' such that $Cost(D') \leq Cost(D)$ as follows. Let $\mathcal{G}(D)|_C = \mathcal{G}_p \cup \mathcal{G}_c$ be the induced subgraph partition of D. To construct a canonical ring partition design D' with at most the same cost we complete every virtual path in $\mathcal{G}_p$ to a plain virtual cycle by adding one new connection. (This is always doable since every virtual path in $\mathcal{G}_p$ is plain and is included in some simple cycle in G). By the discussion above, $Cost(D') = n + |\mathcal{G}_p| \leq Cost(D)$. Observation 4 follows.

Observation 4 *If there is a ring partition design for a set of connections C in G then there is a canonical ring partition design with minimum cost.*

It can be proved that Observation 2 holds for canonical ring partition designs D' with equality i.e., $cost(D') = n + |\mathcal{G}_p|$. It is therefore sometimes convenient to consider for simplicity only canonical ring partition designs.

We are now ready to prove the main theorem.

Theorem 3. *$RPA(I) \leq \min(OPT(I) + \frac{3}{5} \cdot n, 2n)$, for every $I = (G, C)$, where $|C| = n$.*

Sketch of Proof: For the analysis we denote by $\mathcal{G}_p^1$ and $\mathcal{G}_c^1$ the sets $\mathcal{G}_p$ and $\mathcal{G}_c$ right after the execution of $ConstructPartition$, and by $\mathcal{G}_p^2$ and $\mathcal{G}_c^2$ the corresponding sets right after the execution of $AdjustPartition$.

We now examine the partition procedure $Partition$. Recall that the end-node graphs are constructed w.r.t. the relation Q which is *true* for a pair of connections c_1 and c_2 iff their routes $\mathcal{R}(c_1)$ and $\mathcal{R}(c_2)$ are disjoint and there is a simple cycle which contains both routes (as sub-paths). Consider a virtual path $P = \langle v_1, c_1, v_2, c_2, \cdots, v_{l-1}, c_l, v_l \rangle \in \mathcal{G}_p^1$. Since P is a virtual path in the equivalent subgraph-partition $\mathcal{G}_\mathcal{E}$, it holds that $(c_i, c_{i+1}) \in Q$, for every $i = 1, \cdots, l-1$. Let C_P be the set of virtual paths which is the output of $Partition(P)$. By the above discussion, and by the definition of $Partition$, at most one virtual path in C_P contains less than two connections. Such a virtual path can be only the last one, which contains the connection c_l. Let $n_P = |S(P)|$ (i.e., the number of connections in the virtual path P). Let $m_P = |C_P|$ (i.e., the number of plain virtual paths that are the result of applying the partition procedure on P). It follows that $m_P \leq \lfloor \frac{n_P+1}{2} \rfloor$.

Now consider a non-plain virtual cycle $P \in \mathcal{G}_c^1$. Then, by the same considerations, $m_P \leq \lfloor \frac{n_P+1}{2} \rfloor$, where n_P and m_P are defined similarly.

Let $\mathcal{G}_c' \subseteq \mathcal{G}_c^1$ and $\mathcal{G}_c'' \subseteq \mathcal{G}_c^1$ be the sets of non-plain virtual cycles with, respectively, odd and even number of connections, after $ConstructPartition$. Note that $CompletePartition$ adds one new connection for every virtual path $P \in \mathcal{G}_p^2$. We get,

$$\begin{aligned} RPA(I) &= |\mathcal{G}_p^2| + n \\ &\leq \sum_{P \in \mathcal{G}_c'} (\tfrac{n_P}{2} + \tfrac{1}{2}) + \sum_{P \in \mathcal{G}_c''} (\tfrac{n_P}{2}) + \sum_{P \in \mathcal{G}_p^1} (\tfrac{n_P}{2} + \tfrac{1}{2}) + n \\ &\leq \tfrac{3n}{2} + \tfrac{1}{2}|\mathcal{G}_c'| + \tfrac{1}{2}|\mathcal{G}_p^1| \end{aligned}$$

Observe that a non-plain virtual cycle in $\mathcal{G}_P^1$ contains at least 4 connections, since otherwise clearly there are two consecutive connections that are not disjoint in the cycle, which is not possible by the definition of the algorithm. It follows that $|\mathcal{G}_c'| \leq \frac{n}{5}$. We get, $RPA(I) \leq n + \frac{3}{5} \cdot n + \frac{1}{2}|\mathcal{G}_p^1|$. Now, by Observation 3, we can show that $OPT(I) \geq n + |\mathcal{G}_p^1|$ (since in the first step RPA finds maximum matchings in the end-node graphs). Thus, $RPA(I) \leq OPT(I) + \frac{3}{5} \cdot n$.

Observe that RPA constructs a canonical solution, i.e., there is at most one new connection in every ring. Clearly, there is at least one connection from the initial set in every ring. It follows, $RPA(I) \leq 2n$. ■

Note that since $OPT(I) \geq n$, this is actually better than a $\frac{8}{5}$-approximation.

Time complexity. The time complexity depends on the exact format of the input for the algorithm and the data structures which are used in order to represent the physical topology, the set of connections and the auxiliary combinatorial constructions (i.e., the end-node graphs, and the subgraph partition). It is clear however that this time is polynomial in the size of C and G. It is well-known that it takes $O(\sqrt{|V|} \cdot |E|)$ time to find a maximum matching in a graph $G = (V, E)$ ([MV80]) and that it takes $O(|E|)$ time to find whether two paths are disjoint, or whether there exists a disjoint path between a given path's endpoints. For special topologies these tasks can be significantly simpler. For instance, clearly in the ring physical topology case, every plain virtual path can be completed to a plain virtual cycle, thus the relation Q can be simplified to $Q(c_1, c_2) = disjoint(c_1, c_2)$. The end-node graphs are bipartite, and finding maximum matchings in bipartite graphs is considerably easier ([vL90]). Also, to find a disjoint path between the endpoints of a given simple path is trivial. In any case, for the applications of RPA for the design of optical networks time-efficiency is not crucial since the algorithm is applied only in the design stage of the network and it is reasonable to invest some preprocessing time once in order to achieve better network designs.

4.4 Special Cases

4.5 Optimal Cases

Since the MCRPD problem is NP-hard (Theorem 1) it is natural to try and find restricted families of topologies for which it can be solved in polynomial time. Unfortunately, we actually proved in Theorem 1 that the MCRPD problem is NP-hard for every family of topologies that contains cycles with unbounded length (e.g., rings). Since trees do not support ring partition designs, this implies that the problem is NP-hard for every family of topologies which is of interest in this setting. This observation motivates the question of finding polynomially solvable classes of instances of the problem when taking into account not only the topology of the network but also the initial set of connections.

The *induced graph* $IG_C = (IV_C, IE_C)$ for a set of connections C in G is the subgraph of G which includes all the edges and nodes of G that are used by at least one connection in C.

A natural question is whether applying restrictions on the induced graph suffices to guarantee efficient optimal solution to the problem. We answer this question negatively by showing that the problem remains NP-hard even for the most simple case where the induced graph is a chain.

Theorem 4. *The MCRPD problem is NP-hard even if the induced graph for the set of connections C in G is a chain (or a set of chains).*

Next we show that if, in addition to an induced graph with no cycles, the network topology satisfies a certain condition (w.r.t. the initial set of connections), then RPA finds a minimum cost ring partition design.

Theorem 5. *$RPA(I) = OPT(I)$ for every instance $I = (G, C)$ which satisfies the following two properties.*

No Cycles. *The induced graph $IG_C = (IV_C, IE_C)$ is a forest.*

Completion. *For every plain virtual path P over C, there is a simple cycle in G that contains the route of P, $\mathcal{R}(P)$, as a sub-path.*

We discuss below some cases in which the conditions in Theorem 5 are satisfied. A perfectly-connected graph (PC, in short) satisfies that every simple path in it is included in a simple cycle. Clearly, if a graph is perfectly connected than the completion property is satisfied for every initial set of connections. This property also guarantees that there is a ring partition design D for *every* initial set of connections C. A natural question is to characterize perfectly connected graphs. We give a full characterization of perfectly connected graphs by proving that a graph is PC iff it is randomly Hamiltonian. Randomly Hamiltonian graphs are defined and characterized in [CK68].

Theorem 6. *A graph G is perfectly connected iff it is one of the following: a ring, a complete graph, or a complete bipartite graph with equal number of nodes in both sets.*

We note that RPA does not have to be modified in order to give an optimal result for instances which satisfy the conditions in Theorem 5. However, we can benefit from recognizing in advance such instances since in these cases the procedure $AdjustPartition$ can be skipped. The Recognition can be done easily for specific topologies (e.g., rings), and in polynomial time in the general case.

4.6 Bounded Length Connections in Rings

We analyze the performance of RPA in the case of a ring physical topology, when there is a bound on the length of connections in the initial set.

Theorem 7. *$RPA(I) \leq \min(OPT(I) + \frac{3k}{2m} \cdot n, 2n)$, for every instance $I = (R_m, C)$ of $MCRPD_{\mathcal{R}}$, if for every connection $c \in C$, $length(\mathcal{R}(c)) \leq k$, for any constant k, $1 \leq k \leq m-1$.*

Note that RPA does not guarantee that the same bound on the length holds also for connections in the ring partition design which is constructed. Indeed, the case where the length of connections in the solution must be bounded is inherently different, and the main results in this paper do not hold for it.

4.7 Approximations Based on the Load

Let the *load* l_e of an edge $e \in E$ be the number of connections in C which use e, and $l_I = \max_{e \in E} l_e$. Recall the definition of an induced graph $IG_C = (IV_C, IE_C)$ for a set of connections C in G (Section 4.5). We add to this definition a weight function $w : IE_C \rightarrow N$ that assigns a weight for every edge that is equal to its load. Although in the worst case the load of an instance is equal to the number of connections $|C|$, usually it is substantially smaller. Therefore, it is interesting to bound the cost of a design as a function of the load.

For this purpose, we assume that the route of every virtual path is a sub-path is some simple cycle in G (i.e., the completion property). Let $W = \sum_{e \in E} l_e$. Now consider the weighted induced graph $IG_C = (IV_C, IE_C, W_C)$ for C. Let T_{max} be a maximum-weight spanning tree in IG_C, $W_{T_{max}} = \sum_{e \in T_{max}} l_e$, and $W_{G-T_{max}} = W - W_{T_{max}}$. Following is a description of a modified version of RPA, termed RPA$_l$. We temporarily remove all connections that use edges that are not in T_{max}. Next, we find a ring

partition design for the remaining set of connections (using RPA). Last, we insert back the removed connections and complete each one of them to a virtual cycle by adding a new connection. We prove that the cost of the resulting ring partition design is larger by at most $2W_{G-T_{max}}$ than the optimal one. (Note that an improved heuristics might be to repeat the same process with the remaining set of connections.)

Theorem 8. *$RPA_l(I) \leq OPT(I) + 2W_{G-T_{max}}$, for every instance $I = (G, C)$ which satisfies the completion property.*

For the case of a ring physical topology, it holds $RPA_l(I) \leq OPT(I) + \min_{e \in E} l_e$. A slightly better bound is given for this case in [GLS98].

Note that there might be a set of connections C'_{min} with size smaller than $W_{G-T_{max}}$ such that the induced graph for the remaining set $C \setminus C'_{min}$ is a forest. However, we prove in Proposition 9 that finding a minimum set of connections whose removal leaves us with an induced graph with no cycles is NP-hard.

Proposition 9. *Finding a minimum set of connections $C' \subseteq C$ in a graph G such that the induced graph for the remaining set $C \setminus C'$ does not contain cycles is NP-hard.*

5 Summary and Future Research

In this paper we studied the MCRPD problem for which the input is an initial set of lightpaths in a network and the goal is to augment this set by adding lightpaths such that the result is a ring partition design with minimum cost. We have shown an approximation algorithm for this problem that guarantees $Cost(D) \leq \min(OPT + k \cdot n, 2n)$, where $k = \frac{3}{5}$, n is the number of lightpaths in the initial set, and OPT is the cost of an optimal solution. Moreover, we have shown that, unless $P = NP$, there is no approximation algorithm A for this problem that guarantees $Cost(D) \leq OPT + n^{\alpha}$, for every constant $\alpha < 1$. The main open question here is whether the constant k can be improved.

Ring partition designs are necessary for the near term future of optical networks since they support a SONET higher layer network which is configured in the form of rings. However it is claimed that the core network architecture will have to change and that SONET will give way to a smart optical layer. Incorporating new technologies it might be possible to re-route lightpaths dynamically. In these cases other less restrictive survivability conditions might be considered. While less restrictive survivability conditions might be less expensive to implement, the price to pay is of a more complex protection mechanism that is executed for every failure. The challenge here is two folded. First, to study the gain in the cost of the network when less restrictive survivability conditions are considered. Second, to study the algorithmic and technological issues of implementing protection mechanisms in the optical domain based on these conditions.

Acknowledgment We would like to thank Ornan (Ori) Gerstel for introducing us to this problem and for very helpful discussions.

References

[AA98] M. Alanyali and E. Ayanoglu. Provisioning algorithms for WDM optical networks. In *Proc. IEEE INFOCOM '98*, pages 910–918, 1998.

[ACB97] J. Armitage, O. Crochat, and J. Y. Le Boudec. Design of survivable WDM photonic network. In *Proc. IEEE INFOCOM '97*, pages 244–252, 1997.

[CK68] G. Chartrand and H. V. Kronk. Randomly traceable graphs. *SIAM J. Appl. Math.*, 16:696–700, 1968.

[EMZ00] T. Eilam, S. Moran, and S. Zaks. Approximation algorithms for survivable optical networks. Technical Report 2000-05, Department of Computer Science, Technion, Haifa, Israel, April 2000.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Woodland Hills, CA, 1979.

[GLS98] O. Gerstel, P. Lin, and G. Sasaki. Wavelength assignment in WDM ring to minimize cost of embedded SONET rings. In *Proc. IEEE INFOCOM '98*, pages 94–101, 1998.

[GRS97] O. Gerstel, R. Ramaswami, and G.H. Sasaki. Fault tolerant multiwavelength optical rings with limited wavelength conversion. In *Proc. IEEE INFOCOM '97*, pages 507–515, 1997.

[HNS94] Y. Hamazumi, N. Nagatsu, and K. Sato. Number of wavelengths required for optical networks with failure restoration. In *Optical Fiber Communication*, pages 67–68, Febuary 1994.

[MV80] S. Micali and V.V. Vazirani. An $O(\sqrt{V} \cdot E)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st Ann. Symp. Foundations of Computer Science*, pages 17–27, 1980.

[RS97] R. Ramaswami and A. Segall. Distributed network control for optical networks. *IEEE/ACM Transactions on Networking*, 5(6):936–943, 1997.

[RS98] Rajiv Ramaswami and Kumar N. Sivarajan. *Optical Networks: A Practical Perspective*. Academic Press/ Morgan Kaufmann, 1998.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A, chapter 10. The MIT Press, 1990.

Distributed Cooperation During the Absence of Communication*

Grzegorz Greg Malewicz[1], Alexander Russell[1], and Alex A. Shvartsman[1,2]

[1] Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269, USA.
{greg,acr,alex}@cse.uconn.edu
[2] Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

Abstract. This paper presents a study of a distributed cooperation problem under the assumption that processors may not be able to communicate for a prolonged time. The problem for n processors is defined in terms of t tasks that need to be performed efficiently and that are known to all processors. The results of this study characterize the ability of the processors to schedule their work so that when some processors establish communication, the wasted (redundant) work these processors have collectively performed prior to that time is controlled. The lower bound for wasted work presented here shows that for any set of schedules there are two processors such that when they complete t_1 and t_2 tasks respectively the number of redundant tasks is $\Omega(t_1 t_2/t)$. For $n = t$ and for schedules longer than $\sqrt{n}$, the number of redundant tasks for two or more processors must be at least 2. The upper bound on pairwise waste for schedules of length $\sqrt{n}$ is shown to be 1. Our efficient deterministic schedule construction is motivated by design theory. To obtain *linear* length schedules, a novel deterministic and efficient construction is given. This construction has the property that pairwise wasted work increases gracefully as processors progress through their schedules. Finally our analysis of a random scheduling solution shows that with high probability pairwise waste is well behaved at all times: specifically, two processors having completed t_1 and t_2 tasks, respectively, are guaranteed to have no more than $t_1 t_2/t + \Delta$ redundant tasks, where $\Delta = O(\log n + \sqrt{t_1 t_2/t}\sqrt{\log n})$.

1 Introduction

The problem of cooperatively performing a set of tasks in a decentralized setting where the computing medium is subject to failures is a fundamental problem in distributed computing. Variations on this problem have been studied in in message-passing models [3, 5, 7], using group communications [6, 9], and in shared-memory computing using deterministic [12] and randomized [2, 13, 16] models.

* This work was supported by NSF Grant CCR-9988304 and a grant from AFOSR. The work of the third author was supported by a NSF Career Award.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 119–133, 2000.

We consider the abstract problem of performing t tasks in a distributed environment consisting of n processors. We refer to this as the DO-ALL problem. The problem has simple and efficient solutions in synchronous fault-free systems; however, when failures and delays are introduced the problem becomes very challenging. Dwork, Halpern and Waarts [7] consider the DO-ALL problem in message-passing systems and use a work measure W defined as the number of tasks executed, counting multiplicities, to assess the computational efficiency. A more conservative measure [5] includes any additional steps taken by the processors, for example steps taken for coordination and waiting for messages. Communication efficiency M is gauged using the message complexity, accounting for all messages sent during the computation. It is not difficult to formulate solutions for DO-ALL in which each processor performs each of the t tasks. Such solutions have $W = \Omega(t \cdot n)$, and they do not require any communication, i.e., $M = 0$. Another extreme is the synchronous model with fail-stop processors, where each processor can send 0-delay messages to inform their peers of the computation progress. In this case one can show that $W = O(t + n \log n / \log \log n)$. This work is efficient (there is a matching lower bound, cf. [12]), and the upper bound does not depend on the number of failures. However the number of messages is more than quadratic, and can be $\Omega(n^2 \log n / \log \log n)$ [3]. Thus satisfactory solutions for DO-ALL must incorporate trade-off between communication and computation.

In failure- and delay-prone settings it is difficult to precisely control the trade-off between communication and computation. In some cases [7] it is meaningful to attempt to optimize the overall *effort* defined as the sum of work and message complexities, in other cases [5] an attempt is made to optimize efficiency in a *lexicographic* fashion by first optimizing work, and then communication. For problems where the quality of distributed decision-making depends on communication and can be traded off for communication, the solution space needs to consider the possibility of *no communication*. Notably, this is the case in the load-balancing setting introduced by Papadimitriou and Yanakakis [18] and studied by Georgiades, Mavronicolas and Spirakis [8]. In this work we study the ability of n processors to perform efficient scheduling of t tasks (initially known to all processors) during prolonged periods of *absence of communication*.

This setting is interesting for several reasons. If the communication links are subject to failures, then each processor must be ready to execute all of the t tasks, whether or not it is able to communicate. In realistic settings the processors may not initially be aware of the network configuration, which would require expenditure of computation resources to establish communication, for example in radio networks. In distributed environments involving autonomous agents, processors may *choose* not to communicate either because they need to conserve power or because they must maintain radio silence. Regardless of the reasons, it is important to direct any available computation resources to performing the required tasks as soon as possible. In all such scenarios, the t tasks have to be scheduled for execution by all processors. The goal of such scheduling must be to control redundant task executions in the absence of communication and during the period of time when the communication channels are being (re)established.

For a variation of DO-ALL Dolev *et al.* [6] showed that for the case of dynamic changes in connectivity, the termination time of any on-line task assignment algorithm can be greater than the termination time of an off-line task assignment algorithm by a factor linear in n. This means that an on-line algorithm may not be able to do better than the trivial solution that incurs linear overhead by having each processor perform all the tasks. With this observation [6] develops an effective strategy for managing the task execution redundancy and prove that the strategy provides each of the n processors with a schedule of $\Theta(n^{1/3})$ tasks such that at most one task is performed redundantly by any two processors.

In this work we advance the state-of-the-art with the ultimate goal of developing a general scheduling theory that helps eliminate redundant task executions in scenarios where there are long periods of time during which processors work in isolation. We require that all tasks are performed even in the absence of communication. A processor may learn about task executions either by executing a task itself of by learning that the task was executed by some other processor. Since we assume initial lack of communication and the possibility that a processor may never be able to communicate, each processor must know the set of tasks to perform. We seek solutions where the isolated processors can execute tasks independently such that when any two processors are able to communicate, the number of tasks they have *both* executed is as small as possible. We model solutions to the problem as sets of n lists of distinct tasks from $\{1, \ldots, t\}$. We call such lists *schedules*.

Consider an example with two processors ($n = 2$). Let the schedule of the first processor be $\langle 1, 2, 3, \ldots, t\rangle$, and the schedule of the second processor be $\langle t, t-1, t-2, \ldots, 1\rangle$. In the absence of communication each processor works without the knowledge of what the other is doing. If the processors are able to communicate after they have completed t_1 and t_2 tasks respectively and if $t_1 + t_2 \leq t$ then no work is wasted (no task is executed twice). If $t_1 + t_2 > t$, then the redundant work is $t_1 + t_2 - t$. In fact this is a lower bound on waste for any set of schedules. If some two processors have individually performed all tasks, then the wasted work is t.

Contributions. This paper presents new results that identify limits on bounded-redundancy scheduling of t tasks on n processors during the absence of communication, and gives efficient and effective constructions of bounded-redundancy schedules using deterministic and randomized techniques.

Lower Bounds. In Section 3 we show that for any n schedules for t tasks the worst case pairwise redundancy when one processor performs t_1 and another t_2 tasks is $\Omega(t_1 t_2/t)$, e.g., the pairwise wasted work grows quadratically with the schedule length, see Figure 1.(a). We also show that for $n = t$ and for schedules with length exceeding $\sqrt{n}$, the number of redundant tasks for two (or more) processors must be at least two.

When $t \gg n$ scheduling is relatively easy initially by assigning chunks of t/n tasks to each processor. Our deterministic construction focuses on the most challenging case when $t = n$.

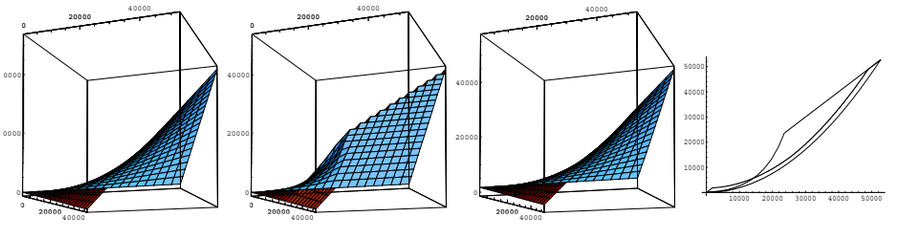


Fig. 1. Pairwise waste (redundancy) as a function of advancement through schedules for $n = t$: (a) lower bound, (b) deterministic construction (c) randomized construction, (d) diagonal vertical cut.

Deterministic Construction of Short Schedules. We show in Section 4 that it is in fact possible to construct schedules of length $\Theta(\sqrt{n})$ such that exactly *one* redundant task is performed for *any* pair of processors. This result exhibits a connection between design theory [10, 4] and the distributed problem we consider. Our design-theoretic construction is efficient and practical. The schedules are constructed by each processor independently in $O(\sqrt{n})$ time.

Deterministic Construction of Long Schedules. Design theory offers little insight on how to extend a set of schedules into longer schedules in which waste is increased in a controlled fashion. We show in Section 5 that longer schedules with controlled waste can be constructed in time linear in the length of the schedule. This deterministic construction yields schedules of length $\frac{4}{9}n$ such that pairwise wasted work increases gradually as processors progress through their schedules. For each pair of processors p_1 and p_2, the overlap of the first t_1 tasks of processor p_1 and the first t_2 tasks of processor p_2 is bounded by $O\left(\frac{t_1 t_2}{n} + \sqrt{n}\right)$. The upper bound on pairwise overlaps is illustrated in Figure 1(b). The quadratic growth in overlap is anticipated by our lower bound. The overall construction takes linear time and, except for the first $\sqrt{n}$ tasks, the cost of constructing the schedule is completely amortized.

Randomized Constructions. Finally, in Section 6, we explore the behavior of schedules selected at random. Specifically, we explore the waste incurred when each processor's schedule is selected uniformly among all permutations on $\{1, \ldots, t\}$. For the case of pairwise waste, we show that with high probability these random schedules enjoy two satisfying properties: (i) for each pair of processors p_1, p_2, the overlap of the first t_1 tasks of processor p_1 and the first t_2 tasks of processor p_2 is no more than $\frac{t_1 t_2}{t} + O\left(\log n + \sqrt{\frac{t_1 t_2}{t} \log n}\right)$, (ii) all but a vanishing fraction of the pairs of processors experience no more than a single redundant task in the first $\sqrt{t}$ tasks of their schedules. This is illustrated in Figure 1(c). As previously mentioned, the quadratic growth observed in property (i) above is unavoidable.

The results represented by the surfaces in Figures 1(a), (b) and (c) are compared along the vertical diagonal cut in Figure 1(d).

2 Definition and Models

We consider the abstract setting where n processors need to perform t independent tasks, where $n \leq t$. The processors have unique identifiers from the set $[n] = \{1, \ldots, n\}$, and the tasks have unique identifiers from the set $[t] = \{1, \ldots, t\}$. Initially each processor knows the tasks that need to be performed and their identifiers, which is necessary for solving the problem in absence of communication.

A *schedule* L is a list $L = \langle \tau^1, \ldots, \tau^b \rangle$ of distinct tasks from $[t]$, where b is the *length of the schedule* ($b \geq 0$). A *system of schedules* $\mathcal{L}$ is a list of schedules for n processors $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$. When each schedule in the system of schedules $\mathcal{L}$ has the same length b, we say that $\mathcal{L}$ has length b. Given a schedule L of length b, and $c \geq 0$, we define the *prefix schedule* L^c to be: $L^c = \langle \tau^1, \ldots, \tau^c \rangle$, if $c \leq b$, and $L^c = L$, if $c > b$. For a system of schedules $\mathcal{L}$ and a vector $\boldsymbol{a} = \langle a_1, \ldots, a_n \rangle$ ($a_i \geq 0$) a system of schedules $\mathcal{L}^{\boldsymbol{a}} = \langle L_1^{a_1}, \ldots, L_n^{a_n} \rangle$ is called a *prefix system of schedules*.

Sometimes we the order of tasks in a schedule is irrelevant, and we introduce the notion of *plan* as an unordered set of tasks. Given a schedule $L = \langle \tau^1, \ldots, \tau^a \rangle$ we define the *plan* $P = P(L)$ to be the set $P = \{\tau^1, \ldots, \tau^a\}$. Given a schedule L and $c \geq 0$, we write P^c to denote the plan corresponding to the schedule L^c (the set of the first c tasks from schedule L). For a system of schedules $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$, a *system of plans* is the list of plans $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$, where P_i is the plan for schedule L_i.

We can represent a system of plans as a matrix called a *scheme*. Specifically, given a system of plans $\mathcal{P}$ we define the *scheme* $\mathcal{S}$ to be the $n \times t$ matrix $(s_{i,j})$ such that $s_{i,j} = 1$ if $j \in P_i$, and $s_{i,j} = 0$ otherwise. Conversely, a scheme $\mathcal{S}$ yields a system of plans $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$, where $P_i = \{m : s_{i,m} = 1\}$; we say that $P_1, \ldots, P_n$ are the plans of scheme $\mathcal{S}$. A scheme is called r-regular if each row has r ones, and k-uniform if each column has k ones. Since scheme and system of plans representations are equivalent, we choose the most convenient notation depending on the context. When the ordering of tasks is important, we use the schedule representation.

To assess the quality of scheme $\mathcal{S}$, we are interested in quantifying the "wasted" (redundant) work performed by a collection I of processors when each processor i ($i \in I$) performs all tasks assigned to it by the corresponding plan P_i of $\mathcal{S}$. We formalize the notion of *waste* as follows.

Definition 1. *For a collection $I \subseteq [n]$ of processors and a scheme $\mathcal{S}$ the* I-waste *of $\mathcal{S}$, denoted $w_I(\mathcal{S})$, is defined as $w_I(\mathcal{S}) = \sum_{i \in I} |P_i| - \left|\bigcup_{i \in I} P_i\right|$, where $P_1, \ldots, P_n$ are the plans of $\mathcal{S}$.*

In general, we are interested in bounding the worst case redundant work of any set of k processors that may (re)establish communication after they perform all tasks assigned to them. Hence we introduce *k-waste* by ranging *I-waste* over all subsets I of size k:

Definition 2. *For a scheme $\mathcal{S}$ the* k-waste *of $\mathcal{S}$ is the quantity $w_k(\mathcal{S}) = \max_{I \subseteq [n], |I| = k} w_I(\mathcal{S})$.*

For a system of schedules $\mathcal{L}$ we write $w_k(\mathcal{L})$ to stand for $w_k(\mathcal{S})$, where $\mathcal{S}$ is the scheme induced by $\mathcal{L}$. In our work we are mostly interested in bounding k-waste for the case when $k = 2$. Observe that $w_{\{i,j\}}(\mathcal{S})$ is exactly $|P_i \cap P_j|$, so that in this case we are interested in controlling *overlaps*:

Definition 3. *We say that a scheme $\mathcal{S}$ is* λ-bounded *if $|P_i \cap P_j| \leq \lambda$ for all $i \neq j$. More generally, $\mathcal{S}$ is* $[\lambda, u]$-bounded *if for all sets $U \subseteq [n]$ of cardinality u we have $\left|\bigcap_{j \in U} P_j\right| \leq \lambda$. We say that $\mathcal{S}$ has* λ-overlap *(or is* λ-overlapping*) if there exists $i \neq j$ so that $|P_i \cap P_j| \geq \lambda$. More generally, $\mathcal{S}$ has* $[\lambda, u]$-overlap *if there is a set $U \subseteq [n]$ of cardinality u such that $\left|\bigcap_{j \in U} P_j\right| \geq \lambda$.*

In this work we assume that it takes unit time to add, multiply or divide two $\log(\max\{n, t\})$-bit numbers.

3 Lower Bounds on Processor-Pairs Overlaps

In this section we show lower bounds for 2-waste. We prove that 2-waste has to grow *quadratically* with the length of system of schedules, and is inversely proportional to t. This is intuitive; if $t \gg n$ then it is easy to construct n schedules of at least $\lfloor t/n \rfloor$ tasks such that the resulting scheme is 0-bounded, i.e., the 2-waste of the scheme is 0. On the other hand if $n = t$ then any system of schedules of length at least 2 must be 1-overlapping. A system of 1-bounded schedules of length $\Theta(\sqrt[3]{n})$ for $t = n$ tasks was designed by Dolev *et al.* [6]. We show that for $n = t$ no schedules can have the length greater than $\sqrt{n}$ and still be 1-bounded.

We first show a key lemma that uses a probabilistic argument (see [1] for other proofs with this flavor). Recall that given a schedule L_i, the plan P_i^a is the set of the first a tasks in L_i.

Lemma 1. *Let $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ be a system of schedules of length t, let $0 \leq a \leq t$, $0 \leq b \leq t$, and $\lambda = \max_{i \neq j} |P_i^a \cap P_j^b|$. Then $(n-1)\lambda \geq \frac{n}{t}ab - \min\{a, b\}$.*

Proof. We select i and j independently at random among $[n]$ and bound the expected value of $|P_i^a \cap P_j^b|$ in two ways. First observe that we have the total of n^2 pairs for i and j. If $i \neq j$ then the cardinality of the intersection is bounded by λ. If $i = j$ then the cardinality is obviously $\min\{a, b\}$. Hence

$$\mathbf{E}[|P_i^a \cap P_j^b|] \leq \frac{n(n-1)\lambda + n \cdot \min\{a,b\}}{n^2}$$

For the second bound we consider t random variables X_τ, indexed by $\tau \in [t]$, defined as follows: $X_\tau = 1$ if $\tau \in P_i^a \cap P_j^b$, 0 otherwise. Observe that $\mathbf{E}[|P_i^a \cap P_j^b|] = \mathbf{E}[\sum_{\tau \in [t]} X_\tau]$. By linearity of expectation, and the fact that the events are independent, we may recompute this expectation

$$\mathbf{E}[|P_i^a \cap P_j^b|] = \sum_{\tau \in [t]} \mathbf{E}[X_\tau] = \sum_{\tau \in [t]} \mathsf{Pr}[\tau \in P_i^a] \cdot \mathsf{Pr}[\tau \in P_j^b]$$

Now we introduce the function $x^m(\tau)$, equal to the number of the prefixes of schedules of length m to which τ belongs, i.e., $x^m(\tau) = |\{i : \tau \in P_i^m\}|$. Using the fact that $\mathsf{Pr}[\tau \in P_i^m] = \frac{x^m(\tau)}{n}$, and twice the Cauchy-Schwartz inequality, we can rewrite the expectation as follows.

$$\mathbf{E}[|P_i^a \cap P_j^b|] = \frac{1}{n^2} \sum_{\tau \in [t]} x^a(\tau) x^b(\tau) \geq$$

$$\frac{1}{n^2} \sqrt{\sum_{\tau \in [t]} x^a(\tau)^2} \sqrt{\sum_{\tau \in [t]} x^b(\tau)^2} \;\geq\; \frac{1}{tn^2} \sqrt{\left(\sum_{\tau \in [t]} x^a(\tau)\right)^2} \sqrt{\left(\sum_{\tau \in [t]} x^b(\tau)\right)^2}$$

Finally, since $|P_i^m| = m$, we have that $\sum_{\tau \in [t]} x^m(\tau) = m \cdot n$. Hence $\mathbf{E}[|P_i^a \cap P_j^b|] \geq \frac{ab}{t}$, and the result follows.

For any given system of schedules $\mathcal{L}$, Lemma 1 leads to a lower bound on the pairwise overlap for any two processors i and j when i performs the tasks in P_i^a and j performs the tasks in P_j^b. The lower bound in the next theorem states that the pairwise overlap must be proportional to $a \cdot b$ (see Figure 1(a) for the case when $n = t$).

Theorem 1. *Let $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ be a system of schedules of length t, and let $0 \leq a \leq t$, $0 \leq b \leq t$. Then $\max_{i \neq j} |P_i^a \cap P_j^b| \geq \lceil \frac{n}{t(n-1)} ab - \frac{\min\{a,b\}}{n-1} \rceil = \Omega(\frac{ab}{t})$.*

Immediate consequence of Theorem 1 is that 2-waste must grow quadratically with the length of the schedule. Observe that k-waste, for $k \geq 2$, must be at least as big as 2-waste, because additional processors can only increase the number of tasks executed redundantly. Hence our next result is that k-waste must grow quadratically with the length of the schedule.

Corollary 1. *If $\mathcal{L}$ is a n-processor system of schedules of length r for $t = n$ tasks, where $t \geq r$, then $w_k(\mathcal{L}) \geq \lceil \frac{r \cdot (r-1)}{n-1} \rceil$.*

Finally we show that no 1-bounded schedules exist of length greater than $\sqrt{n - 3/4} + \frac{1}{2} > \sqrt{n}$.

Corollary 2. *If $r > \sqrt{n - 3/4} + \frac{1}{2}$ then any n-processor schedule of length r for n tasks is 2-overlapping.*

This result is tight: in Section 4 we construct an infinite family of 1-bounded schedules of length $\sqrt{n - 3/4} + \frac{1}{2}$.

4 Construction of Deterministic "Square-root" Plans

We now present an efficient construction of deterministic 1-bounded schedules with maximal $\Theta(\sqrt{n})$ length, for $n = t$. In the rest of this section we assume that $n = t$.

We briefly introduce the concept of *design*, the major object of interest in design theory. A reader interested in this subject is referred to, e.g., [10]. A design is a set of n points and t blocks (subsets of points) with the following properties. Each block contains exactly k points, each point is contained in (is on) exactly r blocks, number of blocks any subset of σ points intersects (is on) is exactly λ. An object with such properties is called σ-(n, k, λ) design. A design can be represented by an *incidence matrix* $(a_{i,j})$ of zeros and ones. Numbering points and blocks, an element $a_{i,j}$ of the matrix is 1 if point i is on block j and otherwise 0. Designs have many interesting properties. One fact is that a

σ-(n,k,λ) design is also a u-(n,k,λ) design for $0 \leq u \leq \sigma$. Not surprisingly for smaller u the number of blocks a subset of u points is on increases. This number is given by[1]: $\lambda \cdot \frac{(n-u)^{\underline{\sigma-u}}}{(k-u)^{\underline{\sigma-u}}}$, (see [10] Theorem 1.2).

We now give the result linking design theory to our setting.

Theorem 2. *The incidence matrix of any σ-(n,k,λ) design with t blocks yields a $[\Lambda,u]$-bounded scheme $(0 \leq u \leq \sigma)$ for n processors and t tasks, where each processor executes $r = \frac{t}{n}k$ tasks, each task is executed k times, and $\Lambda = \lambda \cdot \frac{(n-u)^{\underline{\sigma-u}}}{(k-u)^{\underline{\sigma-u}}}$.*

Proof. Take any σ distinct points of the design. By the definition of σ-(n,k,λ) design the number of blocks on these σ points is equal to λ. Hence the number of tasks executed in common by any σ processors is exactly λ. The formula for Λ results from Theorem 1.2 [10]. This is because the design is a $(\sigma-(\sigma-u))$-(n,k,Λ) design, i.e., u-(n,k,Λ) design, for Λ as in that theorem. Moreover, since $t \cdot k = n \cdot r$ (see Corollary 1.4 [10]), each processor executes $r = \frac{t}{n}k$ tasks.

Theorem 2 makes it clear that we need to look for designs with large k and small λ because such designs yield long plans (large r) with small overlap (small Λ). We will consider a special case of this theorem for $\sigma = 2$. In this case we want to guarantee that 2-waste is exactly λ (note that when $u = \sigma = 2$, we have $\lambda = \Lambda$).

We use a well-known construction of a $2-(q^2+q+1, q+1, 1)$ design, for a prime q. The algorithm is presented in Figure 2. It has the following properties: (1) For a given number $i \in \{0,\ldots,q^2+q\}$, the value of a function $\mathsf{blocksOnPoint}(i)$ is a set of $q+1$ distinct integers from $\{0,\ldots,q^2+q\}$. (2) For $i \neq j$ the intersection of the set $\mathsf{blocksOnPoint}(i)$ with the set $\mathsf{blocksOnPoint}(j)$ is a singleton from $\{0,\ldots,q^2+q\}$. For a proof these two standard facts from design theory see e.g. [10,15]. Invoking the function $\mathsf{blocksOnPoint}(i)$ for any i requires finding at most two multiplicative inverses b^{-1} and c^{-1} in $\mathbb{Z}_q$. We can do this in $O(\log q)$ by using the Extended Euclid's Algorithm (see [14], page 325). The worst case time of finding inverses is bounded, by the Lamé theorem, by $O(\log q)$, see [14], page 343. This cost is subsumed by q iterations of the loop. Hence the total time cost of the function is $O(q)$.

Theorem 3. *If $r \cdot (r-1) = n-1$ and $r-1 = q$ is prime then it is possible to construct a r-regular r-uniform 1-bounded scheme for n processors and n tasks. Each plan is constructed independently in $O(\sqrt{n})$ time.*

Using our construction we can quickly compute schedules of size approximately $\sqrt{n}$ for n processors and $t = n$ tasks, provided we have a prime q such that $q(q+1) = n-1$. Of course in general, for a given n there may not be a prime q that satisfies $q(q+1) = n-1$. This however does not limit our construction. We discuss this in more detail in Section 5

[1] The expression $y^{\underline{x}}$ is the "falling power" defined as $y(y-1)(y-2)\ldots(y-x+1)$, with $y^{\underline{0}} = y^0 = 1$.

```
vectorToIndex( a, b, c )                    indexToVector( i )
  if a = 1 then return b · q + c              if i = q · q + q then return (0, 0, 1)
  else if b = 1 then return q · q + c         else if i ≥ q · q then return (0, 1, i − q · q)
       else return q · q + q                       else return (1, i div q, i mod q)

blocksOnPoint( i )
  (a, b, c) = indexToVector( i )
  block = ∅
  if a = 1 ∧ b ≠ 0 ∧ c ≠ 0 then block ∪= {vectorToIndex( 0, 1, − b · c^-1 )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, (−1 − c · d) · b^-1, d )}
  if a = 1 ∧ b = 0 ∧ c ≠ 0 then block ∪= {vectorToIndex( 0, 1, 0 )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, d, −c^-1, d )}
  if a = 1 ∧ b ≠ 0 ∧ c = 0 then block ∪= {vectorToIndex( 0, 0, 1 )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, −b^-1, d )}
  if a = 1 ∧ b = 0 ∧ c = 0 then block ∪= {vectorToIndex( (0, 0, 1) )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 0, 1, d )}
  if a = 0 ∧ b = 1 ∧ c ≠ 0 then block ∪= {vectorToIndex( 0, 1, −c^-1 )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, d, −d · c^-1 )}
  if a = 0 ∧ b = 1 ∧ c = 0 then block ∪= {vectorToIndex( (0, 0, 1) )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, 0, d )}
  if a = 0 ∧ b = 0 ∧ c = 1 then block ∪= {vectorToIndex( (0, 1, 0) )}
        for d = 0 to q − 1 do block ∪= {vectorToIndex( 1, d, 0 )}
  return block
```

Fig. 2. Algorithm for finding $q+1$ blocks on a point of a 2-$(q^2+q+1, q+1, 1)$ design. The notation $x \cup= y$ stands for $x = x \cup y$. Boldface font denotes arithmetic in $\mathbb{Z}_q$.

5 Constructing Long Deterministic Schedules

Applying design theory principles to constructing longer schedules is *not* necessarily a good idea. If we took a design with blocks of size $k > \sqrt{n}$ we could build a corresponding system of schedules using Theorem 2. Observe that Theorem 1 guarantees that such system would have overlap $\Omega(\frac{k^2}{n})$. Unfortunately there would be no guarantee that the overlap would increase *gradually* as processors progress through their schedules. In particular, $\Omega(\frac{k^2}{n})$ overlap may be incurred even if two processors "meet" only after executing $O(\frac{k^2}{n})$ tasks.

In this section we present a construction for longer schedules with the goal of maintaining a graceful degradation of overlap. Our novel construction extends the $\sqrt{n}$-length system of plans obtained in Theorem 3 so that the increase of overlap is controlled as the number of tasks executed by each processor grows. In the following sections we construct *raw schedules*, and then show how to use them to produce schedules with graceful degradation of overlap for arbitrary value of n.

Raw Schedules. In this section we build long raw schedules that have repeated tasks. We assume that $n = r^2 - r + 1$ and $r = q + 1$ for a prime q and use the construction from Theorem 3. Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be the resulting 1-bounded system of n plans of length r, where P_u is the plan for each $u \in \{1, \ldots, n\}$. For

a processor u ($1 \leq u \leq n$) let $L_u = \langle t_u^1, \ldots, t_u^r \rangle$ be the sequence of tasks, in some order, from the plan P_u constructed as in Theorem 3. We introduce the term *raw schedule* to denote a sequence of task identifiers where some tasks may be repeated.

We now present and analyze a system $\mathcal{R}(\mathcal{P})$ of raw schedules. For each processor u, we construct the raw schedule R_u of length $r^2 \geq n$ by concatenating ($\circ$) distinct L_i, where $i \in P_u$. Specifically, we let $R_u = L_{t_u^1} \circ L_{t_u^2} \circ \ldots \circ L_{t_u^r}$. Thus the raw schedule for processor u is $\langle t^1_{t^1_u}, .., t^r_{t^1_u}, t^1_{t^2_u}, .., t^r_{t^2_u}, \ldots\ldots, t^1_{t^r_u}, .., t^r_{t^r_u} \rangle$. Given $R_u = \langle \tau_u^1, \ldots, \tau_u^{r^2} \rangle$ we define $R_u^a = \langle \tau_u^1, \ldots, \tau_u^a \rangle$ to be the the prefix of R_u of length a, and $T_u^a = \{\tau_u^1, \ldots, \tau_u^a\}$ for $0 \leq a \leq r^2$.

A direct consequence of Theorem 3 is that raw schedules can be constructed efficiently.

Theorem 4. *Each raw schedule in $\mathcal{R}(\mathcal{P})$ can be constructed in $O(n)$ time.*

Note that it is not necessary to precompute the entire raw schedule, instead it can be computed in r-size segments as needed. Some of the tasks in a raw schedule may be repeated and consequently the number of distinct tasks in a raw schedule of length r^2 may be smaller than r^2 – naturally processors do not execute repeated instances of tasks. For the proof of graceful increase of pairwise redundancy it is important to show that the number of distinct tasks in our raw schedules increases gracefully.

Theorem 5. *For any $R_u = L_{t_u^1} \circ L_{t_u^2} \circ \ldots \circ L_{t_u^r} = \langle \tau^1, \ldots, \tau^{r^2} \rangle$ and $1 \leq a \leq r^2$: $|T_u^a| = |\{\tau^1, \ldots, \tau^a\}| \geq (\lceil \frac{a}{r} \rceil - 1)(r - \frac{1}{2}(\lceil \frac{a}{r} \rceil - 2)) + \max\{0,\ a - (\lceil \frac{a}{r} \rceil - 1)(r+1)\}$.*

Proof. Consider the task τ^a. It appears in $L_{t_u^i}$, where $i = \lceil \frac{a}{r} \rceil$. For tasks that appear in plans $P_{t_u^1}, \ldots, P_{t_u^{i-1}}$ the number of repeated tasks is at most $1 + \ldots + (i-2) = (i-1)(i-2)/2$ because $\mathcal{P}$ is a 1-bounded system of plans (any two of these plans intersect by exactly one, see Theorems 3). Hence there are at least $(i-1)r - (i-1)(i-2)/2$ distinct tasks in the raw schedule $L_{t_u^1} \circ \ldots \circ L_{t_u^{i-1}}$.

We now assess any additional distinct tasks appearing in $P_{t_u^i}$. Task τ^a is the task number $a - (i-1)r$ in $L_{t_u^i}$. Since $\mathcal{P}$ is 1-bounded, up to $i-1$ tasks in $P_{t_u^i}$ may already be contained $P_{t_u^1}, \ldots, P_{t_u^{i-1}}$. Of course in no case may the number of redundant tasks exceed $a - (i-1)r$. Hence the number of additional distinct tasks from $P_{t_u^i}$ is at least $\max\{0,\ a-(i-1)r-(i-1)\} = \max\{0,\ a-(i-1)(r+1)\}$.

Corollary 3. *Any R_u contains at least $\frac{1}{2}(r^2 + r) = \frac{1}{2}n + r - \frac{1}{2}$ distinct tasks.*

Together with Theorem 4, this result also shows that the schedule computation is fully amortized, since it takes $O(n)$ time to compute a schedule that includes more than $n/2$ distinct tasks.

For any processors u and w we wish to determine $\{u, w\}$-waste as u and w progress through the raw schedules R_u and R_w. We now show that for $1 \leq a, b \leq r^2$ the size of $T_u^a \cap T_w^b$ grows gracefully as a and b increase.

Theorem 6. *For any R_u, R_w and $0 \leq a, b \leq r^2$: $|T_u^a \cap T_w^b| \leq \min\{\ a,\ b,\ r - 1 + \lceil \frac{a}{r} \rceil \cdot \lceil \frac{b}{r} \rceil\ \}$.*

Proof. By the definition of $\mathcal{P}$ and the raw schedules R_u and R_w, $\tau_u^a \in P_{t_u^i}$, where $i = \lceil \frac{a}{r} \rceil$, and $\tau_w^b \in P_{t_w^j}$, where $j = \lceil \frac{b}{r} \rceil$. Therefore, $T_u^a \subseteq P_{t_u^1} \cup \ldots \cup P_{t_u^i}$ and $T_w^b \subseteq P_{t_w^1} \cup \ldots \cup P_{t_w^j}$. Consequently,

$$T_u^a \cap T_w^b \subseteq (P_{t_u^1} \cup \ldots \cup P_{t_u^i}) \cap (P_{t_w^1} \cup \ldots \cup P_{t_w^j}) = \bigcup_{1 \le x \le i,\ 1 \le y \le j} (P_{t_u^x} \cap P_{t_w^y}).$$

Since the system of plans $\mathcal{P}$ is 1-bounded, R_u and R_w contain at most one common P_z for some $1 \le z \le r$. In the worst case, for the corresponding P_z, this contributes $|P_z \cap P_z| = |P_z| = r$ tasks to the intersection of T_u^a and T_w^b. On the other hand, $|P_{t_u^x} \cap P_{t_w^y}| \le 1$ when both t_u^x and t_w^y are not z, again because $\mathcal{P}$ is 1-bounded. Thus, $|T_u^a \cap T_w^b| \le r + |\bigcup_{1 \le x \le i,\ 1 \le y \le j,\ x \ne y \ne z} (P_{t_u^x} \cap P_{t_w^y})| \le r + i \cdot j - 1$. Finally, the overlap cannot be greater than $\min\{a, b\}$.

In the following theorem we show how the useful work (not redundant) grows as processors progress through their schedules.

Theorem 7. *For any processors u and w:*
(a) If $i + j \le r$ then $|T_u^{(i \cdot r)} \cup T_w^{(j \cdot r)}| \ge r(i + j) - r + 1 - \frac{1}{2}((i + j)^2 + i + j)$,
(b) If $i + j > r$ then $|T_u^{(i \cdot r)} \cup T_w^{(j \cdot r)}| \ge \frac{r^2}{2} - \frac{r}{2} + \frac{9}{8}$.

Proof. By Theorem 5 $|T_u^{(i \cdot r)}| \ge i \cdot r - i(i-1)/2$ and $|T_w^{(j \cdot r)}| \ge j \cdot r - j(j-1)/2$, and by Theorem 6 $|T_u^{(i \cdot r)} \cap T_w^{(j \cdot r)}| \le r - 1 + i \cdot j$. Thus:

$$|T_u^{(i \cdot r)} \cup T_w^{(j \cdot r)}| = |T_u^{(i \cdot r)}| + |T_w^{(j \cdot r)}| \ge (i + j)(r - \tfrac{i+j-1}{2}) - r + 1$$

Consider the function $f(i + j) = f(x) = x \cdot (r - \frac{x-1}{2}) - r + 1 = -\frac{1}{2}x^2 + (r + \frac{1}{2})x + (1 - r)$. It is nonnegative for $2 \le x \le 2r$. Additionally $f(x)$ grows from r, for $x = 2$, to a global maximum of $\frac{r^2}{2} - \frac{r}{2} + \frac{9}{8}$, for $x = r + \frac{1}{2}$, and then decreases to 1, for $x = 2r$. Because $|T_u^{(i \cdot r)}|$ and $|T_w^{(j \cdot r)}|$ are monotone nondecreasing in i and j respectively (the number of tasks already performed by processors cannot decrease), we have that $|T_u^{(i \cdot r)} \cup T_w^{(j \cdot r)}| \ge \frac{r^2}{2} - \frac{r}{2} + \frac{9}{8}$ for $i + j > r$.

Deterministic Construction for Arbitrary n. We now discuss practical aspects of using the system of raw schedules $\mathcal{R}(\mathcal{P})$. Recall that a raw schedule for a processor contains repeated tasks. When a schedule is *compacted* by removing all repeated tasks, the result may contain about half of all tasks (Corollary 3). To construct a *full* schedule that has all $t = n$ distinct tasks, we append the remaining tasks at the end of a compacted schedule (in arbitrary order). For the system $\mathcal{R}(\mathcal{P})$ we call such a system of schedules $\mathcal{F}(\mathcal{P}) = \langle F_1, \ldots, F_n \rangle$. For a schedule F_i we write N_i to denote the corresponding plan. In this section we use our results obtained for raw schedules to establish a bound on pairwise overlap for $\mathcal{F}(\mathcal{P})$. Recall that by construction, the length of $\mathcal{F}(\mathcal{P})$ is $q^2 + 1 + 1$, where q is a prime. We show that common padding techniques can be used to construct schedules for arbitrary $n = t$ such that the pairwise overlap is similarly bounded.

First we analyze overlaps for a system of schedules $\mathcal{F}(\mathcal{P})$. Assume that a processor u advanced to task number $i \cdot r$ in its raw schedule R_u ($1 \le i \le r$). Then, by Theorem 5, it has executed at least $i(r - \frac{i-1}{2})$ distinct tasks. Conversely, for a given x we can define $g(x, r)$ to be the number of segments of the raw schedules R_u that are sufficient to include x distinct tasks, i.e., $|T_u^{r \cdot g(x,r)}| \ge x$. Solving the

quadratic equation $g(x,r)(r-\frac{g(x,r)-1}{2}) = x$ yields $g(x,r) = \lceil \frac{1+2r-\sqrt{(1+2r)^2-8x}}{2} \rceil$, for $x = 0, \ldots, \frac{1}{2}(r^2+r)$ (observe that $g(0,r) = 0, g(1,r) = 1, g(r,r) = 1, g(r+1,r) = 2, g(\frac{1}{2}(r^2+r),r) = r$). In the next theorem we use the definition of g and the result from Theorem 6 to construct a system of schedules with bounded overlaps (see Figure 1.b for the plot of the upper bound).

Theorem 8. *For $n = q^2+q+1$, $q = r-1$ prime, the system of schedules $\mathcal{F}(\mathcal{P})$ can be constructed deterministically in time $O(n)$ independently for each processor. Pairwise overlaps are bounded by:*

$$|N_u^a \cap N_w^b| \leq \begin{cases} \min\{a, b, r-1+g(a,r)\cdot g(b,r)\}\ , & a,b \leq \frac{1}{2}(r^2+r), \\ \min\{a,b\}, & \textit{otherwise.} \end{cases}$$

We next show that for long lengths pairwise overlap is strictly less than $\min\{a,b\}$ (the trivial part of the upper bound shown in Theorem 8). Assume that processors u and w have advanced to task number $i \cdot r$ in R_u and R_w respectively ($1 \leq i \leq r$). By Theorem 5 the number of distinct tasks executed by each processor is at least $i(r-\frac{i-1}{2})$. By Theorem 6 the overlap is at most $r-1+i^2$. Equating the two expressions yields an equation, solutions to which tell us for which i the overlap does not exceed the number of distinct tasks in the schedule. The first (trivial) solution $i = 1$ simply describes the possibility of two processors executing the same r tasks when the first task identifier in P_u is the same as that of P_w. The second solution $i = \frac{2}{3}(r-1)$, with Theorem 5, gives the number of distinct tasks in each schedule, which is no less than $\frac{4}{9}r^2+\frac{1}{9}(r-5)$. This gives guarantees that, using $\mathcal{R}(\mathcal{P})$, there are no two processors that execute the same subsets of tasks when each executes up to $\frac{4}{9}r^2+\frac{1}{9}(r-5)$ tasks. Hence as long as processors have not executed more that $\frac{4}{9}n - \Theta(\sqrt{n})$ tasks, the nontrivial part of the upper bound in Theorem 8 applies. The remaining tasks (approximately $\frac{5}{9}$ of the tasks) can be chosen by the processors arbitrarily (for example using a permutation) since our approach does not provide non-trivial overlap guarantees in that region. Note however, that for schedules longer than $\frac{4}{9}n$ the lower bound on 2-waste, by Theorem 1, is approximately $\frac{16}{81}n$, which is already linear in n.

We now discuss the case when the number of processors n is not of the form q^2+q+1, for some prime q. Since primes are dense, for any fixed $\epsilon > 0$ and sufficiently large n, we can choose[2] a prime p in $O(n)$ time such that $n-1 \leq p(p+1) \leq (1+\epsilon)n-1$. Using standard padding techniques we can construct a system of schedules of length n with overlap bounded similarly to Theorem 8. An easy analysis yields that the upper bound is strictly lower than the trivial bound as long as processors advance at most $\frac{4}{9}n - \Theta(\sqrt{n}) - \Theta(n\sqrt{\epsilon})$ through their schedules.

In our presentation we assume that a suitable prime is available. The prime can be computed as follows: Find an integer $p \in [\sqrt{n}, \sqrt{n}(1+\epsilon)]$ that satisfies: 1) $n-1 \leq p(p+1) \leq (1+\epsilon)n-1$, and 2) p is not divisible by any of $2, 3, 4, 5, \ldots, \lceil n^{1/4}(1+\epsilon) \rceil$. This gives $O(\epsilon n^{3/4})$ time algorithm. Alternatively, if

[2] This results from the Prime Number Theorem. Due to lack of space we show this in the technical report [15].

we assume the Extended Riemann Hypothesis, we can use an algorithm from [17] to find the prime in $O(\epsilon\sqrt{n}\log^4 n \log\log\log n)$. In any case the cost is expended once at the beginning of the construction, and this prime can be used multiple times so that this cost can be amortized over long-lived computations. Moreover, this cost does not distort the linear complexity of schedule construction. Finally observe that the schedules are produced in segments of size $\Theta(\sqrt{n})$. Thus if processors become able to communicate prior to the completion of all tasks then at most $\sqrt{n}$ tasks would have been scheduled unnecessarily.

6 Randomized Schedules

In this section we examine randomized schedules that, with high probability, allow us to control waste for the complete range of schedule lengths.

When the processors are endowed with a reasonable source of randomness, a natural candidate scheduling algorithm is RANDOM, where processors select tasks by choosing them uniformly among all tasks they have not yet completed. This amounts to the selection, by each processor i, of a random permutation $\pi_i \in S_{[t]}$ after which the processor proceeds with the tasks in the order given by π_i: $\pi_i(1), \pi_i(2), \ldots$. ($S_{[t]}$ denotes the collection of all permutations of the set $[t]$.)

These permutations $\{\pi_i \,|\, i \in [n]\}$ induce a system of schemes: specifically, coupled with a length $\ell_i \leq t$ for each processor i, such a family of permutations induces the plans $\mathcal{S}^i_{n,t} = \pi_i([\ell_i])$ which together comprise the scheme $\mathcal{S}[\boldsymbol{\ell}]$. Our goal is to show that these schemes are well behaved for each $\boldsymbol{\ell}$, guaranteeing that waste will be controlled. For 2-waste this amounts to bounding, for each pair i, j and each pair of lengths ℓ_i, ℓ_j, the overlap $|\pi_i([\ell_i]) \cap \pi_j([\ell_j])|$. Observe that when these π_i are selected at random, the expected size of this intersection is $\ell_i\ell_j/t$, and our goal will be to show that with high probability, each such intersection size is near this expected value. This is the subject of Theorem 9 below:

Theorem 9. *Let π_i be a family of n permutations of $[t]$, chosen independently and uniformly at random. Then there is a constant c so that with probability at least $1 - 1/n$, the following is satisfied:*

1. *$\forall i, j \leq n$ and $\forall \ell_i, \ell_j \leq t$, $|\pi_i([\ell_i]) \cap \pi_j([\ell_j])| \leq \frac{\ell_i\ell_j}{t} + \Delta$, for $\Delta = \Delta(\ell_i, \ell_j) = c\max\left(\log n, \sqrt{\frac{\ell_i\ell_j}{t}\log n}\right)$*
2. *$\forall z \leq c\log n$, the number of pairs i, j for which $\left|\pi_i([\sqrt{t}]) \cap \pi_j([\sqrt{t}])\right| > z$ is at most $\frac{n^{3/2}}{z-1}$.*

In particular, for each $\boldsymbol{\ell}$, $w_2(\mathcal{S}[\boldsymbol{\ell}]) \leq \max_{i,j} \frac{\ell_i\ell_j}{t} + \Delta(\ell_i, \ell_j)$.

Observe that Theorem 1 shows that schemes with plans of size ℓ must have ℓ^2/t overlap; hence these randomized schemes, for long regular schedules (i.e., where the plans considered have the same size), offer nearly optimal waste.

The following part of the section is devoted to proving Theorem 9. The analysis is divided into two sections, the first focusing on arbitrary pairs of lengths ℓ_i, ℓ_j, and the second focusing on specifically on "small" lengths $\ell < \sqrt{t}$.

Behavior for arbitrary lengths.
Consider two sets $A \subset [t]$ and $B \subset [t]$, A being selected at random among all sets of size $d_A\sqrt{t}$ and B at random among all sets of size $d_B\sqrt{t}$. Then $\mathsf{Exp}\left[|A \cap B|\right] = d_A d_B$. Judicious application of standard Chernoff bounds coupled with an approximation argument yields the following theorem:

Theorem 10. *Let A and B be chosen randomly as above. Then there exists a constant $c > 0$ so that for all n and $t \leq n$, $\mathsf{Pr}\left[|A \cap B| \geq d_A d_B + \Delta(d_A, d_B)\right] \leq \frac{1}{2n^5}$ where $\Delta(d_A, d_B) = c\sqrt{\log n}\left(\sqrt{d_A d_B} + \sqrt{\log n}\right)$. (The constant c is independent of t and n.)*

A proof of this fact can be found in a technical report [15]. Let c be the constant guaranteed by the above corollary and let $\mathcal{B}_{i,j}^{\ell_i,\ell_j}$ be the (bad) event that $|\pi_i([\ell_i]) \cap \pi_j([\ell_j])| \geq d_i d_j + \Delta(d_i, d_j)$, where $\ell_i = d_i\sqrt{t}$ and $\ell_j = d_j\sqrt{t}$. Let an event $\mathcal{B}_1$ be defined as disjunction $\bigvee_{i,j,\ell_i,\ell_j} \mathcal{B}_{i,j}^{\ell_i,\ell_j}$. Considering that $\mathsf{Pr}\left[\mathcal{B}_{i,j}^{\ell_i,\ell_j}\right] \leq \frac{1}{2n^5}$, we have $\mathsf{Pr}\left[\mathcal{B}_1\right] \leq n^4 \times \max_{i,j,\ell_i,\ell_j} \mathsf{Pr}\left[\mathcal{B}_{i,j}^{\ell_i,\ell_j}\right] \leq \frac{1}{2n}$. Hence

$$\mathsf{Pr}\left[\forall i, j, \ell_i, \ell_j, |\pi_i([\ell_i]) \cap \pi_j([\ell_j])| \leq d_i d_j + \Delta(d_i, d_j)\right] \geq 1 - \frac{1}{2n}$$

We now concentrate on the behavior of these schedules for lengths $\ell < \sqrt{t}$.

Behavior for short lengths.
Observe that for any pair (i, j) of schedules, $\mathsf{Exp}\left[\left|\pi_i([\sqrt{t}]) \cap \pi_j([\sqrt{t}])\right|\right] = 1$. We would like to see such behavior for each pair (i, j). Let an event $\mathcal{B}_2$ be defined as $\exists i, j \left|\pi_i([\sqrt{t}]) \cap \pi_j([\sqrt{t}])\right| \geq c_0 \log n$. From the previous argument, there is a constant c_0 so that $\mathsf{Pr}\left[\mathcal{B}_2\right] \leq \frac{1}{2n}$. Considering that the expected value of this intersection is 1, we would like to insure some degree of palatable collective behavior: specifically, we would like to see that few of these overlaps are actually larger than a constant, say. To this end, let $I_{i,j} = \left|\pi_i([\sqrt{t}]) \cap \pi_j([\sqrt{t}])\right|$, and observe that $\mathsf{Exp}\left[\sum_{i<j} I_{i,j}\right] = \binom{n}{2}$. We may write $I_{i,j} = \sum_{m=1}^{\ell} X_m$ where X_m is the indicator variable for the event $\pi_i(m) \in \pi_j([\sqrt{t}])$. Observe that these variables are negatively correlated (i.e., $\mathsf{Cov}\left[X_m, X_{m'}\right] < 0$ for each pair) so that $\mathsf{Var}\left[I_{i,j}\right] \leq \sum_{m=1}^{\ell} \mathsf{Var}\left[X_m\right] \leq \sum_{m=1}^{\ell} \mathsf{Exp}\left[X_m\right] \leq \mathsf{Exp}\left[I_{i,j}\right]$. (Recall that for any indicator variable X, $\mathsf{Var}\left[X\right] \leq \mathsf{Exp}\left[X\right]$.) Observe now that the variables $I_{i,j}$ are pairwise independent so that $\mathsf{Var}\left[\sum_{i<j} I_{i,j}\right] = \sum_{i<j} \mathsf{Var}\left[I_{i,j}\right] \leq \binom{n}{2}$, and an application of Chebyshev's inequality to the quantity $\sum_{i<j} I_{i,j}$, yields $\mathsf{Pr}\left[\sum_{i<j} I_{i,j} - \binom{n}{2} > \lambda\sqrt{\mathsf{Var}\left[\sum_{i<j} I_{i,j}\right]}\right] < \frac{1}{\lambda^2}$, so that

$$\mathsf{Pr}\left[\sum_{i<j} I_{i,j} - \binom{n}{2} > \sqrt{2}n^{1.5}\right] < \frac{1}{2n}.$$

Collecting the pieces yields Theorem 9 above, since $\mathsf{Pr}\left[\mathcal{B}_1\right] \leq \frac{1}{2n}$ and $\mathsf{Pr}\left[\mathcal{B}_2\right] \leq \frac{1}{2n}$, $\mathsf{Pr}\left[\mathcal{B}_1 \vee \mathcal{B}_2\right] \leq \frac{1}{n}$, as desired.

Acknowledgements. We thank Shmuel Zaks for motivating parts of our research, and Ibrahim Matta and Eugene Spiegel for their comments.

References

1. Alon, N., Spencer, J. H.: The probabilistic method. John Wiley & Sons Inc., New York (1992). With an appendix by Paul Erdős, A Wiley-Interscience Publication
2. Aumann, Y., Rabin, M.O.: Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation. Foundations of Comp. Sc. (1993) 147–156
3. Chlebus, B.S., De Prisco, R., Shvartsman, A.A.: Performing Tasks on Restartable Message-Passing Processors. Intl Workshop on Distributed Algorithms. Lecture Notes in Computer Science, Vol. 1320. (1997) 96–110
4. Colbourn, C. J., van Oorschot, P. C.: Applications of Combinatorial Designs in Computer Science. ACM Computing Surveys, Vol. 21. **2** (1989)
5. De Prisco, R., Mayer, A., Yung, M.: Time-Optimal Message-Efficient Work Performance in the Presence of Faults. ACM Symposium on Principles of Distributed Computing. (1994) 161–172
6. Dolev, S., Segala, R., Shvartsman, A.A.: Dynamic Load Balancing with Group Communication. Intl Colloquium on Structural Information and Communication Complexity. (1999) 111–125
7. Dwork, C., Halpern, J., Waarts, O.: Performing Work Efficiently in the Presence of Faults. SIAM J. on Computing, Vol. 27 **5**. (1998) 1457–1491
8. Georgiades, S., Mavronicolas, M., Spirakis, P.: Optimal, Distributed Decision-Making: The Case of No Communication. Intl Symposium on Fundamentals of Computation Theory. (1999) 293–303
9. Georgiou, Ch., Shvartsman A: Cooperative Computing with Fragmentable and Mergeable Groups. International Colloquium on Structure of Information and Communication Complexity. (2000) 141–156
10. Hughes, D.R., Piper, F.C.: Design Theory. Cambridge University Press (1985)
11. Ireland, K., Rosen, M.: A Classical Introdiction to Modern Number Theory. 2nd edn. Springer-Verlag (1990)
12. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)
13. Kedem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient Program Transformations for Resilient Parallel Computation via Randomization. ACM Symp. on Theory of Comp. (1992) 306–318
14. Knuth, D.E.: The Art of Computer Programming. 2nd edn. Addison-Wesley Publishing Company, Vol. 2. (1981)
15. Malewicz, G., Russell, A., Shvartsman, A.A.: Distributed Cooperation in the Absence of Communication. Technical Report MIT-LCS-TR-804 available at http://theory.lcs.mit.edu/~alex/mrsTR.ps. (Also: Brief announcement. ACM Symposium on Principles of Distributed Computing. (2000))
16. Martel, C., Park, A., Subramonian, R.: Work-optimal asynchronous algorithms for shared memory parallel computer. SIAM J. on Computing, Vol. 21 **6** (1992) 1070–1099
17. Miller, G.L.: Riemann's Hypothesis and Tests for Primality. Journal of Computer and Systems Sciences, Vol. 13 (1976) 300–317
18. Papadimitriou, C.H., Yannakakis, M.: On the value of information in distributed decision-making. ACM Symp. on Principles of Dist. Computing. (1991) 61–64

On the Importance of Having an Identity or, is Consensus really Universal? (Extended Abstract)

Harry Buhrman[1], Alessandro Panconesi[2], Riccardo Silvestri[3], and Paul Vitanyi1

[1] CWI, Amsterdam
[2] Informatica, Universitá di Bologna,
[3] Informatica, Universitá de L'Aquila

Abstract. We show that Naming– the existence of distinct IDs known to all– is a necessary assumption of Herlihy's universality result for Consensus. We then show in a very precise sense that Naming is harder than Consensus and bring to the surface some important differences existing between popular shared memory models which usually remain unnoticed.

1 Introduction

The consensus problem enjoys a well-deserved reputation in the (theoretical) distributed computing community. Among others, a seminal paper of Herlihy added further evidence in support of the claim that consensus is indeed a key theoretical construct [12]. Herlihy's paper considers the following problem: Suppose that, besides a shared memory, the hardware of our asynchronous, parallel machine is equipped with objects (instantiations) of certain abstract data types $T_1, T_2, \ldots, T_k$; given this, is it possible to implement objects of a new abstract data type Y in a wait-free manner? This question is the starting point of an interesting theory leading to many results and further intriguing questions (see [12, 14] among others). Roughly stated, one of the basic results of this theory, already contained in the original article of Herlihy, is this: If an abstract data type X, together with a shared memory, is powerful enough to implement consensus for n processes in a wait-free manner then, X, together with a shared memory, is also powerful enough to implement in a wait-free manner for n processes any other data structure Y. This is Herlihy's celebrated universality result for consensus.

In this paper we perform an analysis of some of the basic assumptions underlying Herlihy's result and discover several interesting facts which, in view of the above, are somewhat counter-intuitive and that could be provocatively be summarized by the slogans "consensus without naming is not universal" and "naming with randomization is universal." To state our results precisely we shall recall some definitions and known results.

The **naming** problem is as follows: Devise a protocol for a set of n processes such that, at the end, each non faulty process has selected a unique identifier

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 134–148, 2000.

(key). If processes have identifiers to start with then we have the **renaming** problem. Besides time and space, the size of the *name space* too– the set of possible identifiers– is considered to be a resource whose consumption is to be minimized.

We shall concern ourselves with **probabilistic protocols**– every process in the system, modelled as an i/o automaton, has access to its own source of unbiased random bits– for systems consisting of **asynchronous** processes communicating via a **shared memory**. Processes can suffer from **crash failures**. The availability of objects of abstract data type `consensus` and `naming` is assumed.

The protocols we devise are **wait-free** (i.e. $(n-1)$-resilient) in spite of the **adversary**, the "malicious" non-deterministic agent (algorithm) modeling the environment. The adversary decides which, among the currently pending operations, goes on next. Pessimistically one assumes that the adversary is actually trying to force the protocol to work incorrectly and that the next scheduling decision– which process moves next– can be based on the whole past history of the protocol execution so far. This is the so-called *adaptive* or **strong** adversary. In contrast, sometimes it is assumed that the adversary decides the entire execution schedule beforehand. This is the so-called *oblivious* or **weak** adversary.

In the literature two shared-memory models are widespread. The first assumes `multiple reader - multiple writer` registers, in which every location of the shared memory can be written or read by any process. The other model assumes `multiple reader - single writer` registers. Here, every register is owned by some unique process, which is the only process that can write on that register, while every process is allowed to read the contents of any register. If the processes use a common index scheme for other processes registers (an initial consistent numbering among the processes as it is called in [7, 8, 13]), then optimal naming is trivial by having every process rank its own number among the other values and choose that rank-number as its key. To make the problem non-trivial, we assume that each process p accesses the n register by means of a permutation π_p. That is, register π_p^i– p's ith register– will always be the same register, but, for $p \neq q$, π_p^i and π_q^i might very well differ. Besides making the problem nontrivial, this models certain situations in large dynamically changing systems where the consistency requirement is difficult or impossible to maintain [17] or in cryptographical systems where this kind of consistency is to be avoided. In both models reads and writes are atomic operations; in case of concurrent access to the same register it is assumed that the adversary "complies with" some non-deterministic, but fair, policy. In this paper we shall refer to the first as the **symmetric** memory model and to the second as the **asymmetric** memory model. It is worth pointing out that the first model is considered to be more "powerful" than the second. As we shall see, this intuition is somewhat misleading.

In this paper we show the following. *Assume that each processes has access to its own private source of independent random bits.* Then,

- **Consensus is "easy":** Assuming that (a) the memory is symmetric, and (b) processors are identical i/o automata without identifiers then, there exist

wait-free, Las Vegas consensus protocols for n process, for any $n > 1$, which work correctly even assuming the strong adversary. In fact, we exhibit a protocol whose running time (per processor) is polynomial both in expectation and with high probability.

- **Naming is "hard":** In contrast, if (a) the memory is symmetric, (b) processors are identical i/o automata without identifiers which have access to (c) consensus objects then, Las Vegas naming is impossible, even assuming the weak adversary. Note that Montecarlo naming is trivial– it is enough that each process generate $O(\log n)$ many random bits and with probability $1 - o(1)$ no two of them will be identical. [1]
- **Naming + Symmetry = Asymmetry:** If the memory is symmetric and processes have unique identifiers then the memory can be trivially made asymmetric. In this paper we show the other direction of the equivalence above namely, if the memory is asymmetric then, naming *is* possible even against the strong adversary. We exhibit a simple, modular protocol whose running time and space per process are polynomial in n, the number of processors, both in expectation and with high probability. The size of the name space is optimal, thereby improving upon a previous result of [19].

These results show that in a very precise sense, somewhat surprisingly, naming is harder, or perhaps more "fundamental", than consensus. As the second result shows, naming is impossible in a model richer than a model in which consensus is possible. Therefore naming, or some other form of asymmetry, is a necessary assumption in Herlihy's universality result. An inspection of Herlihy's construction does show that the assumption that processes have unique identifiers known to all, i.e. naming, plays a crucial role [12]. One might wonder whether in his model (deterministic, symmetric memory) names can be created from scratch. As we show in this paper, the answer is *"no!"*. In fat, they cannot even be generated if randomness (and consensus objects) are allowed. In view of this it would be more precise to restate Herlihy's result as "consensus plus naming is universal". Notice that, as the third result in the above list shows, if randomness is allowed then, asymmetric memory is powerful enough for naming. Thus, we have yet another indication that randomness increases the power of distributed systems as far as fault-tolerance is concerned.

A fundamental problem to confront with when dealing with parallel or distributed computation is "symmetry breaking." It is well-known that randomness is quite helpful in breaking the symmetry and that identifiers are another effective means to deal with the problem. Our results show that randomness is enough for consensus, a supposedly universal construct, but not for naming. Furthermore, they show that single-writer registers are inherently symmetry-breaking, whereas consensus is not. As a byproduct of our analysis we show that Herlihy's universality result does not apply to another, quite basic data type, unless naming or some other form of asymmetry is assumed. The data type in

[1] Recall that a Las Vegas protocol is always correct and that only the running time is a random variable, while for a Montecarlo protocol correctness too is a random variable.

question, which we call `selectWinner`, selects a unique winner among a set of n invoking processes. This task is impossible with symmetric memory, even if randomness and consensus are available. This result also shows in a different way how the power of randomization to "break the symmetry" is limited.

Interestingly, all randomized consensus algorithms existing in the literature known (to the authors) assume either asymmetric memory or the existence of identifiers (see, for instance, [1, 3, 4, 11]). Our results indicate that these assumptions are not necessary and that randomness is enough to create the "necessary asymmetry" as far as consensus is concerned.

Recently Chandra developed a very fast algorithm for consensus (whose asymptotic performance was subsequently improved by Aumann [9, 6]). His algorithm is much faster than the lower-bound shown by Aspnes for consensus [2]. How is this possible? Aspnes result holds for the asymmetric model, while Chandra uses several assumptions which, a priori, could be responsible for the speed-up: the availability of `multiple reader - multiple writer` registers instead of `multiple reader - single writer` registers– i.e. symmetric instead of asymmetric memory; pre-existing identifiers, and the **intermediate** adversary. This is a third kind of adversary, lying between the weak and the strong. Its behaviour is adaptive, but it has limited access to the outcome of the coin flips in that it can read the outcome of a coin flip only when this is read by some process, and not when the bit is generated (see, among others, [9]).

At first it would seem that the naming assumption must be the least important, perhaps even superfluous, for `multiple reader - multiple writer` registers certainly can cause a speed-up, and assuming a weaker adversary can circumvent impossibility results such as that of Aspnes. In fact, our results shows that, as far as Chandra's protocol is concerned, naming is necessary, for identifiers cannot be generated from scratch in his model. Aumann showed that the lower bound of Aspnes can be circumvented even assuming asymmetrc memory, i.e. `single writer- multiple reader` registers [6] We leave it as an open question whether the same holds without the naming assumption.

Our results also show that the widespread intuition that `multiple reader - multiple writer` registers are more "powerful" than `multiple reader - single writer` registers is somewhat misleading. The intuition is correct under the (quite reasonable) assumption that processes. It is however worth pointing out that the intuition is wrong without this assumption, for naming can be solved with `multiple reader - single writer` registers, but it is impossible with `multiple reader - multiple writer` registers, even if randomness is allowed. This highlights an important difference between these two models which, we feel, it is often overlooked and show that memory assumptions must be carefully stated.

Our first result, a randomized, wait-free protocol for consensus in the symmetric model (which can withstand the strong adversary) is obtained by combining several known ideas and protocols, in particular those in [3] and [9]. When compared to the protocol in [3] it is, we believe, simpler, and its correctness is easier to establish (see [20]). Moreover, it works in the less powerful symmetric

model and can deal with the strong adversary, whereas the protocol in [9] works only against the intermediate adversary. From the technical point of view, our second result is essentially contained in [16] to which we refer for other interesting related results. Related work can be found in [5, ?].

In spite of the fact that we make use of several known technical ingredients, our analysis, we believe, is novel and brings to light for the first time new and, we hope, interesting aspects of fundamental concepts.

2 Consensus is easy, Naming is hard

We start by outlining a consensus protocol assuming that (a) the memory is symmetric, (b) processes are i/o automata **without** identifiers which have access to their own source of (c) random bits. Our protocol is obtained by combining together several known ideas and by adapting them to our setting. The protocol, a randomized implementation of n-process binary consensus for symmetric memory, is a modification of the protocol proposed by Chandra [9]. The original protocol cannot be used in our setting since its shared coins require that processes have unique IDs. Thus, we combine it with a modification of the weak shared coin protocol of Aspnes and Herlihy [3]. The latter cannot be directly used in our setting either, since it requires asymmetric memory. Another difference is that, unlike in Chandra's protocol, we cannot revert to Aspnes' consensus [1]. In this paper we are only interested in establishing the existence of a polynomial protocol and make no attempt at optimization. Since the expected running time of our protocol is polynomial, by Markov's Inequality, it follows that the running time and, consequently, the space used are polynomial with high probability (inverse polynomial probability of failure). Conceivably superpolynomial space could be needed. We leave it as an open problem whether this is necessary. In the sequel we will assume familiarity with the notion of weak shared coin of [3] to which the reader is referred.

The protocol, shown in Figure 1, is based on the following idea. Processes engage in a race of sorts by splitting into two groups: those supporting the 0 value and those supporting the 1 value. At the beginning membership in the two "teams" is decided by the input bits. Corresponding to each team there is a "counter", implemented with a row of contiguous "flags"– the array of booleans MARK[,]– which are to be raised one after the other starting from the left by the team members, cooperatively and asynchronously. The variable $position_p$ of each process records the rightmost (raised) flag of its team the process knows about. The protocol keeps executing the following loop, until a decision is made. The current team of a process is defined by the variable $estimate_p$. The process first increments its own team counter by raising the $position_p$-th flag of its own team (this might have already been done by some other team member, but never mind). This means that, as far as the process is concerned, the value of its own team counter is $position_p$ (of course, this might not accurately reflect the real situation). The process then "reads" the other counter by looking at the other team's row of flags at positions $position_p + 1, position_p, position_p - 1$,

in this order. There are four cases to consider: (a) if the other team is ahead the process sets the variable $newEstimate_p$ to the other team; (b) if the two counters are equal, the process flips a fair coin $X \in \{0,1\}$ by invoking the protocol GETCOIN$_\delta(,)$ and sets $newEstimate_p$ to X; (c) if the other team trails by one, the process sticks to its team, and (d) if the other team trails by two (or more) the process decides on its own team and stops executing the protocol. Before executing the next iteration, the process checks again the counter of its own team. If this has been changed in the meanwhile (i.e. if the $(position_p + 1)$-st flag has been raised) then the process sticks to his old team and continues; otherwise, it does join the team specified by $newEstimate_p$ (which in case of a random coin flip can still be the old team). The array MARK$[i, s]$ implemented with `multiple reader - multiple writer` registers, while the other variables are local to each process and accessible to it only.

A crucial difference between our protocol and that of Chandra concerns procedure GETCOIN$_\delta(,)$. In Chandra's setting essentially it is possible to implement "via software" a *global coin*, thanks to the naming assumption. In the implementation in Figure 1, we use a protocol for a weak shared coin for symmetric memory. For every $b \in \{0,1\}$ and every $i \geq 1$ an independent realization of the weak shared coin protocol is performed. An invocation of such a protocol is denoted by GETCOIN$_\delta(b, i)$, where δ is a positive real that represents the agreement parameter of the weak shared coin (see [3]).

First, we prove that the protocol in Figure 1 is correct and efficient. Later we show how to implement the weak shared coin.

Lemma 1. *If some process decides v at time t, then, before time t some process started executing propose(v).*

Proof. The proof is exactly the same of that of Lemma 1 in [9].

Lemma 2. *No two processes decide different values.*

Proof. The proof is exactly the same of that of case (3) of Lemma 4 in [9].

Lemma 3. *Suppose that the following conditions hold:*

i) MARK$[b, i]$ = `true` *at time t,*
ii) MARK$[1 - b, i]$ = `false` *before time t,*
iii) MARK$[1 - b, i]$ *is set* `true` *at time t' ($t' > t$), and*
iv) *every invocation of both* GETCOIN$_\delta(b, i)$ *and* GETCOIN$_\delta(1 - b, i)$ *yields value b.*

Then, no process sets MARK$[1 - b, i + 1]$ *to* `true`.

Proof. The proof is essentially the same of that of the Claim included in the proof of Lemma 6 in [9].

The next lemma is the heart of the new proof. The difficulty of course is that now we are using protocol GETCOIN$_\delta(,)$ instead of the "global coins" of [9], and have to contend with the strong adversary. The crucial observation is that if

```
{Initialization}
MARK[0, 0], MARK[1, 0] ← true

{Algorithm for process p}

function propose(v): returns 0 or 1
1.  estimate_p ← v
2.  position_p ← 1
3.  repeat
4.      MARK[estimate_p, position_p] ← true
5.      if MARK[1 − estimate_p, position_p + 1]
6.          newEstimate_p ← 1 − estimate_p
7.      else if MARK[1 − estimate_p, position_p]
8.          newEstimate_p ← GETCOIN_δ(estimate_p, position_p)
9.      else if MARK[1 − estimate_p, position_p − 1]
10.         newEstimate_p ← estimate_p
11.     else return(estimate_p) {Decide estimate_p}
12.     if not MARK[estimate_p, position_p + 1]
13.         estimate_p ← newEstimate_p
14.     position_p ← position_p + 1
    end repeat
```

Fig. 1. n-process binary consensus for symmetric memory

two teams are in the same position i and the adversary wants to preserve parity between them, it must allow both teams to raise their flags "simultaneously," i.e. at least one teammate in each team must observe parity in the row of flags. But then each team will proceed to invoke GETCOIN$_\delta$(,), whose unknown outcome is unfavourable to the adversary with probability at least $(\delta/2)^2$.

Lemma 4. *If* MARK$[b, i]$ = `true` *at time* t *and* MARK$[1 - b, i]$ = `false` *before time* t*, then with probability at least* $\delta^2/4$, MARK$[1 - b, i + 1]$ *is always* `false`.

Proof. If MARK$[1 - b, i]$ is always `false`, then it can be shown that MARK$[1 - b, i + 1]$ is always `false` (the proof is the same of that of Lemma 2 in [9]). So, assume that MARK$[1 - b, i]$ is set to `true` at some time t' (clearly, $t' > t$). Since no invocation of both GETCOIN$_\delta(b, i)$ and GETCOIN$_\delta(1 - b, i)$ is made before time t, the values yielded by these invocations are independent of the schedule until time t. Thus, with probability at least $\delta^2/4$, all the invocations of GETCOIN$_\delta(b, i)$ and GETCOIN$_\delta(1 - b, i)$ yield the same value b. From Lemma 3, it follows that, with probability at least $\delta^2/4$, MARK$[1 - b, i + 1]$ is always `false`.

Theorem 1. *The protocol of Figure 1 is a randomized solution to* n*-process binary consensus. Assuming that each invocation of* GETCOIN$_\delta$(,) *costs one unit of time, the expected running time per process* $O(1)$*. Furthermore, with high probability every process will invoke* GETCOIN$_\delta$(,) $O(\log n)$ *many times.*

Proof. **(Sketch)** From Lemma 2 the protocol is consistent and from Lemma 1 it is also valid. Thus, the protocol is correct.

As regarding the expected decision time for any process, let $P(i)$ denote the probability that there is a value $b \in \{0, 1\}$ such that MARK$[b, i]$ is always `false`. From Lemma 4, it follows that

$$P(i) \geq 1 - (1 - \delta^2/4)^{i-1} \qquad i \geq 1$$

Also, if MARK$[b, i]$ is always `false`, it is easy to see that all the processes decide within $i+1$ iterations of the **repeat** loop. Thus, with probability at least $1-(1-\delta^2/4)^{i-1}$, all the processes decide within $i+1$ iterations of the **repeat** loop. This implies that the expected running time per process is O(1). The high probability claim follows from the observation that pessimistically the process describing the invocations of GETCOIN$_\delta$(,) can be modelled as a geometric distribution with parameter $p := (\delta/2)^2$.

We now come to the implementation of the weak shared coin for symmetric memory, which we accomplish via a slight modification of the protocol of Aspnes and Herlihy [3]. In that protocol the n processes cooperatively simulate a random walk with absorbing barriers. To keep track of the pebble a `distributed counter` is employed. The distributed counter is implemented with an array of n registers, with position i privately owned by process i (that is, naming or asymmetric memory is assumed). When process i wants to move the pebble it updates atomically its own private register by incrementing or decrementing it by one. The private register also records another piece of information namely, the number of times that the owner updated it (this allows one to show that the implementation of the read is linearizable). On the other hand, reading the position of the pebble is a non-atomic operation. To read the counter the process scans the array of registers twice; if the two scans yield identical values the read is completed, otherwise two more scans are performed, and so on. As shown in [3], the expected number of elementary operations (read's and write's) performed by each process is $O(n^4)$.

Since in our setting we cannot use single-writer registers, we use an array C[] of n^2 multiple-writer multiple-reader registers for the counter. The algorithm for a process p is as follows. Firstly, p chooses uniformly at random one of the n^2 registers of C[], let it be the kth. Then, the process proceeds with the protocol of Aspnes and Herlihy by using C$[k]$ as its own register and by applying the counting operations to all the registers of C[]. Snce we are using n^2 registers instead of n, the expected number of steps that each process performs to simulate the protocol is $O(n^5)$. The agreement parameter of the protocol is set to $2e\delta$. Since the expected number of rounds of the original protocol is $O(n^4)$, by Markov's Inequality, there is a constant B such that, with probability at least 1/2, the protocol terminates within Bn^5 rounds. It is easy to see that if no two processes choose the same register, then the protocol implements a weak shared coin with the same agreement parameter of the original protocol in $O(n^5)$ many steps. To ensure that our protocol will terminate in any case, if after Bn^5 steps the process

has not yet decided then it flips a coin and decides accordingly. Thus, even in case of collision the protocol terminates with a value 0 or 1 within $O(n^5)$ steps. The probability that no two processes choose the same register is

$$\left(1-\frac{1}{n^2}\right)\left(1-\frac{2}{n^2}\right)\cdots\left(1-\frac{n-1}{n^2}\right) \geq \frac{1}{e}.$$

Thus, the agreement parameter of our protocol is at least $1/2 \cdot 1/e \cdot 2e\delta = \delta$. We have proved the following fact.

Proposition 1. *For any $\delta > 0$, a weak shared coin with agreement parameter δ can be implemented in the symmetric model (with randomization) in $O(n^5)$ steps, even against the strong adversary.*

Therefore the expectd running time per process of the protocol of Theorem 1 is the same $O(n^5)$.

In contrast, no protocol exists in the symmetric model for Naming, even assuming the availability of consensus objects and the weak adversary.

Proposition 2. *Suppose that an asynchronous, shared memory machine is such that:*

- *the memory is symmetric;*
- *every process has access to a source of independent, unbiased random bits, and*
- *consensus objects are available.*

Then, still, Naming is impossible even against a weak adversary.

Proof. **(Sketch)** By contradiction suppose there exist such a protocol. Consider two processes P and Q and let only Q go. Since the protocol is wait-free there exists a sequence of steps $\sigma = s_1 s_2 \ldots s_n$ taken by Q such that Q decides on a name k_σ. The memory goes through a sequence of states $m_0 m_1 \ldots m_n$. The sequence σ has a certain probability $p_\sigma = p_1 p_2 \ldots p_n$ of being executed by Q. Start the system again, this time making both P and Q move, but one step at a time alternating between P and Q. With probability p_1^2 both P and Q will make the same step s_1. A simple case analysis performed on the atomic operations (read, write, invoke consensus) shows that thereafter P and Q are in the same state and the shared memory is in the same state m_1 in which it was when Q executed s_1 alone. This happens with probability p_1^2. With probability p_2^2, if P and Q make one more step each, we reach a situation in which P and Q are in the same state and the memory state is m_2. And so on, until, with probability p_σ^2 both P and Q decide on the same identifier, a contradiction.

Thus, naming is "harder" than consensus. The next fact shows that naming is a necessary assumption in Herlihy's universality construction. The proof is omitted from this extended abstract.

Proposition 3. *If processes are identical, deterministic i/o automata without identifiers then Naming is impossible, even if memory is asymmetric and the adversary is weak.*

3 Randomization → (Naming + Symmetry = Asymmetry)

In this section we exhibit a protocol to show the non-trivial part of the above "equation" namely, that asymmetric memory is enough for naming, provided randomization is available.

Let us start by informally describing protocol `squeeze` whose task is to assign a unique identifier to each one out of a set of n processes, using a name space of size n. For now, let us assume the availability of objects `selectWinner(i)` with the following semantics. The object can be invoked by a process p with a parameter i; the object response is to return the value *"You own key i!"* to exactly one of the invoking processes, and *"Sorry, look for another key"* to all remaining processes. The choice of the "winner" is non-deterministic. Later we will show that `selectWinner` can be implemented efficiently in a wait-free manner in our setting. With `selectWinner` a naming protocol can be easily obtained as follows: Try each key one by one, in sequence, each time invoking `selectWinner`. This is shown in Figure 2. This protocol always works but linearly many processes always perform linearly many invocations of `selectWinner`, something which could be quite expensive. Thus, we turn our attention to protocol `squeeze` which, with high probability, will only perform $O(\log^2 n)$ such invocations.

Let us then turn our attention to protocol `squeeze`. The name space is divided into *segments*, defined by the following recurrence, where p is a parameter to be fixed later:

$$s_k = p(1-p)^{k-1}n$$

s_ℓ is the last value s_i such that $s_i \geq c\log^2 n$ (c a parameter to be set by the user; the bigger the c the higher the probability that `selectWinner` will be invoked only $O(\log^2 n)$ many times). The first segment consists of the key interval $I_0 := [0, s_1)$; the second segment consists of the key interval $I_1 := [s_1, s_1 + s_2)$; the third of the key interval $I_2 := [s_1 + s_2, s_1 + s_2 + s_3)$, and so on. The final segment I_ℓ consists of the last $n - \sum_{j=1}^{\ell} s_j$ keys. In the protocol, each process p starts by selecting a *tentative key* i uniformly at random in I_0. Then, it invokes `selectWinner(i)`; if p "wins," the key becomes final and p stops; otherwise, p selects a second tentative key j uniformly at random in I_1. Again, `selectWinner(j)` is invoked and if p "wins" j becomes final and p stops, otherwise p continues in this fashion until I_ℓ is reached. The keys of I_ℓ are tried one by one in sequence. If at the end p has no key yet, it will execute the protocol `easyButExpensive` of Figure 2 as a back-up procedure. The resulting protocol appears in Figure 3. We will show that: (a) with high probability every process receives a unique key before the back-up procedure, and (b) with probability 1 every process receives a key by the end of the back-up procedure.

The intuition behind `squeeze` is this. On the one hand, if the segments are small enough then, with high probability, for every i, `selectWinner(i)` will be invoked by some process and, consequently, every key i will be assigned (all processes currently looking for a key are "squeezed" in an interval). In turn, this implies that no p will execute the backup procedure. On the other hand, if the segments are big enough , the number of intervals I_i, corresponding to the number of attempts before the backup procedure, will be small. Let us now quantify this intuition. Let p_i be defined by the following recurrence

$$p_i := (1-p)^{i-1}n.$$

If there are no crashes, the number of processes which perform the i-th attempt, i.e. those processes which select a tentative key in I_i, is obviously at least p_i. We want to show that, with high probability, it is at most p_i, for $i < \ell$. If we can show this we are done because then $p_\ell = s_\ell$ and the protocol ensures that every one of the remaining p_ℓ process will receive one of the last s_ℓ keys. A key k is *claimed* if `selectWinner(k)` is invoked by some process. Suppose first that there are no crashes. Then,

$$\Pr[\exists k \in I_i,\ k \text{ not claimed}] \leq \left(1-\frac{1}{s_i}\right)^{p_i} \leq \exp\left\{-\frac{s_i}{p_i}\right\} \leq \exp\left\{-\frac{1}{p}\right\} = \frac{1}{n^c}$$

for any fixed $c > 0$, provided that

$$p := \frac{1}{k \log n}.$$

With this value of p the number of segments, i.e. ℓ, is $O(\log^2 n)$. The expected running time is therefore

$$T(n) = O(\log^2 n)(1-n^{-k+2}) + O(n)n^{-k+2} = O(\log^2 n)$$

for $k > 3$. It can be shown that the running time is the same order of the expectation with high probability.

We now give an informal argument showing that when there are crashes the situation can only improve, deferring a formal justification to the full paper. A run of the protocol can be modelled in an equivalent fashion as follows. We have n bins, one for each key, and n white balls, one for each process. As for the key space, the n bins are organized into segments I_i. A process selects a key at random from I_i by throwing a ball in the corresponding bin interval. A key k is claimed if bin k is hit by at least one ball. A run without crashes corresponds to the following process. The n balls are thrown independently at random into the first segment I_1; for each bin which is hit a ball is selected and discarded. The remaining balls are thrown independently at random into I_2, and so on. Let us denote by p_i the number of bins from I_i which are hit. A run with crashes is modelled as follows. When the adversary crashes process p the corresponding ball

is painted red. In our balls-and-bins experiment we keep throwing all balls, red and white, but when a bin is hit we always discard a white ball. If we denote by w_i the number of processes which perform the ith attempt and by r_i the number of processes crashed before their ith attempt, then $p_i \leq r_i + w_i$. This is because for each ball which is discarded in a run without crashes we can always find a corresponding (unique) ball in the other run which corresponding to a crashed process or to a process who obtained a key. Therefore, with high probability, at the end each process is either crashed or it has received a unique key.

We now show how to implement `selectWinner` in a wait-free manner in polynomial time. We will assume the availability of objects of type `consensus(i,b)` where $1 \leq i \leq n$ and $b \in \{0,1\}$. Each invoking process p will perform the invocation using the two parameters i and b; the object response will be the same to all processes and will be *"The consensus value for i is v"* where v is one of the bits b which were proposed. This can be assumed without loss of generality since consensus can be implemented in a wait-free manner in the asymmetric model. Using consensus objects will simplify the presentation. The protocol for `selectWinner`, shown in Figure 4, is as follows. Each process p generates a bit b_1^p at random and invokes `consensus(1,`b_1^p`)`. Let v_1 be the response of the consensus object. If $b_1^p \neq v_1$ then p is a *loser* and exits the protocol. Otherwise, p is still in the game. Now the problem is to ascertain whether p is alone, in which case it is the *winner*, or if there are other processes still in the game. To this end, each remaining process scans the array $W[1,i]$, for $1 \leq i \leq n$, which is initialized to all 0's. If $W[1,i]$ contains a 1 then p declares itself a *loser* and exits; otherwise it writes a 1 in its private position $W[1,p]$ and scans $W[1,-]$ again. If $W[1,-]$ contains a single 1, namely $W[1,p]$ then p declares itself the *winner* and grabs the key, otherwise it continues the game that is, it generates a second bit b_2^p at random, invokes `consensus(2,`b_2^p`)`, and so on. The following facts establish the correctness of the protocol. Their proof is omitted from this extended abstract.

Proposition 4. *If p declares itself the winner then it is the only process to do so.*

Proposition 5. *There is always a process which declares itself the winner. Moreover, with probability $1 - o(1)$ every process p generates $O(\log n)$ many random bits b_i^p.*

These facts establish the correctness of the protocol and the high probability bound on the running time, since consensus can be implemented in the asymmetric model in polynomial time.

Remark 1: Protocol `squeeze` is also a good renaming protocol. Instead of the random bits, each process can use the bits of its own IDs starting, say, from the left hand side. Since the ID's are all different the above scheme will always select a unique winner within $O(|ID|)$ invocation of consensus.

```
protocol simpleButExpensive(): key;

begin
  for k := 1 to n do
    if selectWinner(k) = ''You own key k!'' then return(k);
end
```

Fig. 2. Simple but expensive protocol for Naming

```
protocol squeeze(): key;

begin
  for i := 1 to ℓ do begin
    k := random key in interval I_i;
    if selectWinner(k) = ''You own key k!'' then return(k);
    end;
  for k := n - s_ℓ to n do          {try key in I_ℓ one by one}
    if selectWinner(k) = ''You own key k!'' then return(k);
  return(simpleButExpensive())          {back up procedure}
end
```

Fig. 3. Protocol squeeze

```
protocol selectWinner(i: key);

myBit := ''private bit of executing process'';
attempt := 1;
repeat
  b := random bit;
  if (b = consensus(i, b)) then begin
    scan W[attempt,j] for 1 ≤ j ≤ n;
    if (W[attempt,j] = 0, for all j) then begin
      W[attempt,myBit] := 1;
      scan W[attempt,j] for 1 ≤ j ≤ n;
      if (W[attempt,j] = 0, for all j <> myBit) then return(i); {key is grabbed!}
      else attempt := attempt + 1;                              {keep trying}
    else return(''Sorry, look for another key.'');
 end repeat
```

Fig. 4. Protocol selectWinner

Remark 2: The only part of protocol **squeeze** that actually uses the memory is protocol `selectWinner`. In view of Fact 2 this task must be impossible with symmetric memory, even if randomness and consensus are available. Thus, this is another task for which, strictly speaking, Herlihy's result does not hold and it is another example of something that cannot be accomplished by the power of randomization alone.

References

1. J. Aspnes, Time- and space-efficient randomized consensus. Journal of Algorithms 14(3):414-431, May 1993.
2. J. Aspnes, Lower bounds for distributed coin-flipping and randomized consensus. Journal of the Association for Computing Machinery 45(3):415-450, May 1998.
3. J. Aspnes and M.Herlihy, Fast randomized consensus using shared memory, Journal of Algorithms 11(3):441-461, September 1990.
4. J. Aspnes and O.Waarts, Randomized consensus in $O(n \log n)$ operations per processor, SIAM Journal on Computing 25(5):1024-1044, October 1996.
5. A. Attiya, Gorbach, S. Moran, Computing in totally anonymous shared memory systems, DISC 98, LNCS 1499, pp. 49–61
6. Y. Aumann, Efficient Asynchronous Consensus with the Weak Adversary Scheduler, in Proceedings of the 16th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1997), pp. 209-218
7. A. Bar-Noy and D. Dolev, Shared Memory vs. Message-passing in an Asynchronous Distributed Environment. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, 1989, pp. 307–318.
8. E. Borowsky and E. Gafni, Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, 1993, pp. 41–52.
9. T. D. Chandra, Polylog Randomized Wait-Free Consensus, in Proceedings of the 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1996)
10. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. Journal of the ACM, 43(4):685-722, July 1996.
11. T. D. Chandra and S. Toueg, Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225-267, March 1996.
12. M. Herlihy, Wait-Free Synchronization, preliminary version in Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1988)
13. M. Herlihy and N. Shavit, The Asynchronous Computability Theorem for t-Resilient Tasks. In *Proc. 25th ACM Symp. Theory of Computing*, 1993, pp. 111–120.
14. P. Jayanti, Robust wait-free hierarchies. Journal of the ACM, 44(4):592-614, July 1997.
15. P. Jayanti and S. Toueg, Wake-up under read/write atomicity, WDAG 1990, LNCS 486, pp. 277–288.
16. S. Kutten, R. Ostrovsky and B. Patt-Shamir. The Las-Vegas Processor Identity Problem (How and When to Be Uniique), Proceedings of the 1st Israel Symposium on Theory of Computing and Systems, 1993.

17. R.J. Lipton and A. Park, Solving the processor identity problem in $O(n)$ space, *Inform. Process. Lett.*, 36(1990), 91–94.
18. Wai-Kau Lo and V. Hadzilacos. Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems, Proceedings of the 8th International Workshop on Distributed Algorithms. Terschelling, The Netherlands, September–October 1994, pp. 280–295.
19. A. Panconesi, M. Papatriantafilou, P. Tsigas and P. Vitanyi, Randomized naming using wait-free shared variables. Distributed Computing (1998) 11:113–124
20. A. Pogosyants, R. Segala and Nancy Lynch, Verification of the Randomized Consensus Algorithm of Aspnes and Herlihy: a Case Study, MIT Technical Memo number MIT/LCS/TM-555, June 1997.

Polynomial and Adaptive Long-lived $(2k-1)$-Renaming*
(Extended Abstract)

Hagit Attiya and Arie Fouren

Department of Computer Science, The Technion, Haifa 32000, Israel

Abstract. In the long-lived M-renaming problem, processes repeatedly obtain and release new names taken from a domain of size M. This paper presents the first polynomial algorithm for long-lived $(2k-1)$-renaming. The algorithm is adaptive as its step complexity is $O(k^4)$; here k is the point contention—the maximal number of simultaneously active processes in some point of the execution. Polynomial step complexity is achieved by having processes help each other to obtain new names, while adaptiveness is achieved by a novel application of *sieves*.

1 Introduction

Distributed coordination algorithms are designed to accommodate a large number of processes, each with a distinct identifier. Often, only a few processes simultaneously participate in the coordination algorithm [19]. In this case, it is worthwhile to *rename* the participating processes [6, 21]: Before starting the coordination algorithm, a process uses getName to obtain a unique *new name*—a positive integer in the range $\{1,\ldots,M\}$; the process then performs the coordination algorithm, using the new name instead of its identifier; when the coordination algorithm completes, releaseName allows the name to be re-used later.

A renaming algorithm guarantees *adaptive name space* if M is a function of k, the maximal number of processes *simultaneously* obtaining new names; k is called the *point contention*. Obviously, M should be as small as possible, preferably *linear* in k; it is known that $M \geq 2k-1$ [15, 18]. For the renaming stage to be useful it must also have *adaptive step complexity*: The number of steps a process takes in order to obtain a new name (or to release it) is a function of k. Under this definition, getName is delayed only when many processes participate simultaneously. (Precise definitions appear in Section 2.)

This paper presents an adaptive algorithm for long-lived renaming, using read and write operations. In our algorithm, a process obtains a name in the range $\{1,\ldots,2k-1\}$ with $O(k^4)$ steps. Thus, the algorithm's step complexity is a function of the maximal number of processes simultaneously active at some

* Supported by the fund for the promotion of sponsored research at the Technion. Due to space limitations, many details are omitted from this extended abstract; a full version of the paper is available through www.cs.technion.ac.il/~hagit/pubs.html.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 149–163, 2000.

point during the getName operation. This is the first long-lived renaming algorithm with optimal name space whose step complexity is polynomial in the point contention. All previous long-lived renaming algorithms providing linear name space have exponential step complexity [16].

The algorithm uses a building block called *sieve* [9]; sieves are employed in all known algorithms which adapt to point contention using read and write operations [1, 2, 4, 5]. A process tries to win in a sequence of sieves, one after the other, until successful in some sieve; when successful, the process suggests to reserve this sieve for some (possibly other) process. The sieve is reserved for one of the processes suggested for it. A process fails in a sieve only if some other process is inside this sieve; this is used to show that a process accesses sieve s only if $O(s)$ processes simultaneously participate.

Attiya et al. [6] introduce the *one-shot* renaming problem and present a $(2k-1)$-renaming algorithm for the message passing model; Bar-Noy and Dolev [10] translate this algorithm to the shared-memory model. The complexity of these algorithms is exponential [16]. Gafni [17] describes a one-shot $(2k-1)$-renaming algorithm with $O(n^3)$ step complexity. Borowsky and Gafni [12] present a one-shot $(2k-1)$-renaming algorithm with $O(N^2 n)$ step complexity.

Several adaptive renaming algorithms were suggested recently. Attiya and Fouren [7] present a $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity. Afek and Merritt [3] extend this algorithm to a $(2k-1)$-renaming algorithm with $O(k^2)$ step complexity. The $(2k-1)$-renaming algorithms of Gafni [17] and Borowsky and Gafni [12] can be made adaptive using an adaptive collect operation [8]. These algorithms are *one-shot* and adapt to the *total contention*: Their step complexity depends on the total number of operations performed so far, and does not decrease when the number of participating processes drops.

Burns and Peterson [15] present a long-lived $(2k-1)$-renaming algorithm whose step complexity is exponential [16].

Anderson and Moir [21] considered a system where many processes (N) may participate in an algorithm but in reality no more than $n \ll N$ processes are active. This paper and subsequent work [14, 20, 22] presented long-lived renaming algorithms where n is known in advance, culminating in a long-lived $(2k-1)$-renaming algorithm [20]. This algorithm employs Burns and Peterson's algorithm [15] and thus its step complexity is exponential in n.

Long-lived renaming algorithms that adapt to point contention were presented recently [1, 2, 9]. Some of these algorithms have $O(k^2 \log k)$ step complexity but they yield a name space of size $O(k^2)$; others provide linear name space (either $2k-1$ or $6k-1$ names), but their step complexity is exponential.

2 Preliminaries

In the *long-lived M-renaming* problem, processes $p_1, \ldots, p_n$ repeatedly acquire and release distinct names in the range $\{1, \ldots, M\}$. A solution supplies two procedures: getName returning a *new name*, and releaseName; p_i alternates between invoking $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$, starting with $\mathsf{getName}_i$.

Consider α, an execution of a long-lived renaming algorithm; let α' be a finite prefix of α. Process p_i is *active* at the end of α', if α' includes an invocation of $\mathsf{getName}_i$ without a return from the matching $\mathsf{releaseName}_i$. A long-lived renaming algorithm should guarantee *uniqueness* of new names: Active processes hold distinct names at the end of α'.

The *point contention* (abbreviated *contention* below) at the end of α', denoted $\mathrm{PntCont}(\alpha')$, is the number of active processes at the end of α'. Consider a finite interval β of α; we can write $\alpha = \alpha_1\beta\alpha_2$. The contention during β, denoted $\mathrm{PntCont}(\beta)$, is the maximum contention in prefixes $\alpha_1\beta'$ of $\alpha_1\beta$. If $\mathrm{PntCont}(\beta) = k$, then k processes are simultaneously active at some point during β.

A renaming algorithm has an *adaptive name space* if there is a function M, such that the name obtained in an interval of $\mathsf{getName}$, β, is in the range $\{1, \ldots, \mathrm{M}(\mathrm{PntCont}(\beta))\}$.

A renaming algorithm has *adaptive step complexity* if there is a bounded function S, such that the number of steps performed by p_i in any interval of $\mathsf{getName}_i$, β, and in the matching $\mathsf{releaseName}_i$ is at most $\mathrm{S}(\mathrm{PntCont}(\beta))$. The contention during an interval is clearly bounded by n. Therefore, $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$ terminate within a bounded number of steps of p_i, regardless of the behavior of other processes; hence, the algorithm is *wait-free*.

3 The Basic Sieve

This section describes the basic sieve [9], re-organized so it can be extended (in Section 4.2 below) to support reservations.

A sieve allows processes to obtain a view of the processes accessing the sieve concurrently, or no view (an empty view). There is a unique non-empty set of *candidates* which are seen by all processes getting a view. The sieve guarantees agreement on the information announced by candidates; this synchronizes the processes accessing the sieve and allows to exchange information in an adaptive manner.

A sieve has an infinite number of copies; at each point in the execution, processes are only "inside" a single copy of the sieve. The number of the current copy is monotonically incremented by one.

The sieve supports the following operations:

$\mathsf{read}(s.\mathit{count})$: get the number of the current copy of sieve s.
$\mathsf{openFor}(s, c)$: returns **all**, if all operations can enter copy (s, c), and $\emptyset$ if no process can enter copy (s, c).
$\mathsf{enter}(s, c, \mathit{info})$: enter copy (s, c), announcing *info*, returns the set of candidates, together with the information announced by them.
$\mathsf{exit}(s, c)$: leave the sieve and activate the next copy, $(s, c+1)$, if possible.

To access sieve s, a process first reads the number of the current copy from $s.\mathit{count}$. The process enters the sieve using enter only if $\mathsf{openFor}$ returns **all**, and leaves the sieve using exit. A process accesses sieve s in the following order:

```
1. c = read(s.count)                          // find the current copy
2. if (openFor(s, c) == all) then             // copy (s, c) is open
3.    enter(s, c, info)                       // enter (s, c) and announce info
...
4.    exit(s, c)                              //leave the sieve
```

3.1 Implementation of the Basic Sieve

Sieves are implemented using ideas from the Borowsky-Gafni simulation [11, 13], modified to guarantee adaptive step complexity.

A sieve has an infinite number of copies. At each point of the execution, candidates are inside a single copy; these processes access the sieve simultaneously.

A process tries to get inside a sieve by checking if it is among the first processes to access the current copy of this sieve. It succeeds only if this copy is free (no other process is already inside it), and no candidate is inside the previous copy. If a process does not get inside the sieve, then some concurrent process is already inside the sieve. In this manner, the sieve "catches" at least one process: one of the processes which access the current copy enters the sieve. This property makes the sieve a useful tool in adapting to contention.

For each sieve there is an integer variable, *count*, indicating the current copy of the sieve; it is initially 1. There is an infinite number of copies for each sieve, numbered $1, 2, \ldots$. Each copy has the following data structures.

- An array $R[1, \ldots, N]$ of views; all views are initially empty. Entry $R[id_i]$ contains the view obtained by process id_i in this copy.
- An array $done[1, \ldots, N]$ of Boolean variables; all entries are initially **false**. Entry $done[id_i]$ indicates whether process id_i is done with this copy.
- A Boolean variable *allDone*, initially **false**. Indicates whether all processes which could be inside this copy are done.
- A Boolean variable *inside*, initially **false**. Indicates whether some process is already inside this copy.

With each copy, we associate a separate adaptive procedure for one-shot scan of the participating processes, latticeAgreement [7], with $O(k \log k)$ step complexity.

The pseudocode appears in Algorithm 1. For simplicity, we associate a virtual Boolean variable, *allDone*, with copy 0 of every sieve, whose value is **true**.

3.2 Properties of the Basic Sieve

As mentioned before, a process accessing the sieve gets a view of the processes in the sieve concurrently with it, or gets an empty view. Among the processes which get a view, there is a unique non-empty set of *candidates* which are seen by all processes; candidates are in the same copy of the sieve simultaneously. Some candidates are *winners*, who update this copy's data structures, e.g., increment *count* by 1, in a synchronized manner.

Algorithm 1 Long-lived adaptive sieve: code for process p_i.

```
data types:
    processID: int 1 ... N                                          // process' id
    view: vector of ⟨ID : processID, INFO : information field⟩

view procedure enter(s, c: int, info: information field) // enter (s, c) with info
1:  s.inside[c]=true          // notify that there is a process inside copy (s, c)
2:  V = s.latticeAgreement[c]( info )          // announce info (from [7])
3:  s.R[c][id_i] = V                           // save the obtained view
4:  return candidates(s,c)                     // return the set of candidates

void procedure exit( s, c: int )                                    // leave the sieve
1:  s.done[c][id_i] = true                                          // p_i is done
2:  W=candidates(s,c)                          // get the set of candidates
3:  if (id_i ∈ W) then s.count = c + 1         // p_i is a winner in (s, c)
4:  if (W ≠ ∅ and ∀id_j ∈ W, s.done[c][id_j] == true) then // candidates are c
5:     releaseSieve(s, c, W)                                        // release the sieve

boolean procedure openFor(s, c)          // check whether copy (s, c) is open
1:  if (s.allDone[c − 1] and     // all candidates of the previous copy are done
          not s.inside[c])           // and no process is inside the current copy
2:  then return all                          // the copy is open for all processes
3:  else return ∅                            // the copy is open for no process

view procedure candidates(s,c)                  // returns the candidates of (s, c)
1:  V = s.R[c][id_i]
2:  W = min{s.R[c][id_j] | id_j ∈ V and s.R[c][id_j] ≠ ∅} // min by containmer
3:  if ∀id_j ∈ W, s.R[c][id_j] ⊇ W then return W
4:  else return ∅

void releaseSieve(s, c, W)       // update data structures and release the sieve
1:  s.allDone[c] = true                                             // release the sieve
```

For copy c of sieve s, procedure candidates(s, c) returns either an empty view (in Line 4) or a non-empty view (in Line 3). The key agreement property of the sieve is that all non-empty views returned by candidates(s, c) are equal:

Lemma 1. *If W_1 and W_2 are non-empty views returned by invocations of candida c) then $W_1 = W_2$.*

Process p_i is a *candidate* in copy c of sieve s if it appears in the non-empty view returned by candidates(c, s) (by Lemma 1, this view is unique). Process p_i is a *winner* in copy c of sieve s if the view obtained by p_i from candidates(s,c) contains p_i itself (see Line 2 of exit(s,c)), and in particular, is not empty. Note that a winner is in particular, a candidate.

Lemma 2. *If p_i is a candidate in copy c of sieve s, then p_i appears in every non-empty view returned by an invocation of* candidates*(s, c).*

Process p_i is *inside* copy c of sieve s when it sets $s.inside[c]$ to **true** in Line 1 of enter. A process inside copy c of sieve s is *done* after it assigns **true** to $s.done[c][id_i]$ in Line 1 of exit(s,c). The next lemma states that candidates are only inside a single copy of a specific sieve.

Lemma 3. *If process p_i is inside copy c of sieve s then all candidates of smaller copies, $1, \ldots, c-1$, of sieve s are done.*

By Lemmas 2 and 3 and the code, processes write the same values to $s.inside[c]$ and $s.allDone[c]$. Thus, each of these variables changes only once during the execution. Also, $s.count$ increases exactly by 1. This implies the next proposition:

Proposition 4 (Synchronization). *The variables of sieve s change in the following order (starting with $s.count = c-1$):*

1.	$inside[c-1] =$ ***true***	
2.	$count = c$	
3.	$allDone[c-1] =$ ***true***	(s,c) *is open*
4.	$inside[c] =$ ***true***	(s,c) *is not open*
5.	$count = c+1$	
6.	$allDone[c] =$ ***true***	$(s,c+1)$ *becomes open*
7.	$inside[c+1] =$ ***true***	$(s,c+1)$ *is not open*
8.	$count = c+2$ etc.	

It can be shown that processes enter a specific copy of a sieve *simultaneously*, and can not do so at different points. The next lemma states that at least one of the processes which are inside a copy of a sieve is a winner.

Lemma 5. *At least one of the processes which assign true to $s.inside[c]$ is a winner of copy c of sieve s.*

By Lemma 2, a non-empty view returned by candidates includes all candidates. It is possible that p_i obtains an empty view from candidates and is not a winner, yet later processes will see p_i in a non-empty view obtained from candidates.

Process p_i *accesses* copy (s,c) if it reads c from $s.count$; p_i *enters* (s,c) if it performs enter(s,c); otherwise, p_i *skips* (s,c).

The next lemma shows that a process does not win a sieve only if some concurrent process is a candidate in this sieve.

Lemma 6. *If p_i accesses sieve s and does not win, then there is a candidate of sieve s which is inside sieve s concurrently with p_i.*

The step complexity of enter is dominated by the $O(k \log k)$ step complexity of latticeAgreement [7]; the step complexity of exit is $O(k)$.

4 Polynomial Adaptive Long-lived $(2k-1)$-Renaming

Our algorithm employs a helping mechanism from the renaming algorithm of Gafni [17]. In Gafni's algorithm, a process tries to reserve names for processes with smaller priority and then obtains a name reserved for it. When there are several reservations for the same process, a reservation for a small name overrules reservations for larger names.

Gafni's algorithm is not long-lived as it has no mechanism for releasing a name or cancelling a reservation. His algorithm represents names with arrays; this is not adaptive since $O(N)$ operations are required to read the arrays.

Our algorithm extends the basic sieve (described in Section 3) to support *reservations*: If sieve s is reserved for process p_i, then only p_i can enter the sieve and get a new name s. Sieves can be entered repeatedly, in an adaptive manner, which makes them appropriate for long-lived adaptive algorithms.

As in the basic sieve, the modified sieve has a set of *candidates*; the sieve is reserved for the minimal process suggested by the candidates. In Gafni's algorithm, no reservation is made when several processes are suggested for the same name; in our algorithm, progress is made even if there is a collision since there is agreement on the reserved process.

In our algorithm, processes' priorities are based on *timestamps*. In a getName operation, process p_i acquires a timestamp TS_i, and the operation is identified by $\langle TS_i, id_i \rangle$. Timestamps are comparable, and earlier operations have smaller timestamps. In contrast, Gafni's algorithm uses static priorities for processes, based on their identifiers.

4.1 The Renaming Algorithm

In Algorithm 2, a process wishing to obtain a new name first gets a timestamp. Then, it repeatedly tries to reserve names for processes with smaller timestamps until it sees a reservation for itself. Finally, it gets a name from one of the sieves reserved for it.

To obtain the optimal name space, reservations are made only for processes which are active when getName starts. The set of active processes is maintained using the following adaptive procedures of Afek et al. [4]: In join(id_i), a process announces it is active, while in leave(id_i), a process announces it is no longer active; getSet returns the current set of active processes. It is guaranteed that processes which complete join before the beginning of getSet and start leave after the end of getSet appear in the set returned by getSet. A process completing join after the beginning of getSet or starting leave before the end of getSet may or may not appear in this set. The step complexity of these procedures is $O(k^3)$ [4].

Timestamps are obtained (in an adaptive manner) in procedure getTS, from a separate chain of sieves, using an idea we presented in [1]. A timestamp is a sequence of integers read from the counters of these sieves. Timestamps are compared by lexicographic order; if two timestamps have different lengths, then the shorter one is extended with the necessary number of $+\infty$. It can be shown that non-overlapping getTS operations return monotonically increasing timestamps.

In order to reserve a name, process p_i finds the smallest "empty" sieve which is not reserved, or which is overruled by a reservation in a smaller sieve. p_i enters the current copy of this sieve, announcing which process it is suggesting. Then, p_i computes the candidates of this copy; if all candidates of this copy are done, then p_i reserves the next copy of the sieve for the minimal process suggested by them, and exits the sieve immediately. Since processes agree on the suggestions of candidates, it is clear for which process to reserve the next copy.

To get a name for itself, p_i tries to enter sieves reserved for it. If p_i is the single winner in sieve s, then its new name is s; otherwise, p_i moves to the preceding (smaller) sieve; in this case, a smaller sieve must be reserved for p_i.

An unbounded array, $res[\langle TS_i, p_i \rangle]$, is used to notify process p_i that a reservation was made for its operation with timestamp TS_i. Section 6 discusses how to bound this array as well as the number of copies for each sieve.

4.2 The Modified Sieve

The basic sieve is modified so that if sieve s is *reserved* for an operation oid_i, then only process p_i performing oid_i can enter sieve s. The modified sieve supports the same operations as the basic sieve, with openFor extended as follows:

openFor(s, c): returns **all** if all operations can enter copy (s, c), $\emptyset$ if no process can enter copy (s, c), and oid_i if only oid_i can enter copy (s, c).

Algorithm 3 presents the modified procedures, openFor and releaseSieve.

Process p_i executing oid_i accesses the modified sieve in the same order as for the basic sieve, adapted to accommodate the change in openFor:

```
1. c = read(s.count)                          // find the current copy
2. if (openFor(s, c) == all or oid_i)   // copy (s, c) is open or reserved for oid_i
3.    then enter(s, c, info)                  // enter (s, c) and announce info
...
4.    exit(s, c)                                          //leave the sieve
```

5 Proof of Correctness

5.1 Uniqueness

The uniqueness of new names follows from properties of the basic sieve, proved in Section 3.2. In getNameForSelf, a process gets a name s only if it is a single winner in sieve s (that is, candidates returns $\{oid_i\}$). Lemmas 1 and 3 imply:

Proposition 7. *No two processes hold the same name after a finite prefix.*

Algorithm 2 Long-lived $(2k-1)$-renaming: code for process p_i.

```
data types:
    timeStamp: string of integers
    operationID: ⟨TS : timeStamp, ID : processID⟩

int procedure getName( )        // get a new name from the range 1 ... 2k − 1
1:  TS_i = getTS( )                                          // get a timestamp
2:  oid_i = ⟨TS_i, id_i⟩                                  // id of the current operation
2:  currOID[id_i] = oid_i          // announce the id of the current operation
3:  join(oid_i)                                          // join the active set [4]
4:  O = getSet()                   // get operations of the active processes [4]
5:  repeat
5:    oid_r = min{oid ∈ O | res[oid] == ⊥}                    // try to reserve for
6:    reserve(oid_r)               // the earliest operation without a reservation
7:  until (start_sieve = res[oid_i] ≠ ⊥)                      // have a reservation
8:  return getNameForSelf(oid_i, start_sieve)  // get a name in a reserved sieve

void procedure reserve(oid_r)                                // try to reserve a sieve
1:  s = 0
2:  repeat forever                                             // loop over sieves
3:    s++
4:    c = s.count
5:    if (openFor(s, c) == all)    // copy is empty and without a reservation
7:      enter(s, c, oid_r )                              // try to reserve for oid_r
8:      exit(s, c)                                                // leave the sieve
9:      return

int procedure getNameForSelf( oid_i, start_sieve ) // get a name in a reserved sieve
1:  for (s = start_sieve down to 1)
2:    c = s.count
3:    if (openFor(s, c) == oid_i) then  // (s, c) is empty and reserved for oid_i
6:      W = enter(s, c, ⊥)                    // no suggestion for the next copy
7:      if (W == {oid_i}) then return s   // p_i is the single winner in sieve s
8:      else exit(s, c)                  // a smaller sieve is reserved for oid_i

void procedure releaseName( )
1:  exit(s, c) // leave the sieve; (s, c) is remembered from the last call to enter
2:  leave(oid_i)                                          // leave the active set [4]
```

Algorithm 3 The modified sieve: new code for openFor and releaseSieve.

```
data type:
accessList: operationID ⋃ {∅, all}        // operations which can access a sieve

accessList procedure openFor(s, c)          // which operations can access (s, c)
1:  if (not s.allDone[c − 1] or s.inside[c]) then return ∅   // as in Algorithm 1
2:  oid_r = s.res[c]
3:  if (oid_r == ⊥                                                  // no reservation
4:     or for some s' ∈ {s − 1, ..., 1}, c' = s'.count and // reservation for oid_r is
          s'.allDone[c' − 1] and s'.res[c'] == oid_r  // overruled in a smaller sieve
5:     or currOID[oid_r.ID] ≠ oid_r)                  // p_r started a new operation
6:  then return all                                  // the sieve is open for any operation
7:  else return oid_r                                // the sieve is reserved for oid_r

void releaseSieve(s, c, W)   // reserve (s, c) for the earliest operation suggested
1:  oid_r = min_⟨id, oid⟩∈W oid    // earliest operation suggested by candidates
2:  s.res[c + 1] = oid_r                          // reserve the next copy for oid_r
3:  s.allDone[c] = true                                                // as in Algorithm 1
4:  res[oid_r] = s                  // notify p_r that it has a reservation in sieve s
```

5.2 Properties of the Reservations

Sieve s is *reserved* for operation oid_r after a finite prefix γ, denoted $Res(s, \gamma) = oid_r$, if and only if $s.allDone[s.count - 1] =$ **true** (candidates of the previous copy are done), $s.inside[s.count] =$ **false** (no process is in the current copy) and $s.res[s.count] = oid_r$ (the current copy is reserved for oid_r).

By Lemmas 2 and 3 and the code, winners of copy (s, c) reserve the next copy for the same operation. This implies following extension of Proposition 4:

Proposition 8 (Synchronization). *The variables of a sieve s change in the following order (starting with $s.count = c - 1$):*

1.	$inside[c-1] =$ ***true***	
2.	$count = c$	
3.	$res[c] = oid_r$	
4.	$allDone[c-1] =$ ***true***	(s, c) *becomes reserved for* oid_r
5.	$inside[c] =$ ***true***	(s, c) *is no longer reserved*
6.	$count = c + 1$	
7.	$res[c+1] = oid_{r'}$	
8.	$allDone[c] =$ ***true***	$(s, c+1)$ *becomes reserved for* $oid_{r'}$
9.	$inside[c+1] =$ ***true***	$(s, c+1)$ *is no longer reserved*
10.	$count = c + 2$ etc.	

The next lemma shows that if openFor(s, c) returns oid_i, then sieve s is indeed open for oid_i. The first three statements mean that $Res(s, \gamma) = oid_i$.

Lemma 9. *If openFor(s,c) returns oid_r, then there is a prefix γ which ends on the execution interval of Line 1 of openFor(s,c), such that $s.allDone[s.count-1] =$ **true**, $s.inside[s.count] =$ **false**, $s.res[s.count] = oid_r$, and $s.count = c$ at the end of γ.*

$MinRes(oid_i, \gamma)$ denotes the minimal sieve reserved for an operation oid_i (of process p_i) after an execution prefix γ. Since a reservation for oid_i in a smaller sieve overrules reservations for this operation in larger sieves, $MinRes(oid_i, \gamma)$ is the actual reservation for oid_i. For convenience, it is ∞ if no sieve is reserved for oid_i. It is 0 after p_i enters the copy of the sieve from which it gets its new name; i.e., p_i writes **true** into $s.inside[s.count]$, where $s.count$ is the copy of sieve s from which p_i gets its new name.

We can prove that if openFor(s,c) returns **all**, then either there is no reservation in sieve s or the reservation is overruled by a smaller sieve. This allows to show that a process does not destroy the minimal reservation for another process. This is the key to proving that *MinRes* monotonically decreases:

Lemma 10. *Assume γ is a finite prefix. Then for every prefix γ' of γ and every operation oid_i, $MinRes(oid_i, \gamma') \geq MinRes(oid_i, \gamma)$.*

The next lemma shows that openFor recognizes the minimal reservation.

Lemma 11. *If openFor(s,c) returns oid_r, then there is a prefix γ which ends on the execution interval of openFor(s,c), such that $MinRes(oid_r, \gamma) = s$.*

Proof. By Lemma 9 there is a prefix γ which ends on the execution interval of Line 1 of openFor such that $Res(s, \gamma) = oid_r$. Therefore, $MinRes(oid_r, \gamma) \leq s$.

Assume, by way of contradiction, that $MinRes(oid_r, \gamma) = s' < s$. Let γ' be the shortest prefix such that $MinRes(oid_r, \gamma') = s'$. Suppose that p_i reads $s-1.count, \ldots, 1.count$ (in Line 4 of openFor) at the ends of prefixes $\gamma_{s-1}, \ldots, \gamma_1$, respectively. By the code, $\gamma_{s-1}, \ldots, \gamma_1$ end after the end of γ.

If $\gamma_{s'}$ is included in γ', then by Lemma 10, $s > MinRes(oid_r, \gamma_{s-1}) \geq \ldots \geq MinRes(oid_r, \gamma_{s'}) \geq s'$. By the pigeon hole principle, $MinRes(oid_r, \gamma_\ell) = \ell$ for some $\ell \in \{s-1, \ldots, s'\}$. By the sieve's properties, p_i reads $\ell.allDone[c-1] =$ **true**, and $\ell.res[c] = oid_r$ (since these variables do not change after γ_ℓ). Thus, the condition in Line 4 of openFor holds for p_i and openFor returns **all**. This is a contradiction, and the lemma follows.

Otherwise, $\gamma_{s'}$ is not included in γ'. Since s' is the minimal sieve reserved for oid_r, no process writes to $s'.count$ from the end of γ' until p_r writes a new value to $currOID[oid_r]$. Since p_i reads $s'.count$ before p_r writes a new value to $currOID[oid_r]$, the sieve's properties imply that p_i reads **true** from $s'.allDone[c-1]$, and oid_r from $s'.res[c]$ (since these variables do not change after $\gamma_{s'}$). Thus, the condition in Line 4 of openFor holds for p_i and openFor returns **all**. This is a contradiction, and the lemma follows. □

5.3 Size of the Name Space

Let δ_i be an interval of $\mathsf{getName}_i$; p_i has several attempts (calls to reserve) to make a reservation; let β_i be the interval of one invocation of reserve. Partition β_i into disjoint intervals during which p_i accesses different sieves: If p_i accesses sieve s then $\beta_i|_s$ is the interval which starts when p_i reads from $s.count$ (or the beginning of β_i, if $s = 1$) and ends when p_i reads from $s + 1.count$ (or the end of β_i, if p_i is a winner in sieve s).

The next lemma shows that when a process skips a sieve in reserve, either some concurrent process is a candidate in this sieve or the sieve is reserved for another process. It follows from Lemmas 6 and 11 and the code.

Lemma 12. *If in reserve p_i skips sieve (s, c) in reserve then either*
(1) there is a candidate in (s, c) concurrently with p_i, or
(2) there is an operation oid_r such that $MinRes(oid_r, \gamma) = s$, for some prefix γ ending in $\beta|_s$.

A potential method is used to show that only sieves $< 2k$ are reserved; this bounds the name space, since a process obtains its new name only from a sieve reserved for it. We define two sets of simultaneously active operations and show that at least one of them grows by 1 when p_i skips a sieve in reserve. Each set contains at most $k - 1$ concurrent operations (except p_i); this implies p_i skips at most $2k - 1$ sieves, and makes a reservation in a sieve $\leq 2k$. However, as we show, the *minimal* reservation for a process is always $< 2k$.

The first set, E_s, contains operations whose attempt ends in a sieve $\leq s$, while the second set, R_s, contains operations reserved in a sieve $\leq s$. Formally, for a prefix γ and a process p_i which performs an attempt with interval β_i, define:

$$E_s(\gamma) = \{oid_q \mid q \text{ is a candidate in a sieve} \leq s \text{ whose attempt covers the end of } \gamma\}$$

$$R_s(\gamma, p_i) = \{oid_q \mid q \text{ is active at the end of } \gamma \text{ and } MinRes(oid_q, \beta_i|_s) \leq s\}$$

E contains operations whose *attempt* covers the end of γ while R contains operations whose *whole* execution interval covers the end of γ. This reflects the fact that E depends on an operation's execution, while R depends on the minimal sieve reserved for an operation by others. In the definition of R_s, γ is used to determine which operations are active, but *MinRes* is evaluated at the end $\beta_i|_s$.

The next key lemma shows that a process accesses a sieve with a high index only if many processes are active concurrently with it. The proof is by induction on the sieve's number, and considers the different cases in which a process skips a sieve; the proof is omitted due to lack of space.

Lemma 13. *Let β_i be the interval of an attempt by p_i. If p_i skips sieve s, then there is a prefix γ which ends in $\beta_i|_1 \beta_i|_2 \ldots \beta_i|_s$ such that $|E_s(\gamma)| + |R_s(\gamma, p_i)| \geq s$.*

The next lemma shows that the minimal sieve reserved for a process is $< 2k$.

Lemma 14. *Let δ_i be the execution interval of oid_i of p_i If for some prefix γ_i, $MinRes(oid_i, \gamma_i)$ is not ∞, then $MinRes(oid_i, \gamma_i) < 2k$, where $k = PntCont(\delta_i)$.*

Proof. Let $MinRes(oid_i, \gamma_i) = s$, and let p_j be a process which suggests oid_i in sieve s. Let $\beta_j|_s$ be the interval of p_j's attempt in which it skips sieves $1, \ldots, s-1$ and suggests oid_i in $(s, c-1)$. By Lemma 13, $s-1 \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)|$, for some prefix γ_j which ends in $\beta_j|_s$. Let $k = \mathrm{PntCont}(\beta_j|_s)$; $k \leq \mathrm{PntCont}(\delta_i)$, since $\beta_j|_s$ is contained in δ_i. By definition, $R_{s-1}(\gamma_j, p_j)$ and $E_{s-1}(\gamma_j)$ contain operations which are simultaneously active at the end of γ_j; hence, $|R_{s-1}(\gamma_j, p_j)| \leq k-1$. By definition, $oid_j \notin E_{s-1}(\gamma_j)$, which implies that $|E_{s-1}(\gamma_j)| \leq k-1$.

If $oid_i \in R_{s-1}(\gamma_j, p_j)$, then by the definition of R_{s-1}, $MinRes(oid_i, \beta_j|_{s-1}) \leq s-1$. The properties of the sieve imply that p_j accesses $(s, c-1)$ before sieve s is reserved for oid_i. That is, $\beta_j|_{s-1}$ ends before γ_i. Therefore, by Lemma 10, $MinRes(oid_i, \gamma_i) \leq MinRes(oid_i, \beta_j|_{s-1}) \leq s-1$. The lemma follows since $s-1 \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)| \leq 2k-1$.

If $oid_i \notin R_{s-1}(\gamma_j, p_j)$, then $|R_{s-1}(\gamma_j, p_j)| \leq k-1$. In this case, the lemma follows since $s \leq |R_{s-1}(\gamma_j, p_j)| + |E_{s-1}(\gamma_j)| + 1 \leq 2k-1$. □

This proves that the minimal reservation for an operation is in a sieve $< 2k$. We now show that p_i succeeds to get a name from one of the sieves reserved for it. The next lemma states that if p_i skips sieve s in getNameForSelf, then a smaller sieve is reserved for p_i.

Lemma 15. *Assume that in getNameForSelf, p_i reads s.count at the end of prefix γ_s and $(s-1)$.count at the end of prefix γ_{s-1}; that is, p_i skips sieve s. If $MinRes(oid_i, \gamma_s) \leq s$ then $MinRes(oid_i, \gamma_{s-1}) \leq s-1$.*

Finally, we prove that the algorithm provides optimal name space.

Lemma 16. *getNameForSelf returns a new name in the range $\{1, \ldots, 2k-1\}$.*

Proof. By the algorithm, p_i calls getNameForSelf (Line 8 of getName) after it reads a non-$\perp$ value, say, ss, from $res[oid_i]$. Sieve ss is reserved for oid_i before ss is written to $res[oid_i]$.

Let s_m be the minimal value of $MinRes(oid_i)$. In getNameForSelf, p_i accesses sieves $ss, ss-1, \ldots$. If p_i does not win some sieve s, then Lemma 15 implies that a smaller sieve is reserved for p_i. Thus, p_i wins sieve s_m at latest and getNameForSelf returns a name $s > 0$.

It remains to show that the new name is $< 2k$. By the algorithm, openFor(s, c) returns oid_i (Line 3). Lemma 11 implies that $MinRes(oid_i, \gamma) = s$, for some prefix γ. By Lemma 14, $MinRes(oid_i, \gamma) < 2k$, and hence, $s < 2k$. □

5.4 Step Complexity

Let $elected(s, c)$ be the minimal operation identifier suggested by candidates in copy (s, c). By Lemma 1, processes agree on the set of candidates and the next copy of the sieve $(s, c+1)$ is reserved for $elected(s, c)$. If process p_i gets its new name from copy (s, c), then p_i is the single candidate in (s, c) and it writes $\perp$ in Line 5 of exit; in this case, $elected(s, c) = \perp$. Since a process makes suggestions in one sieve at a time, we can bound the number of different copies in which the same operation is elected.

Lemma 17. *An operation is elected in at most $2k$ copies.*

Line 6 of getName guarantees that p_i repeatedly calls reserve as long as no reservation was made for it. To bound the number of attempts p_i makes, we bound the number of different operations encountered by p_i as $elected(s, c)$ and the number of copies they are elected in. Using similar arguments, we also prove that $O(k)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = \perp$.

Lemma 18. *$O(k^2)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = oid_j \neq \perp$. $O(k)$ attempts of p_i end in a copy (s, c) such that $elected(s, c) = \perp$.*

Thus, p_i discovers that a sieve is reserved for oid_i and calls getNameForSelf after $O(k^2)$ attempts. In each attempt, p_i skips $O(k)$ sieves and enters one sieve; in getNameForSelf, p_i enters $O(k)$ sieves. Skipping a sieve requires $O(k)$ steps (Line 9 of reserve); entering a sieve requires $O(k \log k)$ steps [7]. Therefore, the total step complexity of the algorithm is $O(k^2[k \cdot k + k \log k] + k \cdot k \log k) = O(k^4)$.

6 Discussion

We present a long-lived $(2k-1)$-renaming algorithm with $O(k^4)$ step complexity, where k is the maximal number of simultaneously active processes in some point of the execution.

The algorithm described here uses an unbounded amount of shared memory. First, each sieve has an unbounded number of copies. Copies can be re-cycled (and hence bounded), see [1]. Second, an unbounded array is used to notify a process about reservations for each of its operations. This array can be replaced by a two-dimensional array in which each process announces for each other process the last timestamp it reserved for (and in which sieve). A process checks, for every process in the current active set whether a reservation was made for its current timestamp; then it tries to enter from the minimal reservation made for it, if there are any. The details are postponed to the full version of the paper.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Available at www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz, Apr. 1999.
2. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 91–103, 1999.
3. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 105–112, 1999.
4. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. In *Proc. 40th IEEE Symp. Foundations of Comp. Sci.*, pages 262–272, 1999.

5. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symp. Principles of Dist. Comp.*, 2000. To appear.
6. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.
7. H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th ACM Symp. Principles of Dist. Comp.*, pages 277–286, 1998.
8. H. Attiya and A. Fouren. An adaptive collect algorithm with applications. Submitted for publication. Available at www.cs.technion.ac.il/~hagit/pubs/AF99ful.ps.gz, Aug. 1999.
9. H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 956, Faculty of Computer Science, The Technion, Mar. 1999. Available at www.cs.technion.ac.il/~hagit/pubs/tr0956.ps.gz.
10. A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Math. Sys. Theory*, 26(1):21–39, 1993.
11. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. Theory of Comp.*, pages 91–100, 1993.
12. E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. Principles of Dist. Comp.*, pages 41–52, 1993.
13. E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. Technical Report MIT/LCS/TM-573, Laboratory for Computer Science, MIT, Dec. 1997.
14. H. Buhrman, J. A. Garay, J.-H. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proc. 14th ACM Symp. Principles of Dist. Comp.*, pages 194–203, 1995.
15. J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. Principles of Dist. Comp.*, pages 145–158, 1989.
16. A. Fouren. Exponential examples for two renaming algorithms. Available at www.cs.technion.ac.il/~hagit/pubs/expo.ps.gz, Aug. 1999.
17. E. Gafni. More about renaming: Fast algorithm and reduction to the k-set test-and-set problem. Unpublished manuscript, 1992.
18. M. Herlihy and N. Shavit. A simple constructive computability theorem for wait-free computation. In *Proc. 26th ACM Symp. Theory of Comp.*, pages 243–252, 1994.
19. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Feb. 1987.
20. M. Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Programming*, 30(3):287–308, May 1998.
21. M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.
22. M. Moir and J. A. Garay. Fast long-lived renaming improved and simplified. In *Proc. 10th Int. Workshop on Dist. Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 1996.

Computing with Infinitely Many Processes

under assumptions on concurrency and participation

(Extended abstract)

Michael Merritt* Gadi Taubenfeld**

Abstract. We explore four classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. We survey and generalize work carried out in models with finite bounds on the number of processes, and prove several new results. These include improved bounds for election when participation is required and a new adaptive algorithm for starvation-free mutual exclusion in a model with unbounded concurrency. We also explore models where objects stronger than atomic registers, such as test&set bits, semaphores or read-modify-write registers, are used.

1 Introduction

1.1 Motivation

We explore several classic problems in concurrent computing (election, mutual exclusion, and consensus) when the number of processes which may participate is infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. This paper surveys and generalizes work carried out in models with finite bounds on the number of processes, and proves several new results. These include improved bounds for election when participation is required and a new adaptive algorithm for starvation-free mutual exclusion when the number of concurrent processes is not bounded *a priori*.

Processes: In most work on the design of shared memory algorithms, it is assumed that the number of processes is finite and *a priori* known. Here we investigate the the design of algorithms assuming no *a priori* bound on the number of processes. In particular, we assume the number of processes may be infinite. Although, in practice, the number of processes is always finite, algorithms designed for an infinite number of processes (when possible) may scale well: their time

* AT&T Labs, 180 Park Av., Florham Park, NJ 07932-0971. mischu@research.att.com.

** The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs. gadi@cs.openu.ac.il.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 164–178, 2000.

complexity may depend on the actual contention and not on the total number of processes. Starvation-free mutual exclusion has been solved in such a model using two registers and two weak semaphores (see Theorem 15) [FP87]. Wait-free solvability of tasks when there is no upper bound on the number of participating processes has also been investigated [GK98], but in this earlier work no run has an infinite number of participating processes. A model in which processes are continuously being created and destroyed (and hence their number is not *a priori* known) has also been considered [MT93].

Concurrency: An important factor in designing algorithms where the number of processes is unknown, is the *concurrency level,* the maximum number of processes that may be active simultaneously. (That is, participating at the same instant of time.[3]) We distinguish between the following concurrency levels:

- *finite:* there is a finite bound (denoted by c) on the maximum number of processes that are simultaneously active, over all runs. (The algorithms in this paper assume that c is known.)
- *bounded:* in each run, there is a finite bound on the maximum number of processes that are simultaneously active.
- *unbounded:* in each run, the number of processes that are simultaneously active is finite but can grow without bound.
- *infinite:* the maximum number of processes that are simultaneously active may be infinite. (A study of infinite concurrency is mainly of theoretical interest.)

The case of infinite concurrency raises some issues as to the appropriate model of computation and semantics of shared objects. We assume that an execution consists of a (possibly infinite) sequence of *group steps*, where each group step is itself a (possibly infinite) sequence of steps, but containing at most *one* primitive step by each process. Hence, no primitive step has infinitely many preceding steps by any single process. The semantics of shared objects need to be extended (using limits) to take into account behavior after an infinite group step. This leaves some ambiguity when a natural limit does not exist. For example, what value will a read return that follows a step in which all processes write different values? Natural limits exist for all the executions we consider. In particular, in the algorithm in Figure 5, the only write steps assign the value 1 to shared bits. The well-ordering of primitive steps in an execution of this model assures that there is a first write to any bit, and all subsequent reads will return 1.

Participation: When assuming a fault-free model with *required participation* (where every process has to start running at some point), many problems are solvable using only constant space. However, a more interesting and practical

[3] We have chosen to define the concurrency level of a given algorithm as the maximum number of processes that can be simultaneously active. A weaker possible definition of concurrency, which is not considered here, is to count the maximum number of processes that actually take steps while some process is active.

situation is when participation is not required, as is usually assumed when solving resource allocation problems. For example, in the mutual exclusion problem a process can stay in the reminder region forever and is not required to try to enter its critical section.

We use the notation $[\ell, u]$-*participation* to mean that at least ℓ and at most u processes participate. Thus, for a total of n processes (where n might be infinite) $[1, n]$-participation is the same as saying that participation is not required, while $[n, n]$-participation is the same as saying that participation is required. Requiring that all processes must participate does not mean that there must be a point at which they all participate at the same time. That is, the concurrency level might be smaller that the upper bound on the number participating processes. Notice also that if an algorithm is correct assuming $[\ell, u]$-participation, then it is also correct assuming $[\ell', u']$-participation, where $\ell \leq \ell' \leq u' \leq u$. Thus, any solution assuming that participation is not required, is correct also for the case when participation is required, and hence it is expected that such solutions (for the case where participation is not required) may be less efficient and harder to construct.

1.2 Properties

We define two properties of algorithms considered in the paper.

Adaptive algorithms: An algorithm is *adaptive* if the time complexity of processes' operations is bounded by a function of the actual concurrency. The term *contention sensitive* was first used to describe such algorithms [MT93], but later the term *adaptive* become commonly used. Time complexity is computed using the standard model, in which each primitive operation on an objects is assumed to take no more than one time unit. In the case of mutual exclusion algorithms, we measure the maximum time between releasing the critical section, until the critical section is re-entered. For models in which there is a minimum bound on the number of participating processes, we measure one time unit to the first point at which the minimum number of processes have begun executing the algorithm.

Symmetric algorithms: An algorithm is *symmetric* if the only way for distinguishing processes is by comparing (unique) process identifiers. Process id's can be written, read, and compared, but there is no way of looking inside any identifier. Thus, identifiers cannot be used to index shared registers. Various variants of symmetric algorithms can be defined depending on how much information can be derived from the comparison of two unequal identifiers. In this paper we assume that id's can only be compared for equality. (In particular, there is no total order on process id's in symmetric algorithms.)[4]

[4] Styer and Peterson, [SP89], discuss two types of symmetry. Our notion corresponds to their stronger restriction, "symmetry with equality". Some of our asymmetric algorithms, presented later, work in systems with their weaker symmetry restriction, "symmetry with arbitrary comparisons," in that they depend upon a total order of the process id's, but do not (for example) use id's to index arrays.

1.3 Summary of results

We assume the reader is familiar with the definitions of the following problems:

1. The mutual exclusion problem, which is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65];
2. The consensus problem, which is to design a protocol in which all correct processes reach a common decision based on their initial opinions [FLP85];
3. The (leader) election problem, which is to design a protocol where any fair run has a finite prefix in which all processes (correct or faulty) commit to some value in $\{0, 1\}$, and exactly one process (the leader) commits to 1; We do not require the leader to be identified to the other processes, although we do require all processes to terminate in fair runs. In a model with infinitely many processes, identifying the leader obviously requires infinite space–this weaker assumption makes the lower bounds more complex. To prevent trivial solutions, it is assumed that the ids are not initially known to the processes, although we do assume that process identities are natural numbers.
4. The *wait-free naming* problem which is to assign unique names to initially identical processes. Every participating process is able to get a unique name in a finite number of steps regardless of the behavior of other processes.

We show that even with a fixed bound on concurrency and required participation, election (using registers) requires infinite shared space. Among the novel algorithms presented are two demonstrating that either a single shared register of infinite size, or infinitely many shared bits, suffice for both election and consensus. (In addition, the first algorithm is adaptive.) If in addition test&set bits are used, then solving the above problem requires only finite space; however, a result of Peterson ([Pet94]) implies that the more complex problem of starvation-free mutual exclusion (bounded concurrency, participation not required) still requires infinite space. In fact, even using read-modify-write registers to solve this problem, a result of Fischer *et al* ([F$^+$89]) implies that infinite space is required. However, Friedberg and Peterson ([FP87]) have shown that using objects such as semaphores that allow waiting enables a solution with only constant space.

When there is no upper bound on the concurrency level, we show that even infinitely many test&set bits not suffice to solve naming. Naming can be solved assuming bounded concurrency using test&set bits only, hence it separates bounded from unbounded concurrency. At the boundary of unbounded and infinite concurrency, Friedberg and Peterson ([FP87]) have shown a separation for starvation-free mutual exclusion using registers and weak semaphores: two of each suffice for unbounded concurrency, but the problem cannot be solved using these primitives for infinite concurrency.

The tables below summarize the results discussed in this paper. As indicated, many are derived from existing results, generally proven for models where the number of processes is finite and known. We use the following abbreviations: DF for deadlock-free; SF for starvation-free; mutex for mutual exclusion; U for Upper bound; L for Lower bound; RW for atomic read/write registers; T&S

for test&set bits; wPV for weak semaphores; and RMW for read-modify-write registers. (The default is "No" for the adaptive and symmetric columns. All lower bounds hold for non-adaptive and asymmetric case.)

Problem			Model		Result					
					Properties		Space		Thm	Using
	Name	Bound	Concurrency $c > 1$	Participation	adaptive?	symmetric?	#	size	#	results from
1	ElectioN	L_1	c	$[c, \infty]$			∞ state space		1	
		U	c	$[c, \infty]$			∞	2	2	
	(Upper	U	2	$[2, \infty]$	Y	Y	1	∞	3	
	bounds	U	c	$[c, \infty]$	Y	Y	2	∞	4	
	hold	U	c	$[c, \infty]$	Y		1	∞	5	
	also	U	c	$[\ell, \infty]$ $\ell < c$			$\log(c-\ell+1)+2$	∞	7	
	for									
	consen-	U	c	$[1, \infty]$		Y	$\log c + 1$	∞	6	[SP89]
	sus.)	L_2	c	$[1, \infty]$			$\log c + 1$	2	6	[SP89]
2	DF	U	c	$[1, \infty]$		Y	c	∞	6	[SP89]
	mutex	L	c	$[1, \infty]$			c	2	6	[BL93]
3	Election	L	bounded	$[1, \infty]$			∞	2	6	$1L_2$,2L
	SF	U_1	unbounded	$[1, \infty]$	Y	Y	∞	∞	8	
	mutex	U_2	infinite	$[1, \infty]$			∞	2	9	
4	Test	L	bounded	$[1, \infty]$			∞	2	10	3L,5U
	&	U	unbounded	$[1, \infty]$	Y	Y	∞	∞	10	$3U_1$
	Set	U	infinite	$[1, \infty]$			∞	2	10	$3U_2$
Results using atomic registers in a fault-free model										

Problem			Model				Result					
							Properties		Space		Thm	Using
	Name	Bound	Concurrency $c > 1$	Participation	Fault tolerance	Shared objects	adaptive?	symmetric?	#	size	#	results from
5	Election	L	c	$[\infty, \infty]$	0	T&S			1	2		trivial
	DF mutex	U	infinite	$[1, \infty]$	∞	T&S	Y	Y	1	2		trivial
6	SF	L	bounded	$[1, \infty]$	0	RW,T&S			∞	2	11	[Pet94]
	mutex	U	infinite	$[1, \infty]$	0	RW,T&S			∞	2	9	$3U_2$
7	Naming	L&U	bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	12	
		L	unbounded	$[1, \infty]$	∞	T&S			no alg		12	
8	Consen-	L	2	$[\infty, \infty]$	1	RMW			2	2		trivial
	sus	U	infinite	$[1, \infty]$	∞	RMW	Y	Y	1	3		trivial
		U	infinite	$[\infty, \infty]$	1	RMW	Y		4	2	13	[LA87]
9	SF	L	bounded	$[1, \infty]$	0	RMW			1	∞	14	[F+89]
	mutex	U	unbounded	$[1, \infty]$	0	RMW	Y	Y	1	∞	14	[B+82]
		U	infinite	$[1, \infty]$	0	RMW			∞	2		$3U_2$
10	SF	U	unbounded	$[1, \infty]$	0	RW,wPV	Y	Y	2,2	2	15	[FP87]
	mutex	L	infinite	$[1, \infty]$	0	RW,wPV			no alg		15	[FP87]
Results using stronger shared objects												

2 Atomic registers: participation is required

This section demonstrates that requiring a minimum number of process to participate (the required participation model) is a powerful enabling assumption: the problems we study are solvable with a small number of registers. However, reducing the number of registers does not mean that it is also possible to achieve a finite state space. We begin by showing that any solution to election in this model requires infinite state space. Then we show that election can be solved either by using an infinite number of registers of finite size or a (small) finite number of registers of infinite size.

2.1 A lower bound for election when the concurrency is $c > 1$

Even when participation is required, any solution to election for infinitely many processes must use infinite shared space. This holds even if processes do not need to learn the identity of the leader as part of the election algorithm.

Theorem 1. *There is no solution to election with finite concurrency $c > 1$ and with $[c, \infty]$-participation using finite shared memory (finitely many registers of finite size).*

Proof. Assume to the contrary that there is such an algorithm using finite space. Consider the solo run $solo(p)$ of each process p, and define $write(p)$ to be the subsequence of write events in $solo(p)$. For each process p, let $repeat(p)$ be the first state of the shared memory that repeats infinitely often in $solo(p)$. (Recall that the algorithm uses finite space by assumption. Also, if $write(p)$ is finite, $repeat(p)$ will necessarily be the state after the last write in $write(p)$.)

Let $\beta(p)$ be the finite prefix of $write(p)$ that precedes the first instance of $repeat(p)$. For each such $\beta(p)$, we construct a *signature* subsequence, $sign(p)$, by removing repetitive states and the intervening steps, as follows: Ignoring the states of the processes for now, let $\beta(p) = x_0, x_1, x_2 ...$, where the x_j are the successive states of the shared memory in $\beta(p)$. Suppose x_j is the first state that repeats in $\beta(p)$, and that x_k is the last state in $\beta(p)$ such that $x_j = x_k$. Remove the subsequence $x_{j+1}, ..., x_k$ from $\beta(p)$. The resulting sequence $x_0, ..., x_j, x_{k+1}, ...$ is a subsequence of $\beta(b)$ with strictly fewer repeating states. Repeat this step until no state repeats–the resulting sequence is (the signature) $sign(p)$.

Since there are only finitely many states of the shared memory, there exists a single state s and a single sequence γ such that $s = repeat(p)$ and $\gamma = sign(p)$ for infinitely many processes, p. The solo runs of any subset of $C = p_1, ..., p_c$ of these processes are compatible: by scheduling appropriately, we can construct runs in which each of the processes take the same steps as in their solo runs. By an easy case analysis, in one such run no leader is elected, or there are many runs in which two or more leaders are elected, a contradiction.

The (compatible) run of the processes in C is constructed as follows: Let $\gamma = y_1, ..., y_k$. First run these c process until each one of them is about to execute its first write. Run process p_1 until it takes the last step in $\beta(p_1)$ that leaves the

memory in state y_1. Repeat for p_2 through p_c, then repeat this loop for each state in γ. The resulting finite run leaves the shared memory in state s, and each of the c processes p_i has taken the steps in $\beta(p_i)$. Since the state s repeats infinitely often in the remainder of each of the solo runs $solo(p_i)$, the run can be extended by using a round-robin schedule to extend by some sequence of steps of each process, always returning the memory to state s.

Notice that the signatures are used only in the first stage of the construction, when the processes run until the memory state is s, in such a way that each process "thinks" it ran solo to get there. After reaching (together) the recurring state s, each process can be run solo until s recurs. Since the same state s recurs infinitely many times for each of the processes in C, we can schedule each in turn that has not terminated to take (some finite, nonzero number) of steps until s occurs next, then schedule the next one (i.e., round robin scheduling). Now everyone either runs forever or terminates as in a solo run. □

2.2 Algorithms for election and consensus for concurrency c

Although the previous theorem shows that election for infinitely many processes requires unbounded shared memory, when assuming participation is required, many problems become solvable with a small number of registers. We first study the scenario, in which the concurrency is equal to participation (concurrency c and participation $[c, \infty]$). We show that election can be solved either by (1) using an infinite number of atomic bits, or (2) using a single register of infinite size. We also present two simple symmetric algorithms.

Theorem 2. *For any finite concurrency c, with $[c, \infty]$-participation, there are non-adaptive asymmetric solutions to election (and consensus) using an infinite number of atomic bits.*

Proof. We identify each process with a unique natural number. (But each natural number is not necessarily the identity of a participating process. We always take 0 as a process id not assigned to any process.) The algorithm uses an infinite array $b[0], b[1], ...$ of bits, which are initially 0. The first step of process i is to read $b[0]$. If $b[0]$ value is 1, it knows that a leader has already been elected and terminates. Otherwise, process i sets $b[i]$ to 1. Then, process i scans the array until it notices that c other bits (other than $b[0]$) are set. In order not to miss any bit that is set, it scans the array according to a diagonalization: a schedule that visits each bit an infinite number of times. (A canonical example is to first read $b[1]$; then $b[1], b[2]$; then $b[1], b[2], b[3]$, etc.) Once a process notices that c bits are set, it set $b[0]$ to 1. The process with the smallest id among the c that are set to 1 is the leader. By scanning the bits after reading $b[0]$ as 1, the other processes can also learn the id of the leader. This solution for election can be trivially modified, using one other bit, to solve consensus. □

A symmetric election algorithm is presented in [SP89], for n processes (with concurrency level n), which use only three atomic registers. Below we present three election algorithms: a symmetric algorithm for concurrency 2 using one

register, a symmetric algorithm for concurrency c using two registers, and finally, an asymmetric algorithm for concurrency c using one register. For lack of space, we present only the code of the algorithms, and omit the detailed descriptions.

Theorem 3. *For finite concurrency $c = 2$ with $[2, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using one register.*

Proof. The algorithm is given in Figure 1. This election algorithm can be easily

```
Process i's program
Shared:
        (Leader, Marked): (Process id, boolean) initially (0, 0)
Local:
        local_leader: Process id

1   if Marked = 0 then
2       (Leader, Marked) := (i, 0)
3       await (Leader ≠ i) ∨ (Marked = 1)
4       local_leader := Leader
5       (Leader, Marked) := (local_leader, 1)
6   fi
7   return(Leader)
```

Fig. 1. Symmetric election for concurrency 2

converted into a consensus algorithm by appending the input value (0 or 1) to each process id. The input value of the leader is the consensus value. □

The next algorithm, for concurrency $c \geq 2$, is similar to the previous, in that the last participant to write the $Leader$ field is then $Marked$ as the leader. A second register, $Union$, is used by the first c participants to ensure that they have all finished writing their id into $Leader$. (Note, that in doing so, they learn each other's id's.) Because c is also the number of required participants, each of the first c participants can spin on the $Union$ register until the number of id's in it is c.

Theorem 4. *For any finite concurrency c with $[c, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using two registers.*

Proof. The algorithm is given in Figure 2. As above, this election algorithm can be easily converted into a consensus algorithm. □

Finally, we present an asymmetric solution to election (and consensus) using a single atomic register. Asymmetric algorithms allow comparisons between process id's (we assume a total order), however, since the identities of the participating process are not initially known to all processes, the trivial solution where all processes choose process 1 as a leader is excluded.

Theorem 5. *For any finite concurrency c and $[c, \infty]$-participation there is an adaptive asymmetric solution to election and consensus using one register.*

Shared:
($Leader, Marked$): (Process id, boolean), initially $(0, 0)$
$Union$: set of at most c process ids, initially $\emptyset$
Local:
$local\_leader$: Process id
$local\_union1$: set of at most c process ids
$local\_union2$: set of at most c process ids, initially $\{i\}$

```
1   if Marked = 0 then (Leader, Marked) := (i, 0) fi
2   local_union1 := Union
3   while |local_union1| < c do
4       if ¬(local_union2 ⊆ local_union1) then
5           Union := local_union1 ∪ local_union2
6       fi
7       local_union2 := local_union1 ∪ local_union2
8       local_union1 := Union
9   od
10  local_leader := Leader
11  (Leader, Marked) := (local_leader, 1)
12  return(Leader)
```

Fig. 2. Symmetric election for concurrency c

Proof. The algorithm in Figure 3 is an adaptation of the two register algorithm for the symmetric case (Theorem 4), where only a single register is used. As before, each of the first c processes registers itself in the register and repeatedly updates *Union* with any new information it has obtained (as done in the repeat loop of the symmetric algorithm). The elected process is the process with the minimum id among the first c processes to participate in the algorithm. □

3 Atomic registers: participation is not required

3.1 Consensus, election and mutual exclusion for concurrency c

We next consider the number and size of registers required to solve consensus, election and mutual exclusion in a model in which participation is not required (i.e., $[1, \infty]$-participation).

Theorem 6. *For concurrency level c, the number of atomic registers that are:*
(1) necessary and sufficient for solving deadlock-free mutual exclusion, is c;
(2) necessary and sufficient for solving election, is $\log c + 1$*;*
(3) sufficient for solving consensus, is $\log c + 1$*.*

The proof, which consists of minor modifications of known results, is omitted. Theorem 6 implies that when the concurrency level is not finite, an infinite number of registers are necessary for solving the election and mutual exclusion problems. But are an infinite number of registers sufficient? (We observe, based on a result from [YA94], that if we restrict the number of processes that can try to write the same register at the same time, the answer is no.)

```
Process i's program
Shared: (Union, Sig, Ack): Union a set of at most c process id's,
                0 ≤ Sig ≤ c, Ack ∈ {0,1}, initially (∅, 0, 0)
Local:  local_union1: set of at most c process id's
        local_union2: set of at most c process id's, initially {i}
        myrank, local_sig1, local_sig2: integers between 0 and c, initially 0
        local_ack1: Boolean, initially 0
        leader: process id

1   (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
2   while |local_union1| < c do
3       if ¬(local_union2 ⊆ local_union1) then
4           (Union, Sig, Ack) := (local_union1 ∪ local_union2, 0, 0)
5       fi
6       local_union2 := local_union1 ∪ local_union2
7       (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
8   od
9   local_union2 := local_union1
10  leader := min{q : q ∈ local_union1}
11  if i ∉ local_union1 then
12      return(leader)
13  elseif i ≠ leader then
14      myrank := |{h ∈ local_union1 : h < i}|
15      while local_sig1 < c do
16          if (|local_union1| < c) then (Union, Sig, Ack) := (local_union2, 0, 0)
17          elseif (local_sig1 = myrank) ∧ (local_ack1 = 1) then
18              (Union, Sig, Ack) :=(local_union2, myrank, 0)
19          fi
20          (local_union1, local_sig1, local_ack1) :=(Union, Sig, Ack)
21      od
22      return(leader)
23  else                                        /* |local_union2| = c, i = leader */
24      while local_sig1 < c do
25          if (|local_union1| < c) ∨ (local_sig1 < local_sig2) then
26              (Union, Sig, Ack) := (local_union2, local_sig2, local_ack2)
27          elseif local_ack1 = 0 then                  /* Signal acknowledged */
28              local_sig2 := local_sig2 + 1
29              (Union, Sig, Ack) := (local_union2, local_sig2, 1) /* Send signal. */
30          fi
31          (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
32      od
33      return(i)
34  fi
```

Fig. 3. (Asymmetric) election for concurrency c

3.2 Relating participation and concurrency

The following theorem unifies the results of Theorems 5 (participation is required) and 6 (participation is not required), demonstrating a relation between participation and concurrency in this case.

Theorem 7. *For concurrency c with $[\ell, \infty]$-participation where $l < c$, $\log(c - \ell + 1) + 2$ registers are sufficient for solving election.*

Proof. First, we use the single register algorithm (Theorem 5) as a filter. Here, instead of choosing a single leader, up to $c-\ell+1$ processes may be elected. These processes continue to the next level. To implement this, we slightly modify the single register algorithm. A process, needs to wait only until it notices that ℓ (instead of c) processes are participating, and if it is the biggest among them it continues to the next level. Thus, at most $c - \ell + 1$ (and at least one) processes continue to the next level. In that level, they compete using the algorithm of Theorem 6 (no change is needed in that algorithm), until one of them is elected. This level requires $\log(c - \ell + 1) + 1$ registers. □

3.3 Starvation-free algorithms for unbounded concurrency

Theorem 6 leads naturally to the question of whether an infinite number of registers suffice for solving mutual exclusion with unbounded concurrency, when participation is not required. We present two algorithms that answer this question affirmatively. The first is an adaptive and symmetric algorithm using infinitely many infinite-sized registers. The second is neither adaptive nor symmetric, but uses only (infinitely many) bits.

Theorem 8. *There is an adaptive symmetric solution to election, consensus and starvation-free mutual exclusion for unbounded concurrency using an infinite number of registers.*

Proof. These problems can all be solved by simple adaptations to the deadlock-free mutual exclusion algorithm presented in Figure 4. This algorithm has three interesting properties: it works assuming an unbounded concurrency level, it is adaptive – its time complexity is a function of the actual number of contending processes, and it is symmetric. Except for the non-adaptive algorithm in the next subsection, we know of no mutual exclusion algorithm (using atomic registers) satisfying the first property. In this algorithm, the processes compete in levels, each of which is used to eliminate at least one competing process, until only one process remains. The winner enters its critical section, and in its exit code it publishes the index to the next empty level, so that each process can join the competition starting from that level.

The adaptive deadlock-free algorithm above is easily modified using standard "helping" techniques to satisfy starvation freedom. Because the number of processes is infinite, a global diagonalization is necessary instead of a round-robin schedule: a process helps others in the order given by an enumeration in which

```
Process i's program
Shared:
    next: integer, initially 0
    x[0..∞]: array of integers (the initial values are immaterial)
    b[0..∞], y[0..∞], z[0..∞]: array of boolean, initially all 0
Local:
    level: integer, initially 0
    win: boolean, initially 0

1  start: level := next
2    repeat
3        x[level] := i
4        if y[level]      then  b[level] := 1
5                              await level < next
6                              goto start fi
7        y[level] := 1
8        if x[level] ≠ i then   await (b[level] = 1) ∨ (z[level] = 1)
9                               if z[level] = 1 then await level < next
10                                                   goto start
11                             else level := level + 1 fi
12       else   z[level] := 1
13              if b[level] = 0 then win := 1
14              else level := level + 1 fi fi
15   until win = 1
16   critical section
17   next := level + 1
```

Fig. 4. Adaptive deadlock-free mutual exclusion for unbounded concurrency

every process id appears infinitely often. That is, processes set flags when they leave the remainder section, and before leaving the critical section, a process examines the flag of the next process in the diagonalization and grants the critical section if it determines the associated process is waiting. (Global variables recording progress in this globalization are maintained via the mutual exclusion of the critical section.) Starvation-freedom follows if each process appears infinitely often in the diagonalization. Even simpler, standard modifications convert the above algorithm to solve leader election or consensus. □

The question of designing an adaptive mutual exclusion algorithm, was first raised in [MT93], where a solution was given for a given working system, which is useful provided process creation and deletions are rare (the term *contention sensitive* was suggested but the term *adaptive* become commonly used). In [CS94], the only previously known adaptive mutual exclusion algorithm was presented, in a model where it is assumed that the number of processes (and hence concurrency) is finite. The algorithm exploits this assumption to work in bounded space. The algorithm does not work assuming unbounded concurrency. In [Lam87], a fast algorithm is presented which provides fast access in the absence of contention. However, in the presence of any contention, the winning process may have to check the status of all other n processes (i.e. access n different shared registers) before it is allowed to enter its critical section. A symmetric (non-

adaptive) mutual exclusion algorithm for n processes is presented in [SP89].

Theorem 9. *There is a non-adaptive asymmetric solution to election, consensus and starvation-free mutual exclusion for infinite concurrency using an infinite number of bits.*

```
Process i's program
Shared:
    RaceOwner[1..∞], RaceOther[1..∞], Win[1..∞], Lose[1..∞]: boolean, initially 0
Local:
    index: integer, initially 1

1   RaceOwner[i] := 1
2   if RaceOther[i] = 0 then Win[i] := 1 else Lose[i] := 1 fi
3   repeat forever
4       RaceOther[index] := 1
5       if RaceOwner[Index] = 1 then
6             await (Win[Index] = 1 or Lose[Index] = 1)
7       fi
8       if Win[index] = 1 then return(index) fi
9       index := index + 1
10  end repeat
```

Fig. 5. (Non-adaptive) leader election for infinite concurrency using bits

Proof. Figure 5 presents a simple algorithm for election. Modification to solve consensus is trivial–starvation-free mutual exclusion can be achieved using techniques similar to those in the previous algorithm. □

4 Test&set bits

A test&set bit is an object that may take the value 0 or 1, and is initially set to 1. It supports two operations: (1) a reset operation: write 0, and (2) a test&set operation: atomically assign 1 and return the old value. We first make the following observation:

Theorem 10. *An infinite number of atomic registers are necessary for implementing a test&set bit when concurrency is bounded and sufficient when concurrency is infinite.*

Theorem 11. *For solving starvation-free mutual exclusion, an infinite number of atomic bits and test&set bits are necessary and sufficient when the concurrency level is bounded.*

Proof. In [Pet94], it is proved that n atomic registers and test&set bits are necessary for solving the starvation-free mutual exclusion problem for n processes (with concurrency level n). This implies the necessary condition above. The sufficient condition follows immediately from Theorem 9. □

4.1 Naming

The following theorem demonstrates that in certain cases a problem is solvable assuming bounded concurrency, but is not solvable assuming unbounded concurrency. The *wait-free naming* problem is to assign unique names to initially identical processes. After acquiring a name the process may later release it. A process terminates only after releasing the acquired name. A solution to the problem is required to be *wait-free*, that is, it should guarantee that every participating process will always be able to get a unique name in a finite number of steps regardless of the behavior of other processes. (Proof is omitted.)

Theorem 12. *(1) For bounded concurrency, an infinite number of T&S bits are necessary and sufficient for solving wait-free naming. (2) For unbounded concurrency, there is no solution to wait-free naming, even when using an infinite number of T&S bits.*

5 Stronger Objects

5.1 RMW bits

A read-modify-write object supports a read-modify-write operation, in which it is possible to atomically read a value of a shared register and based on the value read, compute some new value and assign it back to the register. When assuming a fault-free model with required participation, many problems become solvable with small constant space. The proofs of the next two theorems, which consists of minor modifications of known results, are omitted.

Theorem 13. *There is a solution to consensus with required participation, using 4 RMW bits, with infinite concurrency, even assuming at most one process may fail (by crashing).*

We remark that the proof of the theorem above holds only under the assumption that it is known that there exists processes with ids 1 and 2. In the algorithm in [LA87] (which we modify) only one of these processes may be elected.

Theorem 14. *(1) When only a finite number of RMW registers are used for solving starvation-free mutual exclusion with bounded concurrency, one of them must be of unbounded size. (2) One unbounded size RMW register is sufficient for solving first-in, first-out adaptive symmetric mutual exclusion when the concurrency level is unbounded.*

5.2 Semaphores

Given the results so far about starvation-free mutual exclusion, it is natural to ask whether it can be solved with bounded space? The answer, as presented in [FP87], is that using weak semaphores it can be solved with small constant space for unbounded concurrency, but not with infinite concurrency level.

The result below refers to *weak* semaphores, in which a process that executes a V operation will not be the one to complete the next P operation on that semaphore, if another process has been blocked at that semaphore. Instead, one of the blocked processes is allowed to pass the semaphore to a blocked process.

Theorem 15 (Friedberg and Peterson [FP87]). *(1) There is an adaptive symmetric solution to starvation-free mutual exclusion using two atomic bits and two weak semaphores, when the concurrency is unbounded. (2) There is no solution to the starvation-free mutual exclusion problems using any finite number of atomic registers and weak semaphores, when the concurrency is infinite.*

References

[B+82] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, 1982.

[BL93] J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[CS94] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[F+89] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, January 1989.

[FLP85] 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, 1985.

[FP87] S. A. Friedberg and G. L. Peterson. An efficient solution to the mutual exclusion problem using weak semaphores. *Information Processing Letters*, 25(5):343–347, 1987.

[GK98] E. Gafni and E. Koutsoupias. On uniform protocols. `http://www.cs.ucla.edu/~eli/eli.html`, 1998.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[LA87] M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Also published as an AT&T technical memorandum in May 1991.)

[Pet94] G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).

[SP89] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th PODC*, pages 177–191, 1989.

[YA94] J-H. Yang and J.H. Anderson. Time bounds for mutual exclusion and related problems. In *Proc. 26th ACM Symp. on Theory of Computing*, May 1994.

Establishing Business Rules for Inter-Enterprise Electronic Commerce*

Victoria Ungureanu and Naftaly H. Minsky

Rutgers University, New Brunswick, NJ 08903, USA,
{ungurean,minsky}@cs.rutgers.edu

Abstract. Conventional mechanisms for electronic commerce provide strong means for securing transfer of funds, and for ensuring such things as authenticity and non-repudiation. But they generally do not attempt to regulate the activities of the participants in an e-commerce transaction, treating them, implicitly, as *autonomous agents*. This is adequate for most cases of client-to-vendor commerce, but is quite unsatisfactory for *inter-enterprise* electronic commerce. The participants in this kind of e-commerce are not autonomous agents, since their commercial activities are subject to the business rules of their respective enterprises, and to the preexisting agreements and contracts between the enterprises involved. These policies are likely to be independently developed, and may be quite heterogeneous. Yet, they have to *interoperate*, and be brought to bear in regulating each e-commerce transaction. This paper presents a mechanism that allows such interoperation between policies, and thus provides for *inter-enterprise* electronic commerce.

1 Introduction

Commercial activities need to be regulated in order to enhance the confidence of people that partake in them, and in order to ensure compliance with the various rules and regulations that govern these activities. Conventional mechanisms for electronic commerce provide strong means for securing transfer of funds, and for ensuring such things as authenticity and non-repudiation. But they generally do not attempt to regulate the activities of the participants in an e-commerce transaction, treating them, implicitly, as *autonomous agents*.

This is adequate for most cases of client-to-vendor commerce, but is quite unsatisfactory for the potentially more important *inter-enterprise* (also called business-to-business or B2B) electronic commerce[1]. The participants in this kind of e-commerce are not autonomous agents, since their commercial activities are subject to the business rules of their respective enterprises, and to the preexisting agreements and contracts between the enterprises involved. The nature of this situation can be illustrated by the following example.

* Work supported in part by DIMACS under contract STC-91-19999 and NSF grants No. CCR-9626577 and No. CCR-9710575

[1] At present, B2B accounts for over 80% of all e-commerce, amounting to $150 billion in 1999 (cf. [5]). It is estimated that by 2003 this figure could reach $3 trillion.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 179–193, 2000.

Consider a purchase transaction between an agent x_1 of an enterprise E_1 (the client in this case), and an agent x_2 of an enterprise E_2 (the vendor). Such a transaction may be subject to the following set of policies:

1. A policy $\mathcal{P}_1$ that governs the ability of agents of enterprise E_1 to engage in electronic commerce. For example, $\mathcal{P}_1$ may provide some of its agents with budgets, allowing each of them to issue purchase orders only within the budget assigned to it[2].
2. A policy $\mathcal{P}_2$ that governs the response of agents of enterprise E_2 to purchase orders received from outside. For example, $\mathcal{P}_2$ may require that all responses to purchase orders should be monitored—for the sake of internal control, say.
3. A policy $\mathcal{P}_{1,2}$ that governs the interaction between these two enterprise, reflecting some prior contract between them—we will call this an "interaction policy". For example, $\mathcal{P}_{1,2}$ may reflect a *blanket agreement* between these two enterprise, that calls for agents in E_2 to honor purchase orders from agents in E_1, for up to a certain cumulative value—to be called the "blanket" for this pair of enterprises.

Note that policies $\mathcal{P}_1$ and $\mathcal{P}_2$ are formulated separately, without any knowledge of each other, and they are likely to evolve independently. Furthermore, E_1 may have business relations with other enterprises $E_3, ..., E_k$ under a set of different interaction policies $\mathcal{P}_{1,3}, ..., \mathcal{P}_{1,k}$.

The implementation of such policies is problematic due to their *diversity*, their *interconnectivity* and the *large number of participants* involved. We will elaborate now on these factors, and draw conclusions—used as principles on which this work is based.

First, e-commerce participants have little reason to trust each other to observe any given policy—unless there is some enforcement mechanism that compels them all to do so. The currently prevailing method for establishing e-commerc policies is to build an interface that implements a desired policy, and distribute this interface among all who may need to operate under it. Unfortunately, such a "manual" implementation is both unwieldy and unsafe. It is *unwieldy* in that it is time consuming and expensive to carry out, and because the policy being implemented by a given set of interfaces is obscure, being embedded into the code of the interface. A manually implemented policy is *unsafe* because it can be circumvented by any participant in a given commercial transaction, by modifying his interface for the policy. These observations suggest the following principle:

Principle 1 *E-commerce policies should be made* ***explicit****, and be* ***enforced*** *by means of a generic mechanism that can implement a wide range of policies in a uniform manner.*

Second, e-commerce policies are usually enforced by a *central authority* (see for example, NetBill [4], SET [9], EDI [13]) which mediates between interlocutors.

[2] An agent of an enterprise may be a person or a program.

For example, in the case of the $\mathcal{P}_{1,2}$ policy above, one can have the blankets maintained by an authority trusted by both enterprises which will ensure that neither party violates this policy.

However such centralized enforcement mechanism is not scalable. When the number of participants grows, the centralized authority becomes a bottleneck, and a dangerous *single point of failure*. A centralized enforcement mechanism is thus unsuitable for B2B e-commerce because of the huge number of of participants involved—large companies may have as many as *tens of thousand of supplier-enterprises* (cf. [6]). The need for scalability leads to the following principle:

Principle 2 *The enforcement mechanism of e-commerce policies needs to be decentralized.*

Finally, a single B2B transaction is subject to a conjunction of several distinct and heterogeneous policies. The current method for establishing a set of policies is to to *combine* them into a single, global *super-policy*. While an attractive approach for other domains[3], combination of policies is not well suited for inter-enterprise commerce because it does not provide for the *privacy* of the interacting enterprises, nor for the *evolution* of their policies. We will now briefly elaborate on these issues.

The creation of a super-policy requires knowledge of the text of sub-policies. But divulging to a third party the internal business rules of an enterprise is not common practice in todays commerce. Even if companies would agree to expose their policies, it would still be very problematic to construct and maintain the super-policy. This is because, it is reasonable to assume that business rules of a particular enterprise or its contracts with its suppliers—the sub-policies in a B2B scenario—change in time. Each modification at the sub-policy level triggers, in turn, the modification of all super-policies it is part of, thus leading to a maintenance nightmare. We believe, therefore, that it is important for the following principle to be satisfied:

Principle 3 *Inter-operation between e-commerce policies should maintain their* `privacy`*,* `autonomy` *and* `mutual transparency`*.*

We have shown in [10] how fairly sophisticated contracts between autonomous clients and vendors can be formulated using what we call Law-Governed Interaction (LGI). The model was limited however in the sense that policies were viewed as isolated entities. In this paper we will describe how LGI has been extended, in accordance with the principles mentioned above, to support policy inter-operation.

The rest of the paper is organized as follows. We start, in Section 2, with a brief description of the concept of LGI, on which this work is based; in Section 3 we present our concept of policy-interoperability. Details of a secure implementation of interoperability under LGI are provided in Section 4. Section 5 discusses some related work, and we conclude in Section 6.

[3] like for example federation of databases, for which it was originally devised

2 Law-Governed Interaction (LGI)—an Overview

Broadly speaking, LGI is a mode of interaction that allows an heterogeneous group of distributed agents to interact with each other, with confidence that an explicitly specified set of rules of engagement–called the *law* of the group—is strictly observed by each of its member. Here we provide a brief overview of LGI, for more detailed discussion see [10, 11].

The central concept of LGI is that of a policy $\mathcal{P}$, defined as a four-tuple:

$$\langle \mathcal{M}, \mathcal{G}, \mathcal{CS}, \mathcal{L} \rangle$$

where $\mathcal{M}$ is the set of messages regulated by this policy, $\mathcal{G}$ is an open and heterogeneous group of *agents* that exchange messages belonging to $\mathcal{M}$; $\mathcal{CS}$ is a mutable set $\{\mathcal{CS}_x \mid x \in \mathcal{G}\}$ of what we call *control states*, one per member of group $\mathcal{G}$; and $\mathcal{L}$ is an enforced set of "rules of engagement" that regulates the exchange of messages between members of $\mathcal{G}$. We will now give a brief description of the basic components of a policy.

The Law: The law is defined over certain types of events occuring at members of $\mathcal{G}$, mandating the effect that any such event should have—this mandate is called the *ruling* of the law for a given event. The events thus subject to the law of a group under LGI are called *regulated events*—they include (but are not limited to) the sending and arrival of $\mathcal{P}$-messages.

The Group: We refer to members of $\mathcal{G}$ as *agents*, by which we mean autonomous actors that can interact with each other, and with their environment. Such an agent might be an encapsulated software entity, with its own state and thread of control, or it might be a human that interacts with the system via some interface. (Given popular usage of the term "agent", it is important to point out that this term does not imply here either "intelligence" nor mobility, although neither of these is ruled out.) Nothing is assumed here about the structure and behavior of the members of a given $\mathcal{L}$-group, which are viewed simply as sources of messages, and targets for them.

The Control State: The *control-state* $\mathcal{CS}_x$ of a given agent x is the bag of attributes associated with this agent (represented here as Prolog-like terms). These attributes are used to structure the group $\mathcal{G}$, and provide state information about individual agents, allowing the law $\mathcal{L}$ to make distinctions between different members of the group. The control-state $\mathcal{CS}_x$ can be acted on by the primitive operations, which are described below, subject to law $\mathcal{L}$.

Regulated Events: The events that are subject to the law of a policy are called *regulated events*. Each of these events occurs at a certain agent, called the *home* of the event[4] The following are two of these event-types:

[4] strictly speaking, events occur at the controller assigned to the home-agent.

1. `sent(x,m,y)`—occurs when agent `x` sends an $\mathcal{L}$-message `m` addressed to `y`. The sender `x` is considered the *home* of this event.
2. `arrived(x,m,y)`—occurs when an $\mathcal{L}$-message `m` sent by `x` arrives at `y`. The receiver `y` is considered the *home* of this event.

Primitive Operations: The operations that can be included in the ruling of the law for a given regulated event `e`, to be carried out at the home of this event, are called *primitive operations*. Primitive operations currently supported by LGI include operations for testing the control-state of an agent and for its update, operations on messages, and some others. A sample of primitive operations is presented in Figure 1.

Operations on the control-state	
`t@CS`	returns true if term `t` is present in the control state, and fails otherwise
`+t`	adds term `t` to the control state;
`-t`	removes term `t` from the control state;
`t1←t2`	replaces term `t1` with term `t2`;
`incr(t(v),d)`	increments the value of the parameter `v` of term `t` with quantity `d`
`dcr(t(v),d)`	decrements the value of the parameter `v` of term `t` with quantity `d`

Operations on messages	
forward(x,m,y)	sends message `m` from `x` to `y`; triggers at `y` an `arrived(x,m,y)` event
deliver(x,m,y)	delivers to agent `y` message `m` (sent by `x`)

Fig. 1. Some primitive operations

The Law-Enforcement Mechanism: Law $\mathcal{L}_\mathcal{P}$ is enforced by a set of trusted entities called *controllers* that mediate the exchange of $\mathcal{P}$-messages between members of group $\mathcal{G}$. For every active member `x` in $\mathcal{G}$, there is a controller $\mathcal{C}_x$ logically placed between `x` and the communications medium. And all these controller carry the *same law* $\mathcal{L}$. This allows the controller $\mathcal{C}_x$ assigned to `x` to compute the ruling of $\mathcal{L}$ for every event at `x`, and to carry out this ruling locally.

Controllers are *generic*, and can interpret and enforce any well formed law. A controller operates as an independent process, and it may be placed on the same machine as its client, or on some other machine, anywhere in the network. Under Moses (our current implementation of LGI) each controller can serve several agents, operating under possibly different laws.

3 Interoperability Between Policies

In this section we introduce an extension to LGI framework that provides for the interoperability of different and otherwise unrelated policies. This section is

organized as follows: in Section 3.1 we present our concept of interoperability. In Section 3.2 we describe an extension of LGI that supports this concept; the extended LGI is used in Section 3.3 to implement a slightly refined version of the motivating example presented in Section 1. We conclude this Section by showing how privacy, autonomy and transparency of interoperating policies are achieved in this framework.

3.1 A Concept of Interoperability

By "interoperability" we mean here, the ability of an agent $x/\mathcal{P}$ (short for "an agent x operating under policy $\mathcal{P}$") to exchange messages with $y/\mathcal{Q}$, were $\mathcal{P}$ and $\mathcal{Q}$ are different policies[5], such that the following properties are satisfied:

consensus: An exchange between a pair of policies is possible only if it is authorized by both.
autonomy: The effect that an exchange initiated by $x/\mathcal{P}$ may have on the structure and behavior of $y/\mathcal{Q}$, is subject to policy $\mathcal{Q}$ alone.
transparency: Interoperating parties need not to be aware of the details of each other policy.

To provide for such an inter-policy exchange we introduce into LGI a new primitive operation—`export`—and a new event—`imported`, as follows:

- Operation `export(x/`$\mathcal{P}$`,m,y/`$\mathcal{Q}$`)`, invoked by agent x under policy $\mathcal{P}$, initiates an exchange between x and agent y operating under policy $\mathcal{Q}$. When the message carrying this exchange arrive at y it would invoke at it an `imported` event under $\mathcal{Q}$.
- Event `imported(x/`$\mathcal{P}$`,m,y/`$\mathcal{Q}$`)` occurs when a message `m` exported by $x/\mathcal{P}$ arrives at $y/\mathcal{Q}$.

We will return to the above properties in Section 3.4 and show how they are brought to bear under LGI.

3.2 Support for Interoperability under LGI

A policy $\mathcal{P}$ is maintained by a server that provides persistent storage for the law $\mathcal{L}$ of this policy, and the control-states of its members. This server is called the *secretary* of $\mathcal{P}$, to be denoted by $\mathcal{S}_\mathcal{P}$. In the basic LGI mechanism, the secretary serves as a name server for policy members. In the extended model it acts also as a name server for the policies which inter-operate with $\mathcal{P}$. In order to do so $\mathcal{S}_\mathcal{P}$ maintains a list of policies to which members of $\mathcal{P}$ are allowed to export to, and respectively import from (subject of course to $\mathcal{L}_\mathcal{P}$). For each such policy $\mathcal{P}'$, $\mathcal{S}_\mathcal{P}$ records among other information the address of $\mathcal{S}_{\mathcal{P}'}$.

For an agent `x` to be able to exchange $\mathcal{P}$-messages under a policy $\mathcal{P}$, it needs to engage in a connection protocol with the secretary. The purpose of the protocol

[5] It is interesting to note that x and y may actually be the same agent.

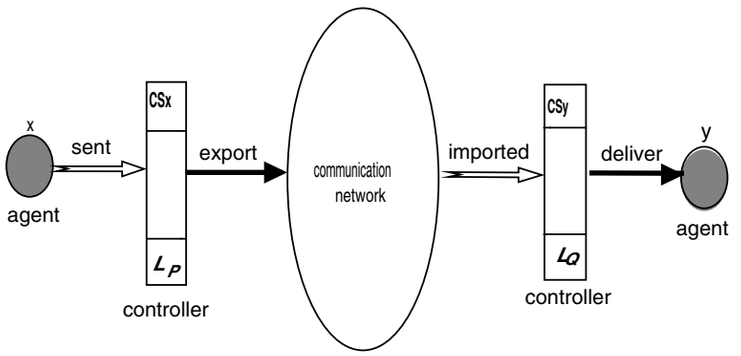


Fig. 2. Policy interoperation

is to assign `x` to a controller $\mathcal{C}_x$ which is fed the law of $\mathcal{P}$ and the control state of `x` (for a detailed presentation of this protocol the reader is referred to [11]).

To see how an `export` operation is carried out, consider an agent `x` operating under policy $\mathcal{P}$, which sends a message `m` to agent `y` operating under policy $\mathcal{Q}$ assuming that `x` and `y` have joined the policy $\mathcal{P}$, respectively $\mathcal{Q}$ (Figure 2). Message `m` is sent by means of a routine provided by the Moses toolkit, which forwards it to $\mathcal{C}_x$—the controller assigned to `x`. When this message arrives at $\mathcal{C}_x$, it generates a `sent(x,m,y)` event at it. $\mathcal{C}_x$ then evaluates the ruling of law $\mathcal{L}_\mathcal{P}$ for this event, taking into account the control-state $\mathcal{CS}_x$ that it maintains, and carries out this ruling.

If this ruling calls the control-state $\mathcal{CS}_x$ to be updated, such update is carried out directly by $\mathcal{C}_x$. And if the ruling for the `sent` event calls for the `export` of `m` to `y`, this is carried out as follows. If C_x does not have the address of C_y, the controller assigned to `y`, it will ask $\mathcal{S}_\mathcal{P}$ for it. When the secretary responds, $\mathcal{C}_x$ will finalize the `export` and will cache the address. As such, forthcoming communication between `x` and `y` will not require the extra step of contacting $\mathcal{S}_\mathcal{P}$.

When the message `m` sent by $\mathcal{C}_x$ arrives at $\mathcal{C}_y$ it generates an `imported(x,m,y)` event. Controller $\mathcal{C}_y$ computes and carries out the ruling of law $\mathcal{L}_\mathcal{Q}$ for this event. This ruling might, for example, call for the control-state $\mathcal{CS}_y$ maintained by $\mathcal{C}_y$ to be modified. The ruling might also call for `m` to be delivered to `y`, thus completing the passage of message `m`.

In general, all regulated events that occur nominally at an agent `x` actually occur at its controller $\mathcal{C}_x$. The events pertaining to `x` are handled *sequentially* in chronological order of their occurrence. The controller evaluates the ruling of the law for each event, and carries out this ruling *atomically*, so that the sequence of operations that constitute the ruling for one event do not interleave with those of any other event occuring at `x`. Note that a controller might be associated with several agents, in which case events pertaining to different agents are evaluated concurrently.

It should be pointed out that the confidence one has in the correct enforcement of the law at every agent depends on the assurance that all messages are mediated by correctly implemented controllers. The way to gain such an assurance is a security issue we will address in Section 4.

3.3 A Case Study

We now show how a slightly refined version of the three policies $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_{1,2}$, introduced in Section 1, can be formalized, and thus enforced, under LGI. We note that $\mathcal{P}_1$ and $\mathcal{P}_2$ do not depend on each other in any way. Each of these policies provides for export to, and import from, the interaction policy $\mathcal{P}_{1,2}$, but they have no dependency on the internal structure of $\mathcal{P}_{1,2}$.

After the presentation of these three policies we will we illustrate the manner in which they interoperate by describing the progression of a single purchase transaction. We conclude this section with a brief discussion.

Policy $\mathcal{P}_1$ Informally, this policy, which governs the ability of agents of enterprise E_1 to issue purchase orders, can be stated as follows:

> For an agent in E_1 to issue a purchase order (PO) it must have a budget assigned to it, with a balance exceeding the price in the PO. Once a PO is issued, the agent's budget is reduced accordingly. If the PO is not honored, for whatever reason, then the client's budget is restored.

Formally, under LGI, the components of $\mathcal{P}_1$ are as follows: the group $\mathcal{G}$ consists of the employees allowed to make purchases. The set $\mathcal{M}$ consists of the following set of messages:

- `purchaseOrder(specs,price,c)`, which denotes a purchase order for a merchandise described by `specs` and for which the client `c` is willing to pay amount `price`.
- `supplyOrder(specs,ticket)`, which represents a positive response to the PO, where `ticket` represents the requested merchandise. (We assume here that the merchandise is in digital form, e.g. an airplane ticket, or some kind of certificate. If this is not the case, then the merchandise delivery cannot be formalized under LGI.)
- `declineOrder(specs,price,reason)` denoting that the order is declined and containing a `reason` for the decline.

The control-state of each member in this policy contains a term `budget(val)`, where `val` is the value of the budget. Finally, the law of this policy is presented in Figure 3. This law consists of three rules. Each rule is followed by an explanatory comment (in italics). Note that under this law, members of $\mathcal{P}_1$ are allowed to interoperate only with members of policy $\mathcal{P}_{1,2}$.

$\mathcal{I}$*nitially*: A member has in his control state an attribute budget(val), where `val` represents the total dollar amount it can spend for purchases.

$\mathcal{R}1$.
```
sent(X1, purchaseOrder(Specs,Price,X1),X2) :-
        budget(Val)@CS, Val>Price,
        do(dcr(budget(Val),Price)),
        do(export(X1/p1,purchaseOrder(Specs,Price,X1),X2/p12)).
```
A `purchaseOrder` *message is exported to the vendor* `X2` *that operates under the inter-enterprise policy* `p12`*—but only if* `Price`*, the amount* `X1` *is willing to pay for the merchandise, is less than* `Val`*, the value of the sender's budget.*

$\mathcal{R}2$.
```
imported(X2/p12,supplyOrder(Specs,Ticket),X1/p1) :-
        do(deliver(X2,supplyOrder(Specs,Ticket),X1)).
```
A `supplyOrder` *message, imported from* `p12`*, is delivered without further ado.*

$\mathcal{R}3$.
```
imported(X2/p12,declineOrder(Specs,Price,Reason),X1/p1) :-
        do(incr(budget(Val),Price)),
        do(deliver(X2,declineOrder(Specs,Price,Reason),X1)).
```
A `declineOrder` *message, imported from* `p12`*, is delivered after the budget is restored by incrementing it with the price of the failed PO.*

Fig. 3. The law of policy $\mathcal{P}_1$

Policy $\mathcal{P}_2$ Informally, this policy which governs the response of agents of E_2 to purchase orders, can be stated simply as follows:

> Each phase of a purchase transaction is to be monitored by a designated agent called `auditor`.

The components of $\mathcal{P}_2$ are as follows: the group $\mathcal{G}$ of this policy consists of the set of employees of E_2 allowed to serve purchase orders, and of a designated agent `auditor` that maintains the audit trail of their activities. For simplicity, we assume here that the set of messages recognized by this policy is the same as for policy $\mathcal{P}_1$—this is not necessary, as will be explained later. The law $\mathcal{L}_2$ of this policy is displayed in Figure 4. Note that unlike $\mathcal{L}_1$, which allows for interoperability only with policy $\mathcal{P}_{1,2}$, this law allows for interoperability with arbitrary policies. (The significance of this will be discussed later.)

Policy $\mathcal{P}_{1,2}$ We assume that there is a *blanket agreement* $\mathcal{P}_{1,2}$ between enterprises E_1 and E_2, stated, informally, as follows:

> A purchase order is processed by the vendor only if the amount offered by the client does not exceed the remaining balance in the blanket.

The components, under LGI, of this policy are as follows: the group $\mathcal{G}$ consists of the set of agents from the vendor-enterprise E_2 that may serve purchase orders, and a distinguished agent called `blanket` that maintains the balance for

$\mathcal{R}1$. `imported(I/IP,purchaseOrder(Specs,Price,X1),X2/p2) :-`
`do(+order(Specs,I,IP)),`
`do(deliver(X2, purchaseOrder(Specs,Price,X1),auditor)),`
`do(deliver(X1,purchaseOrder(Specs,Price),X2)).`

When a `purchaseOrder` *is imported by the vendor* `X2`*, the message is delivered to the intended destination and also to the designated* `auditor`*.*

$\mathcal{R}2$. `sent(X2,supplyOrder(Specs,Ticket),X1) :-`
`order(Specs,I,IP)@CS,do(-order(Specs,I,IP)),`
`do(export(X2/p2,supplyOrder(Specs,Ticket),I/IP)),`
`do(deliver(X2,supplyOrder(Specs,Ticket),auditor)).`

A message sent by the vendor is delivered to the `auditor`*. The message is exported to* `I`*, the interactant from which this order originally came, under interaction policy* `IP`*. (In our case,* `I` *is the object* `blanket` *operating under* $\mathcal{P}_{1,2}$*, but this does not have to be the case, as explained later).*

Fig. 4. The law of policy $\mathcal{P}_2$

the purchases of the client-enterprise E_1. The law $\mathcal{L}_{1,2}$ of this policy is displayed in Figure 5.

The Progression of a Purchase Transaction We explain now how these policies function together, by means of a step-by-step description of the progression of a purchase transaction initiated by a PO `purchaseOrder(specs,price,x1)` sent by agent x_1 of an enterprise E_1 (the client) to an agent x_2 of E_2 (the vendor).

1. The sending by x_1 of a PO to x_2 is handled by policy $\mathcal{P}_1$ (see Rule $\mathcal{R}1$ of $\mathcal{P}_1$) as follows. If the budget of x_1 is smaller than the specified price, then this PO is simply ignored; otherwise the following operations are carried out: (a) the budget of x_1 is decremented by the specified price; and (b) the PO is *exported* to $x_2/\mathcal{P}_{1,2}$, i.e., to agent x_2 under policy $\mathcal{P}_{1,2}$.
2. The import of a PO into $x_2/\mathcal{P}_{1,2}$ forces the PO to be immediately forwarded to an agent called *blanket*. Agent *blanket*, which operates under $\mathcal{P}_{1,2}$, has in its control-state the term *balance*(*val*), where *val* represents the remaining balance under the blanket agreement between the two enterprises (Rule $\mathcal{R}1$ of $\mathcal{P}_{1,2}$).
3. The arrival of a PO at *blanket* agent causes the balance of the blanket to be compared with the price of the PO. If the balance is bigger it is decremented by the price, and the PO is exported to the vendor agent $x_2/\mathcal{P}_2$ (Rule $\mathcal{R}2$ of $\mathcal{P}_{1,2}$); otherwise, a `declineOrder` message is exported back to the client $x_1/\mathcal{P}_1$ (Rule $\mathcal{R}3$ of $\mathcal{P}_{1,2}$). We will assume for now that the former happen; we will see later what happens when a `declineOrder` message arrives at a client.
4. When a PO exported by agent *blanket* (signifying consistency with the blanket agreement) is imported into $x_2/\mathcal{P}_2$, it is immediately delivered to two agents: (a) to the vendor agent x_2 himself, for its disposition; and (b) to to

Initially: Agent `blanket` has in its control state a term of the form `balance(val)`, where `val` denotes the remaining amount of money that the client-enterprise E_1 has available for purchases, at a given moment in time.

$\mathcal{R}1$.
```
imported(X1/p1,purchaseOrder(Specs,Price),X2/p12) :-
        do(forward(X2,purchaseOrder(Specs,Price,X1), blanket)).
```
A `purchaseOrder` *message imported by a vendor* `X2` *is forwarded to* `blanket` *for approval.*

$\mathcal{R}2$.
```
arrived(X2,purchaseOrder(Specs,Price,X1),blanket) :-
        balance(Val)@CS, Val>=Price,
        do(dcr(balance(Val),Price)),
        do(+ order(Specs,Price,X1,X2)),
        do(export(blanket/p12, purchaseOrder(Specs,Price,X1),X2/p2)).
```
If `Price`*, the sum* `X1` *is willing to pay for the merchandise, is less than* `Val` *the value of the balance, then the* `purchaseOrder` *message is exported to* `X2`*, the vendor which originally received the request under policy* `p2`*.*

$\mathcal{R}3$.
```
arrived(X2,purchaseOrder(Specs,Price,X1),blanket) :-
        balance(Val)@CS, Val<Price,
        do(export(X2/p12, declineOrder(Specs,Price,''insufficient
        funds''),X1/p1)).
```
If the balance is less than the `Price` *then a* `declineOrder` *message is exported to* `X1`*, the client which originally issued the* `purchaseOrder`*.*

$\mathcal{R}4$.
```
imported(X2/p2,supplyOrder(Specs,Ticket),blanket/p12) :-
        order(Specs,Price,X1,X2)@CS, do(-order(Specs,Price,X1,X2)),
        do(export(X2/p12, supply(Specs,Ticket ),X1/p1)).
```
A `supplyOrder` *message is exported to the client* `X1` *which issued the order.*

Fig. 5. The law of policy $\mathcal{P}_{1,2}$

the distinguished agent *auditor*, designated to maintain the audit trail of responses of vendor-agents to purchase orders (Rule $\mathcal{R}1$ of $\mathcal{P}_2$).

5. According to policy $\mathcal{P}_2$, agent x_2 that received a PO can respond by a `supplyOrder` message[6] which triggers two operations: (a) the message is exported to *blanket*/$\mathcal{P}_{1,2}$, and (b) a copy of this message is delivered to the *auditor* object (Rule $\mathcal{R}2$ of $\mathcal{P}_2$).
6. An import of the `supplyOrder` response of $x_2/\mathcal{P}_2$ into *blanket*/$\mathcal{P}_{1,2}$ is automatically exported to the client $x_1/\mathcal{P}_1$ (Rule $\mathcal{R}4$ of $\mathcal{P}_{1,2}$).
7. Finally, the import of a `supplyOrder` message into $x_1/\mathcal{P}_1$ causes this message to be delivered to x_1 (Rule $\mathcal{R}2$ of $\mathcal{P}_1$) , while the import of a `declineOrder`

[6] To keep the example simple we did not describe here the case when the vendor decline the PO.

message into $x_1/\mathcal{P}_1$ causes the budget of x_1 to be restored, before the message is delivered to it (Rule $\mathcal{R}3$ of $\mathcal{P}_1$).

Discussion This case study makes the following simplifying assumptions: (1) all three policies use the same set of messages, and (2) the client-enterprise policy $\mathcal{P}_1$ allows for interoperation only with $\mathcal{P}_{1,2}$. These assumptions are not intrinsic to the proposed model and were adopted only in order to make the example as simple as possible. We will explain now the drawbacks of these assumption and show how they can be relaxed, making this case study far more general and flexible.

First, it is unreasonable to assume that completely different enterprises will use the same *vocabulary* with the same *semantic*. While it is required by the model that interoperating policies—in our example $\mathcal{P}_1$ and $\mathcal{P}_{1,2}$ on one side, and $\mathcal{P}_{1,2}$ and $\mathcal{P}_2$ on the other—"understand" each other messages, policies $\mathcal{P}_1$ and $\mathcal{P}_2$ could have used entirely different messages. The translation from $\mathcal{M}_{\mathcal{P}_1}$ to $\mathcal{M}_{\mathcal{P}_2}$ can generally be done by the intermediate policy $\mathcal{P}_{1,2}$.

Second, it is unrealistic to assume that an enterprise will purchase merchandise only from a single vendor E_2, as is required by our current $\mathcal{P}_1$—which is coded to interoperate only with $\mathcal{P}_{1,2}$, representing the contract between E_1 and E_2. In general, one should provide for an agent in E_1 to purchase from other vendors—say from $E_{2'}$, through an inter-enterprise policy $\mathcal{P}_{1,2'}$ reflecting a pre-agreement between E_1 and $E_{2'}$. An analogous flexibility is inherent in $\mathcal{P}_2$, which does not pose any restrictions on the policy it interoperates with, and thus allows for establishing contracts with *different* clients. A similar technique can be used in $\mathcal{P}_1$ to allow purchasing from any number of vendors.

3.4 Assurances

We are in position now to explain how the three properties of our concept of interoperability, namely `consensus`, `autonomy` and `transparency` are satisfied by LGI mechanism.

The consensus condition stipulated that interoperation between a pair of policies should be agreed by both. This property is satisfied by our implementation because for an agent under $\mathcal{P}$ to send a message to an agent under a different policy $\mathcal{Q}$, $\mathcal{P}$ must have a rule that invokes an *export* operation to $\mathcal{Q}$, and $\mathcal{Q}$ must have a rule that responds to an *imported* event from $\mathcal{P}$.

The *autonomy* condition is satisfied, because the effect on $y/\mathcal{Q}$ of a message imported from elsewhere is determined *only* by the *imported*-rules in $\mathcal{Q}$. Finally, *transparency* is satisfied because, when an agent $y/\mathcal{Q}$ handles a message exported from $x/\mathcal{P}$, it has access only to the message itself and to its source, but not to the policy $\mathcal{P}$ under which it has been produced.

4 A Secure Implementation of Interoperability

To prevent malicious violations of the law, the following conditions have to be met: (1) messages are sent and received only via correctly implemented con-

trollers, and (2) messages are securely transmitted over the network. The first of these conditions can be be handled at different levels of security. First, controllers may be placed on trusted machines. Second, controllers may be trusted when built into *physically secure coprocessors* [12].

To ensure condition (2) above we devised and implemented in Moses toolkit the controller–controller authentication protocol displayed in Figure 6. The purpose of the protocol is twofold. First, it has to ensure that messages are securely transmitted over the network—which is a problem traditionally solved by authentication protocols. The second, more challenging goal is to authenticate communicating controllers as genuine controllers operating under inter operating policies.

This protocol assumes that any controller C has a pair of keys $(\mathbf{K_C}, \mathbf{K_C^{-1}})$, where $\mathbf{K_C}$ is the public key and is assumed to be known by the trusted authority, $\mathbf{T}$[7] and $\mathbf{K_C^{-1}}$ is the private key, and therefore known only by itself. Also the protocol assumes that if C is assigned a member in a policy $\mathcal{P}$, then C maintains a list of the policies which inter-operates with $\mathcal{P}$. For every such policy $\mathcal{P}'$ in this list, C records its identifier $\mathbf{id}(\mathcal{P}')$, the hash of the law $\mathbf{H}(\mathcal{L}_{\mathcal{P}'})$, and the address of the secretary of $\mathcal{P}'$. In the current implementation, this information is given to C by the secretary of $\mathcal{P}$ at the time a member in $\mathcal{P}$ is assigned to C.

(**1**) $C_x \rightarrow C_y$:	$\mathbf{x}, \mathbf{m}, \mathbf{y}, \mathbf{i}$, $\mathbf{id}(\mathcal{P})$,$\mathbf{H}(\mathcal{L}_{\mathcal{P}})$, $\{\mathbf{controller}, \mathbf{K_{C_x}}\}_{\mathbf{K_T^{-1}}}$, $\mathbf{S}_{\mathbf{K_{C_x}^{-1}}}(\mathbf{x}, \mathbf{m}, \mathbf{y}, \mathbf{i}, \mathbf{H}(\mathcal{L}_{\mathcal{P}}), \mathbf{H}(\mathcal{L}_{\mathcal{Q}}))$
(**2**) $C_y \rightarrow C_x$:	$\{\mathbf{controller}, \mathbf{K_{C_y}}\}_{\mathbf{K_T^{-1}}}$ $\mathbf{S}_{\mathbf{K_{C_y}^{-1}}}(\mathbf{i}, \mathbf{H}(\mathcal{L}_{\mathcal{Q}}))$

Fig. 6. Controller–controller authentication protocol

The protocol describes the necessary steps that have to be taken when a controller $\mathbf{C_x}$ sends a message $\mathbf{m}$, on behalf of a member $\mathbf{x}$ operating in policy $\mathcal{P}$, to another controller $\mathbf{C_y}$ assigned to a member $\mathbf{y}$ in policy $\mathcal{Q}$. In the first step of the protocol, C_x sends to C_y a message consisting of `x`, `m`, `y`, and an index number `i`. The index `i` is used to prevent replay attacks and it is maintained by both C_x and C_y. In order to identify to C_y the policy $\mathcal{P}$ to which $\mathbf{x}$ belongs, C_x also transmits $id(\mathcal{P})$ the (unique) identifier of $\mathcal{P}$ and the hash of $\mathcal{L}_{\mathcal{P}}$. To authenticate itself to C_y as a genuine controller, C_x sends to C_y its public certificate along with the signature of a a message consisting of `x`, `m`, `y`, `i`, and the hashes of sender and destination laws.

Now, when controller C_y receives the message it first checks whether `y` is allowed to import messages sent by members in $\mathcal{P}$ policy-group. If this is the case,

[7] For simplicity we assume here a unique certifying authority; the protocol could be easily extended to support a hierarchy of such authorities.

C_y recovers $\mathtt{K}_{C_x}$, the public key of C_x, from the certificate and verifies the signature. If the signature is correct[8] C_y is convinced that it is communicating with a genuine controller, because C_x proved it knows $\mathtt{K}_{C_x}^{-1}$ which is authenticated by the certifying authority. The signature also proves that the message was sent under $\mathcal{L}_\mathcal{P}$ and knowledge of $\mathcal{L}_\mathcal{Q}$. If all conditions are met then an `import(x/`$\mathcal{P}$`,m,y/`$\mathcal{Q}$`)` event will be triggered at $\mathcal{C}_y$.

In the second step of the protocol, C_y acknowledges receiving the message by sending to C_x the signature of the index number `i`, and the hash of the law $\mathcal{L}_\mathcal{P}$, together with its own certificate. After C_x verifies the signature, it is assured that message `m` arrived correctly. Moreover, it trusts that it is talking with a genuine controller because C_y proved to know key $\mathtt{K}_{C_y}^{-1}$. By comparing the hash of the law received with its own C_x can decide whether C_y operates under the law it is expected to.

5 Related Work

The fact that participants in an electronic transaction have different policies, and the importance of finding a common ground between them has been recognized by several researchers. Ketchpel and Garcia-Molina [8] studied the transactions that occur between a customer who buys items from different vendors through brokers. The integrity of such transactions is ensured by trusted agents placed between every two principals. Their role is to generate a transaction protocol which satisfies the policies of the two principals. The protocol is automatically generated using a technique called *graph sequencing*. This is an effective technique, but is limited to individual client-vendor situation, in the sense that a particular client (or vendor) is not bound by an enterprise policy.

Abiteboul, Vianu, Fordham and Yesha [1] propose that the transactions between a client and a vendor be mediated by relational transducers. Generally, such a transducer implements the vendor policy, but their mechanism allows for the modification of the policy. This suggests, that in principle it should be possible that a client may add its own policy. However, such a composition of policies is computationally expensive to enforce—it is undecidable in the general case.

Composition of policies in the context of access control has been studied by several authors: Gong and Qian [7] achieve *policy interoperation* by inferring a composed policy based on (compatible) sub-policies. Another approach, which allows for inter-operation of not necessarily compatible policies is *policy combination* [3]. Finally, we are mentioning the *hierarchical composition* of policies presented in [2]. These approaches rely on the assumption that there is a higher authority which is aware of all sub-policies. Such solutions are not applicable to B2B commerce since there is currently no such authority.

[8] For simplicity, we don't discuss here the case $\mathcal{P}$ is not authorized to export messages to $\mathcal{Q}$ policy-group, or the signature is incorrect. Suffices to say that if this is the case C_y will notify C_x, which in turn will notify `x`.

6 Conclusion

This paper addressed the issue of inter-enterprise electronic commerce, which may be subject to a combination of several heterogeneous policies formulated independently by different authorities. Starting from a mechanism, such as LGI, that supports a formal and enforced concept of a policy, we have argued that such an inter-enterprise commerce requires distinct policies to be able to interoperate, while maintaining mutual transparency, and without loosing their autonomy. We have shown how such a concept of policy-interoperation is implemented in LGI, in a secure and scalable manner, and we have demonstrated the application of this facility for inter-enterprise electronic commerce.

References

1. S. Abiteboul, V. Vianu, B. Forham, and Y. Yesha. Relational transducers for electronic commerce. In *Symposium on Principles of Database Systems*, pages 179–187, June 1998.
2. E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo. An authorization model and its formal semantics. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 127–143, September 1998.
3. C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 51–66, September 1998.
4. B. Cox, J. D. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce*, July 1995.
5. The Economist. E-commerce (a survey). pages 6–54. (February 26th 2000 issue).
6. The Economist. Riding the storm. pages 63–64. (November 6th 1999 issue).
7. L. Gong and X. Qian. Computational issues in secure interoperation. *IEEE Transctions on Software Engineering*, pages 43–52, January 1996.
8. S. Ketchpel and H. Garcia-Molina. Making trust explicit in distributed commerce transactions. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 270–281, 1996.
9. SETCo LLC. Set Secure Electronic Transaction protocol. website: `http://www.ibm.com/security/html/prod_setp.html` .
10. N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, May 1998.
11. N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, July 2000.
12. S.W. Smith and S. H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, April 1999.
13. P.K. Sokol. *From EDI to Electronic Commerce—A Bussines Initiative*. Mc Graw-Hill, 1995.

Metering Schemes with Pricing

Carlo Blundo, Annalisa De Bonis, and Barbara Masucci

Dipartimento di Informatica ed Applicazioni Università di Salerno,
84081 Baronissi (SA), Italy, {debonis,ads}@dia.unisa.it,
http://www.dia.unisa.it/~{carblu, masucci}

Abstract. Metering schemes are cryptographic protocols to count the number of visits received by web sites. These measurement systems are used to decide the amount of money to be paid to web sites hosting advertisements. Indeed, the amount of money paid by the publicity agencies to the web sites depends on the number of clients which visited the sites. In this paper we consider a generalization of the metering scheme proposed by Naor and Pinkas [5]. In their scheme a web site is paid if and only if it has been visited by at least a certain number, say h, of clients. In our scheme there are two thresholds, say ℓ and h, with $\ell < h$. If a web site is visited by at most ℓ clients then the web site receives no money. If it receives at least h visits then it receives a full payment. Finally, if it receives a number f of visits comprised between $\ell + 1$ and $h - 1$ then it receives a partial payment which depends on f. We provide lower bounds on the size of the information distributed to clients and to servers by metering schemes and present a scheme which achieves these lower bounds.

1 Introduction

Advertisement payments are one of the major source of revenue for the web sites. The amount of money charged to display ads depends on the number of visits received by the web site. Web advertisers measure the exposure of their ads by obtaining usage statistics about web sites which host their ads. Consequently, advertisers should prevent the web sites from inflating the count of their visits in order to demand more money. For that reason, there should be an audit agency which provides valid and accurate usage measurements of the servers (web sites). To this aim, the audit agency should dispose of a system to measure the interaction between servers and clients which is secure against fraud attempts by the servers and by the clients which visit the web sites. The cryptographic protocol which provides such a system is called *metering scheme*.

Franklin and Malkhi [4] were the first to consider the metering problem in a rigourous theoretical approach. Their solutions offer only a "lightweight security" and cannot be applied if servers and clients have a strong commercial interest to falsify the metering results.

Subsequently, Naor and Pinkas [5] have proposed secure metering schemes as a mean to prevent web servers from inflating the count of their visits. They

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 194–208, 2000.

contemplated a scenario in which there are coalitions of corrupt servers and clients which cooperate in order to inflate the count of visits received by corrupt servers. Moreover, the schemes proposed by Naor and Pinkas [5] protect servers from clients which attempt to disrupt the metering process. In particular, they have considered metering schemes where a server is able to compute its proof[1] for a certain time frame if and only if it has been visited in that time frame by a number of clients larger than or equal to some threshold h. The metering schemes proposed in [5] are efficient and provide a short proof for the metered data. In their schemes a server which has received a number of visits less than h is in the same situation as a server which has received no visit. Consequently, the audit agency will pay nothing to a server which has been visited by less than h clients. The metering scheme in [5] is supposed to be operating for at most τ time frames and during these time frames is *secure*. A metering scheme is considered secure at a certain time frame t if any server which is visited by less than h clients at that time frame has no information about its proof.

In order to have a more flexible payment system which enables to count the exact number of visits that a server has received in any time frame, we introduce *metering schemes with pricing*. In these schemes there are two thresholds ℓ and h, where $\ell < h \leq n$, and any server can be in three different situations in a given time frame t: 1) the server is visited by a number of clients greater than or equal to h; 2) the server is visited by a number of clients smaller than or equal to ℓ; 3) the server is visited by a number of clients comprised between $\ell + 1$ and $h - 1$. The audit agency would pay all the negotiated amount for the exposure of the ads in case 1); it would pay nothing in case 2); and it would pay a smaller sum in case 3). For any server and for any time frame there is a proof associated to any number of client visits comprised between $\ell + 1$ and h. Hence, the audit agency could pay a certain sum, growing with the number of the visits, in case 3).

Metering schemes involve distributing information to clients and servers. In the model we consider the clients receive a certain amount of information from the audit agency and use this information to compute the information passed to the servers when visiting them. Obviously, such information distribution affects the overall communication complexity. A major goal is to construct metering schemes whose overhead to the overall communication is as small as possible. With this motivations, we decided mainly to focus on the size of the information received by clients and servers in metering schemes, as well as on the size of the proof each server computes and sends to the audit agency. In this paper we provide lower bounds on the size of the information distributed to parties and we present a scheme achieving these lower bounds.

[1] In metering schemes, a *proof* is a value that the server can compute at the end of each time frame if and only if it has been visited by a fixed number of clients. Such a value, at the end of each time frame, is sent to the audit agency.

2 The Model

In this section we define metering schemes with pricing in terms of entropy. We use the entropy approach mainly because this leads to a compact and simple description of the schemes and because the entropy approach takes into account all probability distributions on the sets of the proofs generated by servers. For the reader's convenience, the notations introduced in this section are summarized in Appendix B.

In this paper with a boldface capital letter, say $\mathbf{X}$, we denote a random variable taking value on a set, denoted with the corresponding capital letter X, according to some probability distribution $\{Pr_{\mathbf{X}}(x)\}_{x \in X}$. The values such a random variable can take are denoted with the corresponding lower letter. Given a random variable $\mathbf{X}$ we denote with $H(\mathbf{X})$ the Shannon entropy of $\{Pr_{\mathbf{X}}(x)\}_{x \in X}$ (for some basic properties of entropy, consult Appendix A). Let d be an arbitrary positive integer and let $\mathbf{X}_1, \ldots, \mathbf{X}_d$ be d random variables taking values on the sets $X_1, \ldots, X_d$, respectively. For any subset $V = \{i_1, \ldots, i_v\} \subseteq \{1, \ldots, d\}$, with $i_1 \leq \ldots \leq i_v$, we denote with X_V the set $X_{i_1} \times \ldots \times X_{i_v}$ and with $\mathbf{X}_V$ the sequence of random variables $\mathbf{X}_{i_1}, \ldots, \mathbf{X}i_v$.

A *metering system* consists of n clients, say $\mathcal{C}_1, \ldots, \mathcal{C}_n$, m servers, say $\mathcal{S}_1, \ldots, \mathcal{S}_m$, and an audit agency $\mathcal{A}$ whose task is to measure the interaction between the clients and the servers in order to count the number of client visits that any server receives. Servers which have been visited by at least h clients receive a full payment of the negotiated amount of money, whereas those which have received less than ℓ visits receive no money at all. The servers which have been visited by a number of clients comprised between $\ell + 1$ and $h - 1$ receive a partial payment of the negotiated amount of money. Such partial payment grows with the number of clients which have been served. To this aim, a server which has been visited by a number f of clients comprised between $\ell + 1$ and h should be able to provide the audit agency with a proof of the number of visits it has received. A server which has been visited by more than h clients would provide the agency with the same proof it would have provided if it had received h visits. For any $j = 1, \ldots, m$, $t = 1, \ldots, \tau$, and $\ell < f \leq h$, we denote with $p^t_{j,f}$ the proof computed by the server $\mathcal{S}_j$ when it has been visited by f distinct clients in time frame t. We refer to such a proof as the f-proof of $\mathcal{S}_j$ in time frame t. Moreover, we denote with $P^t_{j,f}$ the set of all values that $p^t_{j,f}$ can assume. For any $r = \ell + 1, \ldots, h$, we define $L_r = \{\ell + 1, \ldots, r\}$ and we denote by p^t_{j,L_r} the proofs $p^t_{j,\ell+1} \ldots p^t_{j,r}$. Moreover, we denote with P^t_{j,L_r} the set of all values that p^t_{j,L_r} can assume. We also define $L_r = \emptyset$, for any $r < \ell + 1$. To simplify the notation, we define $P^t_{j,L_\ell} = \emptyset$, for any $j = 1, \ldots, m$ and $t = 1, \ldots, \tau$.

The audit agency provides each client with some information about the servers' proofs. For any $i = 1, \ldots, n$, we denote with c_i the information that the audit agency $\mathcal{A}$ gives to the client $\mathcal{C}_i$, and with C_i the set of all possible values of c_i. The information c_i is used by $\mathcal{C}_i$ to compute the information given to the servers when visiting them. For any $i = 1, \ldots, n$, $j = 1, \ldots, m$, and $t = 1, \ldots, \tau$, we denote with $c^t_{i,j}$ the information that the client $\mathcal{C}_i$ sends to the

server $\mathcal{S}_j$ when visiting it in time frame t. Moreover, we denote with $C^t_{i,j}$ the set of all possible values of $c^t_{i,j}$. We require that for any time frame $t = 1, \ldots, \tau$, each client can compute the piece to be given to any visited server. More formally, it holds that

$$H(\mathbf{C}^t_{i,j}|\mathbf{C}_i) = 0, \quad \text{for } i = 1, \ldots, n,\ j = 1, \ldots, m, \ \text{ and } t = 1, \ldots, \tau. \qquad (1)$$

For any $j = 1, \ldots, m$ and $t = 1, \ldots, \tau$, we denote with $X^t_{j,(d_j)}$ the set of the d_j client visits received by server $\mathcal{S}_j$ in time frame t. We require that, for any time frame $t = 1, \ldots, \tau$ and any $f = \ell + 1, \ldots, h$, any server which has been visited by f different clients in time frame t, can compute its $(\ell+1)$-proof,...,f-proof for time frame t. More formally, it holds that

$$H(\mathbf{P}^t_{j,L_f}|\mathbf{X}^t_{j,(f)}) = 0 \text{ for } j = 1, \ldots, m,\ t = 1, \ldots, \tau, \text{ and } f = \ell + 1, \ldots, h. \quad (2)$$

We assume that a certain number, say c with $c \leq \ell$, of clients and a certain number, say s with $s \leq m$, of servers are corrupt. A *corrupt* server can be assisted by corrupt clients and other corrupt servers in order to inflate the count of its visits. A corrupt client $\mathcal{C}_i$ can donate to a corrupt server the whole information c_i received from the audit agency. At time frame t, a corrupt server can donate to another corrupt server the information that it has received during time frames $1, \ldots, t$. For any $j = 1, \ldots, m$ and $t = 1, \ldots, \tau$, we denote with $V_j^{[t]}$ all the information known by a corrupt server $\mathcal{S}_j$ in time frames $1, \ldots, t$. This information includes the sets of client visits received by server $\mathcal{S}_j$ in time frames $1, \ldots, t$. We also define $V_j^{[0]} = \emptyset$. At time frame t a coalition of s corrupt servers $\mathcal{S}_1, \ldots, \mathcal{S}_s$ which decide to cooperate disposes of all information contained in $V_1^{[t-1]}, \ldots, V_s^{[t-1]}$, and of the information provided by the clients visiting such servers during time frame t.

A metering system must be secure against any attempt by corrupt servers to inflate the number of visits they have received. In other words, any $\alpha \leq c$ corrupt clients colluding with any $\beta \leq s$ corrupt servers should not be allowed to infer any information about the value of the proofs to provide to the audit agency. Formally, let $\mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_\alpha}$ be $\alpha \leq c$ corrupt clients, let $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ be a coalition of $1 \leq \beta \leq s$ corrupt servers, and let $B = \{j_1, \ldots, j_\beta\}$. Assume that at some time frame $t \in \{1, \ldots, \tau\}$ each server in the coalition has been visited by at most $z - \alpha$ clients with $z < h$. Then, for any $f = z + 1, \ldots, h$, the servers in the coalition have no information on their f-proofs. More formally, it holds that

$$H(\mathbf{P}^t_{B,f}|\mathbf{C}_{i_1} \ldots \mathbf{C}_{i_\alpha} \mathbf{X}^t_{j_1,(d_{j_1})} \ldots \mathbf{X}^t_{j_\beta,(d_{j_\beta})} \mathbf{V}_B^{[t-1]}) = H(\mathbf{P}^t_{B,f}), \qquad (3)$$
$$\text{for } z < f \leq h,\ t = 1, \ldots, \tau,\ 0 \leq \alpha \leq c, \text{ and } d_{j_v} + \alpha \leq z, \text{ for } v = 1, \ldots, \beta.$$

A metering system satisfying (1), (2), and (3) is termed an $(\ell, h, n, m, \tau, c, s)$ metering system. A cryptographic protocol realizing such a metering system is called metering scheme with pricing.

We want to point out that our definition of corrupt servers is slightly different from that given by Naor and Pinkas in [5]. Indeed, in their model a corrupt server

can donate to another corrupt server only the information collected during the previous time frames, whereas in our model, which is closer to what can actually happen, a corrupt server can donate also the information provided by the visits received in the current time frame.

3 Lower Bounds

In this section we provide lower bounds on the size of the information distributed to clients by the audit agency and on the size of the information distributed to servers by clients.

Since our goal is to prove a lower bound on the size of the information distributed to clients we consider the worst possible case that, at any time frame $t = 1, \ldots, \tau$ and for any corrupt server $\mathcal{S}_j$, the sets $V_j^{[1]}, \ldots, V_j^{[t-1]}$ contain the maximum possible information, in other words, corrupt servers are supposed to receive visits from all clients during the previous time frames $1, \ldots, t-1$. Formally, it holds that

$$H(\mathbf{C}_{i,j}^{t'} | \mathbf{V}_j^{[t]}) = 0, \quad \text{for } i = 1, \ldots, n,\ j = 1, \ldots, m, \text{ and } 1 \leq t' \leq t \leq \tau - 1. \quad (4)$$

Consequently, one has

$$H(\mathbf{P}_{j,L_h}^{t'} | \mathbf{V}_j^{[t]}) = 0, \quad \text{for } j = 1, \ldots, m, \text{ and } 1 \leq t' \leq t \leq \tau - 1. \quad (5)$$

3.1 Technical Lemmas

In order to prove our lower bound on the size of the information distributed to clients, we will resort to the following technical lemmas. The proofs of these lemmas are omitted and can be easily derived.

Lemma 1. *Let* $\mathbf{A}$ *and* $\mathbf{E}$ *be two random variables such that* $H(\mathbf{A}|\mathbf{E}) = 0$*. Then, for any two random variables* $\mathbf{F}$ *and* $\mathbf{G}$*, one has* $H(\mathbf{G}|\mathbf{AEF}) = H(\mathbf{G}|\mathbf{EF})$.

Lemma 2. *Let* $\mathbf{D}$, $\mathbf{E}$, *and* $\mathbf{F}$ *be three random variables such that* $H(\mathbf{F}|\mathbf{DE}) = 0$ *and* $H(\mathbf{F}|\mathbf{E}) = H(\mathbf{F})$*. It results that* $H(\mathbf{D}|\mathbf{E}) = H(\mathbf{F}) + H(\mathbf{D}|\mathbf{EF})$.

3.2 A Lower Bound on the Size of Clients' Information

In this subsection we present a lower bound on the size of the information given to clients by the audit agency.

The next lemma will be useful to prove a lower bound on the size of the information distributed to clients. Due to space constraints, we omit its proof which can be found in [2].

Lemma 3. *Let* Σ *be an* $(\ell, h, n, m, \tau, c, s)$ *metering system. Let* $\mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_c}$ *be the corrupt clients and let* B*, with* $|B| = \beta \leq s$*, be a set of indices of corrupt servers. For any* $j \in B$*,* $t = 1, \ldots, \tau$*, and* $z = \ell - c, \ldots, h - c$*, let* $X_{j,(z)}^t$ *be a set*

of visits from z clients other than $\mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_c}$ to server $\mathcal{S}_j$ in time frame t. In any metering scheme with pricing for Σ one has

$$H(\mathbf{C}_{i_v}|\mathbf{C}_{\{i_1,\ldots,i_c\}\setminus\{i_v\}}\mathbf{X}^t_{B,(r-c-1)}\mathbf{P}^t_{B,L_{r-1}}\mathbf{V}^{[t-1]}_B)$$

$$\geq H(\mathbf{P}^t_{B,r}) + H(\mathbf{C}_{i_v}|\mathbf{C}_{\{i_1,\ldots,i_c\}\setminus\{i_v\}}\mathbf{X}^t_{B,(r-c)}\mathbf{P}^t_{B,L_r}\mathbf{V}^{[t-1]}_B),$$

for $r = \ell+1, \ldots, h$, $v = 1, \ldots, c$, and $t = 1, \ldots, \tau$.

The following theorem provides a lower bound on the information distributed to clients in metering schemes with pricing.

Theorem 1. *Let Σ be an $(\ell, h, n, m, \tau, c, s)$ metering system. Let B, with $|B| \leq s$, be a set of indices of corrupt servers. In any metering scheme with pricing for Σ, it holds that*

$$H(\mathbf{C}_i) \geq \sum_{t=1}^{\tau} H(\mathbf{P}^t_{B,L_h}), \text{ for any } i = 1, \ldots, n.$$

Proof. W.l.o.g. we will assume that $\mathcal{C}_1, \ldots, \mathcal{C}_c$ be the corrupt clients and prove the bound for $\mathcal{C}_1$.
The following inequality, which will be proved later, holds.

$$H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[t-1]}_B) \geq H(\mathbf{P}^t_{B,L_h}) + H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[t]}_B), \tag{6}$$

for any $t = 1, \ldots, \tau$.

Starting from $H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[0]}_B)$ and iteratively applying inequality (6), we get

$$H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[0]}_B) \geq \sum_{t=1}^{\tau} H(\mathbf{P}^t_{B,L_h}) + H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[\tau]}_B). \tag{7}$$

Hence, one has

$$\begin{aligned} H(\mathbf{C}_1) &\geq H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{V}^{[0]}_B) \text{ (from (21) of Appendix A)} \\ &\geq \sum_{t=1}^{\tau} H(\mathbf{P}^t_{B,L_h}) \text{ (from (7) and (18) of Appendix A).} \end{aligned}$$

Now let us prove inequality (6).
For the sake of simplicity and w.l.o.g. we will assume $B = \{1, \ldots, \beta\}$. For any $j \in B$ and $t = 1, \ldots, \tau$, let $X^t_{j,(\ell-c)}$ be a set of visits from $\ell - c$ clients other than $\mathcal{C}_1, \ldots, \mathcal{C}_c$ to server $\mathcal{S}_j$ in time frame t.
Starting from $H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{X}^t_{B,(\ell-c)}\mathbf{P}^t_{B,L_\ell}\mathbf{V}^{[t-1]}_B)$ and iteratively applying Lemma 3, we get

$$H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{X}^t_{B,(\ell-c)}\mathbf{P}^t_{B,L_\ell}\mathbf{V}^{[t-1]}_B)$$

$$\geq \sum_{r=\ell+1}^{h} H(\mathbf{P}^t_{B,r}) + H(\mathbf{C}_1|\mathbf{C}_2 \ldots \mathbf{C}_c\mathbf{X}^t_{B,(h-c)}\mathbf{P}^t_{B,L_h}\mathbf{V}^{[t-1]}_B). \tag{8}$$

Let us consider the two random variables $\mathbf{A} = \mathbf{X}^t_{B,(h-c)}\mathbf{P}^t_{B,L_h}$ and $\mathbf{E} = \mathbf{V}^{[t]}_B$. Using equations (4) and (5), one can prove that

$$H(\mathbf{X}^t_{B,(h-c)}\mathbf{P}^t_{B,L_h}|\mathbf{V}^{[t]}_B) = 0.$$

Hence, $\mathbf{A}$ and $\mathbf{E}$ verify the hypothesis of Lemma 1, and one has

$$\begin{aligned}
&H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{X}^t_{B,(h-c)}\mathbf{P}^t_{B,L_h}\mathbf{V}^{[t-1]}_B)\\
&\quad \geq H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{X}^t_{B,(h-c)}\mathbf{P}^t_{B,L_h}\mathbf{V}^{[t]}_B) \text{ (from (21) of Appendix A)}\\
&\quad = H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{V}^{[t]}_B) \text{ (from Lemma 1).} \qquad (9)
\end{aligned}$$

It follows that

$$\begin{aligned}
&H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{X}^t_{B,(\ell-c)}\mathbf{P}^t_{B,L_\ell}\mathbf{V}^{[t-1]}_B)\\
&\quad \geq \sum_{r=\ell+1}^{h} H(\mathbf{P}^t_{B,r}) + H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{V}^{[t]}_B) \text{ (from (8)–(9))}\\
&\quad \geq H(\mathbf{P}^t_{B,L_h}) + H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{V}^{[t]}_B) \text{ (from (24) of Appendix A).}
\end{aligned}$$

Inequality (6) follows from the above inequality and from (21) of Appendix A which implies

$$H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{V}^{[t-1]}_B) \geq H(\mathbf{C}_1|\mathbf{C}_2 \dots \mathbf{C}_c\mathbf{X}^t_{B,(\ell-c)}\mathbf{P}^t_{B,L_\ell}\mathbf{V}^{[t-1]}_B).$$

□

In Section 2 we did not make any assumption on the entropies of the random variables $\mathbf{P}^t_{j,f}$ and $\mathbf{P}^t_{j,L_f}$, for $j \in \{1,\dots,m\}$, $f \in \{\ell+1,\dots,h\}$, and $t \in \{1,\dots,\tau\}$. Indeed, our results apply to the general case of arbitrary entropies on proofs. Now suppose that $H(\mathbf{P}^{t_1}_{j_1,f_1}) = H(\mathbf{P}^{t_2}_{j_2,f_2})$ and $H(\mathbf{P}^{t_1}_{j_1,L_f}) = H(\mathbf{P}^{t_2}_{j_2,L_f})$, for all $j_1, j_2 \in \{1,\dots,m\}$, $f_1, f_2, f \in \{\ell+1,\dots,h\}$, and $t_1, t_2 \in \{1,\dots,\tau\}$. We denote these common entropies by $H(\mathbf{P})$ and $H(\mathbf{P}_{L_f})$, respectively. If the proof sequences of the s corrupt servers are statistically independent, then Theorem 1 implies

$$H(\mathbf{C}) \geq s\tau H(\mathbf{P}_{L_h}), \text{ for any client } \mathcal{C}. \qquad (10)$$

Moreover, if for any server $\mathcal{S}_j$, the $(\ell+1)$-proof, ..., $h-$proof associated to $\mathcal{S}_j$ are statistically independent, then inequality (10) implies

$$H(\mathbf{C}) \geq s\tau(h-\ell)H(\mathbf{P}), \text{ for any client } \mathcal{C}. \qquad (11)$$

If the proofs of the servers are also uniformly chosen in a finite field F, then inequality (11) implies

$$H(\mathbf{C}) \geq (h-\ell)s\tau \log|F|, \text{ for any client } \mathcal{C}. \qquad (12)$$

This bound is tight, as in Section 4 we present a protocol for an $(\ell, h, n, m, \tau, c, s)$ metering system in which the audit agency distributes *exactly* this information to clients.

3.3 A Lower Bound on the Size of Servers' Information

In the following we provide a lower bound on the size of the information given to servers by clients in metering schemes with pricing.

The following theorem provides another lower bound on the communication complexity of the metering scheme. It implicitly shows that the size of the information each client has to give out when visiting a server is lower bounded by the size of the proofs the server could reconstruct.

Theorem 2. *Let Σ be an $(\ell, h, n, m, \tau, c, s)$ metering system. In any metering scheme with pricing for Σ, it holds that*

$$H(\mathbf{C}_{i,j}^t) \geq H(\mathbf{P}_{j,L_h}^t), \text{ for any } i = 1, \ldots, n, \ j = 1, \ldots, m, \text{ and } t = 1, \ldots, \tau.$$

Proof. The following inequality, which will be proved later, holds.

$$H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(r-1)}^t \mathbf{P}_{j,L_{r-1}}^t) \geq H(\mathbf{P}_{j,r}^t) + H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(r)}^t \mathbf{P}_{j,L_r}^t), \tag{13}$$

for any $i = 1, \ldots, n$, $j = 1, \ldots, m$, $t = 1, \ldots, \tau$, $r = \ell + 1, \ldots, h$, and any set of visits $X_{j,(r-1)}^t$ from $r-1$ clients other than $\mathcal{C}_i$ to server $\mathcal{S}_j$ in time frame t.

Starting from $H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(\ell)}^t \mathbf{P}_{j,L_\ell}^t)$ and iteratively applying (13), one gets

$$\begin{aligned} H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(\ell)}^t \mathbf{P}_{j,L_\ell}^t) &\geq \sum_{r=\ell+1}^{h} H(\mathbf{P}_{j,r}^t) + H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(h)}^t \mathbf{P}_{j,L_h}^t) \\ &\geq H(\mathbf{P}_{j,L_h}^t) \text{ (from (22) and (18) of Appendix A).} \end{aligned}$$

The theorem follows from the above inequality and from (21) of Appendix A which implies $H(\mathbf{C}_{i,j}^t) \geq H(\mathbf{C}_{i,j}^t | \mathbf{X}_{j,(\ell)}^t \mathbf{P}_{j,L_\ell}^t)$.

Now let us prove inequality (13).
Let $\mathbf{A}' = \mathbf{P}_{j,L_{r-1}}^t$, $\mathbf{D} = \mathbf{C}_{i,j}^t$, $\mathbf{E} = \mathbf{X}_{j,(r-1)}^t \mathbf{P}_{j,L_{r-1}}^t$, $\mathbf{E}' = \mathbf{X}_{j,(r-1)}^t$, and $\mathbf{F} = \mathbf{P}_{j,r}^t$.
If $\ell + 2 \leq r \leq h$, then from (2) one has $H(\mathbf{P}_{j,L_{r-1}}^t | \mathbf{X}_{j,(r-1)}^t) = 0$. If $r = \ell + 1$ then $P_{j,L_\ell}^t = \emptyset$ and consequently $H(\mathbf{P}_{j,L_\ell}^t | \mathbf{X}_{j,(\ell)}) = 0$. Hence, one has that the random variables $\mathbf{A}'$ and $\mathbf{E}'$ verify the hypothesis of Lemma 1 and consequently $H(\mathbf{F}|\mathbf{A}'\mathbf{E}') = H(\mathbf{F}|\mathbf{E}')$. Then, it results that

$$\begin{aligned} H(\mathbf{P}_{j,r}^t | \mathbf{X}_{j,(r-1)}^t \mathbf{P}_{j,L_{r-1}}^t) &= H(\mathbf{P}_{j,r}^t | \mathbf{X}_{j,(r-1)}^t) \text{ (from Lemma 1)} \\ &\geq H(\mathbf{P}_{j,r}^t | \mathbf{X}_{j,(r-1)}^t \mathbf{V}_j^{[t-1]}) \text{ (from (21) of Appendix A)} \\ &= H(\mathbf{P}_{j,r}^t) \text{ (from (3)).} \end{aligned}$$

From the above inequality and from (19) of Appendix A which implies $H(\mathbf{P}_{j,r}^t | \mathbf{X}_{j,(r-1)}^t \mathbf{P}_{j,L_{r-1}}^t) \leq H(\mathbf{P}_{j,r}^t)$, it follows that

$$H(\mathbf{P}_{j,r}^t | \mathbf{X}_{j,(r-1)}^t \mathbf{P}_{j,L_{r-1}}^t) = H(\mathbf{P}_{j,r}^t). \tag{14}$$

Moreover, it results that

$$H(\mathbf{P}^t_{j,r}|\mathbf{X}^t_{j,(r-1)}\mathbf{C}^t_{i,j}\mathbf{P}^t_{j,L_{r-1}}) \leq H(\mathbf{P}^t_{j,r}|\mathbf{X}^t_{j,(r-1)}\mathbf{C}^t_{i,j}) \text{ (from (21) of Appendix A)}$$
$$= 0 \text{ (from (2))}. \quad (15)$$

Equations (14)–(15) imply that the random variables $\mathbf{D}$, $\mathbf{E}$, and $\mathbf{F}$ verify the hypothesis of Lemma 2, and consequently one has $H(\mathbf{D}|\mathbf{E}) = H(\mathbf{F}) + H(\mathbf{D}|\mathbf{EF})$. Hence, one gets

$$\begin{aligned} H(\mathbf{C}^t_{i,j}|\mathbf{X}^t_{j,(r-1)}\mathbf{P}^t_{j,L_{r-1}}) &= H(\mathbf{P}^t_{j,r}) + H(\mathbf{C}^t_{i,j}|\mathbf{X}^t_{j,(r-1)}\mathbf{P}^t_{j,L_{r-1}}\mathbf{P}^t_{j,r}) \\ &\quad \text{(from Lemma 2)} \\ &= H(\mathbf{P}^t_{j,r}) + H(\mathbf{C}^t_{i,j}|\mathbf{X}^t_{j,(r-1)}\mathbf{P}^t_{j,L_r}) \\ &\geq H(\mathbf{P}^t_{j,r}) + H(\mathbf{C}^t_{i,j}|\mathbf{X}^t_{j,(r)}\mathbf{P}^t_{j,L_r}) \\ &\quad \text{(from (21) of Appendix A)}. \end{aligned}$$

Thus, inequality (13) holds. □

If for any server $\mathcal{S}_j$, the $(\ell+1)$-proof, ..., h-proof associated to $\mathcal{S}_j$ are statistically independent and uniformly chosen in a finite field F, then Theorem 2 implies

$$H(\mathbf{C}^t_{i,j}) \geq (h-\ell)\log|F|, \quad \text{for } i = 1,\ldots,n,\ j = 1,\ldots,m, \text{ and } t = 1,\ldots,\tau. \quad (16)$$

This bound is tight, as in Section 4 we present a protocol for an $(\ell, h, n, m, \tau, c, s)$ metering system in which the clients distribute *exactly* this information to servers.

4 The Scheme

In this section we present a scheme for an $(\ell, h, n, m, \tau, c, s)$ metering system achieving the bounds (12) and (16) of Section 3. Along the same line as Naor and Pinkas [5], we use a modified version of Shamir's secret sharing polynomial [7]. The proofs are points of a finite field $GF(q)$ where q is a sufficiently large prime number.

In the following we use the term regular visits to indicate visits performed by non corrupt clients. Moreover, we denote with "$\circ$" an operator mapping each pair (j,t), with $j = 1,\ldots,m$ and $t = 1,\ldots,\tau$, to an element of $GF(q)$, and having the property that no distinct two pairs (j,t) and (j',t') are mapped to the same element.

Initialization:
The audit agency $\mathcal{A}$ chooses $h-\ell$ random polynomials $P_{\ell+1}(x,y),\ldots,P_h(x,y)$ over $GF(q)$, where, for $z=\ell+1,\ldots,h$, the polynomial $P_z(x,y)$ is of degree $z-1$ in x and degree $s\tau-1$ in y. Then, $\mathcal{A}$ sends to each client $\mathcal{C}_i$ the $h-\ell$ univariate polynomials $P_{\ell+1}(i,y),\ldots,P_h(i,y)$ which are of degree $s\tau-1$.

Regular Operation for Time Frame t:
When the client $\mathcal{C}_i$ visits the server $\mathcal{S}_j$ in time frame t, it sends to $\mathcal{S}_j$ the $h-\ell$ points $P_{\ell+1}(i,j\circ t),\ldots,P_h(i,j\circ t)$.

Proof Generation and Verification:
Assume that the server $\mathcal{S}_j$ has been visited by a number z of clients greater than ℓ and less than or equal to h in time frame t. Then, the server performs a Lagrange interpolation of the polynomial $P_z(x,j\circ t)$ and computes the value $P_z(0,j\circ t)$. This value constitutes the z-proof of $\mathcal{S}_j$, i.e., the proof that $\mathcal{S}_j$ has received z visits. The server $\mathcal{S}_j$ sends the pair $(P_z(0,j\circ t),z)$ to the audit agency. The audit agency can verify the proof by evaluating the polynomial $P_z(x,y)$ at the point $(0,j\circ t)$.

Figure 1. A metering scheme for an (ℓ,h,n,m,τ,c,s) metering system.

Theorem 3. *The scheme described in Figure 1 is a metering scheme for an (ℓ,h,n,m,τ,c,s) metering system.*

Proof. We need to prove that the scheme of Figure 1 satisfies equations (1), (2), and (3) of Section 2. It is immediate to verify that the scheme satisfies (1). Indeed, for any $i=1,\ldots,n$, the information given by the audit agency to the client $\mathcal{C}_i$ consists of the univariate polynomials $P_{\ell+1}(i,y),\ldots,P_h(i,y)$, and for any $j=1,\ldots,m$, the information given to the server $\mathcal{S}_j$ by client $\mathcal{C}_i$ is obtained by evaluating the univariate polynomials $P_{\ell+1}(i,y),\ldots,P_h(i,y)$ at $j\circ t$.
It is also very easy to verify that the scheme satisfies equation (2). Assume that a server $\mathcal{S}_j$ has been visited by f, with $\ell+1\le f\le h$, clients at time frame t. Then, $\mathcal{S}_j$ knows f points of each of the polynomials $P_{\ell+1}(x,j\circ t),\ldots,P_f(x,j\circ t)$. Since these polynomials are all of degree less than or equal to $f-1$ in x, then the server can compute their coefficients by using Lagrange interpolation. In particular, it can compute its f-proof for t by evaluating the polynomial $P_f(x,j\circ t)$ in 0. If the server $\mathcal{S}_j$ has been visited by a number of clients greater than or equal to h in time frame t, then it can reconstruct the $h-\ell$ polynomials, i.e., it can reconstruct all the proofs for the time frame t.
Now we need to prove that our scheme satisfies equation (3). We consider the worst possible case that at any time frame $t=1,\ldots,\tau$, all corrupt clients decide to cooperate with all corrupt servers and that corrupt servers have collected the maximum possible information during the previous time frames $1,\ldots,t-1$. In other words, for any time frame $t=1,\ldots,\tau$, we assume that each corrupt client $\mathcal{C}_i$ donates its polynomials $P_{\ell+1}(i,y),\ldots,P_h(i,y)$ to all corrupt servers, and that any corrupt server $\mathcal{S}_j$ knows the polynomials $P_{\ell+1}(x,j\circ t'),\ldots,P_h(x,j\circ t')$, for $t'=1,\ldots,t-1$. In order to prove that our scheme satisfies equation (3), we need to prove that for any time frame $t=1,\ldots,\tau$, and for any $z=\ell+1,\ldots,h$,

a coalition of $\beta \leq s$ corrupt servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ is not able to calculate the proofs $P_z(0, j_1 \circ t), \ldots, P_z(0, j_\beta \circ t)$ if each server in the coalition receives less than $z - c$ regular visits at time frame t. In order to calculate $P_z(0, j \circ t)$, the servers should be able to interpolate either the polynomial $P_z(x, j \circ t)$ or the bivariate polynomial $P_z(x, y)$. Let us suppose that $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ be a coalition of $\beta \leq s$ corrupt servers which decide to cooperate in order to inflate the counts of their client visits at some time frame t, with $1 \leq t \leq \tau$. The information that a corrupt client $\mathcal{C}_i$ donates to a corrupt server is equivalent to the $s\tau$ coefficients of each of the polynomials $P_{\ell+1}(i, y), \ldots, P_h(i, y)$. For $v = 1, \ldots, \beta$, the information collected by each corrupt server $\mathcal{S}_{j_v}$ during the previous time frames is equivalent to the coefficients of the polynomials $P_{\ell+1}(x, j_v \circ t'), \ldots, P_h(x, j_v \circ t')$, for any $t' = 1, \ldots, t-1$. Suppose that at time frame t, the server $\mathcal{S}_{j_v}$, $v \in \{1, \ldots, \beta\}$, receives g_{j_v} regular visits. Then, the overall information on $P_z(x, y)$ held by the servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ consists of

$$cs\tau + \beta(t-1)z + \sum_{v=1}^{\beta} g_{j_v} - c\beta(t-1) \tag{17}$$

points. The first term of (17) is the information donated by the c corrupt clients, the second term is the information collected by all servers in the coalition during the previous time frames, the third term is the information provided by the client visits at time frame t, and the last term is the information which has been counted twice. For any $z = \ell + 1, \ldots, h$, we will prove that the servers in the coalition are unable to interpolate the polynomial $P_z(x, y)$ if each server in the coalition receives less than $z - c$ regular visits. Notice that for any $1 \leq \beta \leq s$, $t = 1, \ldots, \tau$ and $z = \ell + 1, \ldots, h$, if $g_{j_1}, \ldots, g_{j_\beta}$ are all smaller than $z - c$, then expression (17) is strictly less than $zs\tau$. Consequently, for any choice of $a \in GF(q)$ and for any $j = 1, \ldots, m$, there is a polynomial $R(x, y)$ which is consistent with the information held by the servers in the coalition and such that $R(0, j \circ t) = a$. Hence, the corrupt servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ have probability at most $1/q$ of guessing the z-proof $P_z(0, j_v \circ t)$, for any $v = 1, \ldots, \beta$ and any time frame $t = 1, \ldots, \tau$.

Alternatively, any corrupt server $\mathcal{S}_{j_v}$, $v = 1, \ldots, \beta$, might try to calculate its z-proof $P_z(0, j_v \circ t)$ by interpolating the polynomial $P_z(x, j_v \circ t)$. Notice that for any $v, w \in \{1, \ldots, \beta\}$, with $w \neq v$, the information held by the server $\mathcal{S}_{j_w}$ is of no help in calculating the polynomial $P_z(x, j_v \circ t)$. We will prove that any corrupt server $\mathcal{S}_{j_v}$, $v = 1, \ldots, \beta$, is not able to calculate its z-proof $P_z(0, j_v \circ t)$ if it has received less than $z - c$ regular visits. Then, let us assume $g_{j_v} < z - c$, for $v = 1, \ldots, \beta$. Each corrupt client $\mathcal{C}_i$ donates to $\mathcal{S}_{j_v}$ the polynomial $P_z(i, y)$ from which $\mathcal{S}_{j_v}$ can calculate the value $P_z(i, j_v \circ t)$. Notice that for any non corrupt client $\mathcal{C}_i$, the server $\mathcal{S}_{j_v}$ is not able to evaluate the polynomial $P_z(i, y)$. Indeed, in order to evaluate the polynomial $P_z(i, y)$, $\mathcal{S}_{j_v}$ should know $s\tau$ points of this polynomial. Hence, $\mathcal{S}_{j_v}$ can calculate only c values of $P_z(x, j_v \circ t)$ in addition to those provided by the g_{j_v} visits performed by non corrupt clients. Consequently, the overall number of points of $P_z(x, j_v \circ t)$ known to $\mathcal{S}_{j_v}$ is less than z. Let $i_1, \ldots, i_c$ be the indices of the corrupt clients and $d_1, \ldots, d_{g_{j_v}}$ be the indices

of the clients which have visited $\mathcal{S}_{j_v}$ at time frame t. For any choice of a point $a \in GF(q)$ there is a polynomial $Q(x)$ such that $Q(0) = a$ and $Q(i) = P_z(i, j_v \circ t)$, for $i \in \{i_1, \ldots, i_c, d_1, \ldots, d_{g_{j_v}}\}$. Hence, the server $\mathcal{S}_{j_v}$ has probability at most $1/q$ of guessing its z-proof for time frame t. Moreover, as already observed, for any corrupt server $\mathcal{S}_{j_v}$, with $v \in \{1, \ldots, \beta\}$, the servers $\{\mathcal{S}_{j_w}\}_{w \in \{1,\ldots,\beta\} \setminus \{v\}}$ have no information on $P_z(0, j_v \circ t)$. Consequently, for any $v \in \{1, \ldots, \beta\}$, all corrupt servers in the coalition have probability at most $1/q$ of guessing the z-proof of $\mathcal{S}_{j_v}$ for time frame t.

From the above discussion it follows that both in the case when $\mathcal{S}_{j_1} \ldots, \mathcal{S}_{j_\beta}$ are trying to interpolate the bivariate polynomial $P_z(x, y)$, and in the case when $\mathcal{S}_{j_1} \ldots, \mathcal{S}_{j_\beta}$ are trying to interpolate the polynomial $P_z(x, j \circ t)$, the probability that they guess one of the z-proofs $P_z(0, j_1 \circ t), \ldots, P_z(0, j_\beta \circ t)$ is at most $1/q$. Consequently, the probability that a coalition of $\beta \leq s$ corrupt servers guesses the whole vector $(P_z(0, j_1 \circ t), \ldots, P_z(0, j_\beta \circ t))$ is at most $1/q^\beta$. □

Notice that in the above scheme the size of the information given to any client is $(h - \ell) s\tau \log q$, whereas the size of the information that each server receives from a client during a regular visit is $(h - \ell) \log q$. It easy to see that this protocol achieves the bounds (12) and (16) of Section 3.

5 Open Problems

An interesting open problem is to consider metering systems in which each server is associated with a distinct pair of thresholds (ℓ, h). An even more challenging problem would be to consider a generalization of such metering systems in which the thresholds associated to servers may change dynamically at each time frame.

Another open problem is to consider different classes of clients. Each class is assigned a weight and the amount of money paid to servers depends not only on the number of clients they served but also on the classes those clients belong to.

References

1. V. Anupam, A. Mayer, K. Nissim, B. Pinkas, M. K. Reiter, *On the Security of Pay-Per-Click and Other Web Advertising Schemes*, 8th International World Wide Web Conference.
2. C. Blundo, A. De Bonis, and B. Masucci, *Optimal Pricing on the Web*, submitted for publication.
3. T. M. Cover and J. A. Thomas, *Elements of Information Theory.* John Wiley & Sons, 1991.
4. M. Franklin and D. Malkhi, *Auditable Metering with Lightweight Security*, in "Financial Cryptography '97", Lecture Notes in Computer Science, Vol. **1318**, pp. 151–160, 1997.
5. M. Naor and B. Pinkas, *Secure and Efficient Metering*, in "Advances in Cryptology - EUROCRYPT '98", Lecture Notes in Computer Science, Vol. **1403**, pp. 576–590, 1998.

6. M. Naor and B. Pinkas, *Secure Accounting and Auditing on the Web*, Computer Networks and ISDN Systems, Vol. **40**, Issues 1-7, pp. 541–550, 1998.
7. A. Shamir, *How to Share a Secret*, Comm. ACM, Vol. **22**, No. 11, pp. 612–613, 1979.

A Information Theory Background

In this section we review the basic concepts of Information Theory used in our definitions and proofs. For a complete treatment of the subject the reader is advised to consult [3].

Given a probability distribution $\{Pr_{\mathbf{X}}(x)\}_{x\in X}$ on a set X, we define the *entropy* [1] of $\mathbf{X}$, denoted by $H(\mathbf{X})$, as

$$H(\mathbf{X}) = -\sum_{x\in X} Pr_{\mathbf{X}}(x)\log Pr_{\mathbf{X}}(x).$$

The entropy satisfies the property $0 \leq H(\mathbf{X}) \leq \log|X|$, where $H(\mathbf{X}) = 0$ if and only if there exists $x_0 \in X$ such that $Pr_{\mathbf{X}}(x_0) = 1$; whereas $H(\mathbf{X}) = \log|X|$ if and only if $Pr_{\mathbf{X}}(x) = 1/|X|$, for all $x \in X$.

Given two sets X and Y and a joint probability distribution on their cartesian product, the *conditional entropy* $H(\mathbf{X}|\mathbf{Y})$, is defined as

$$H(\mathbf{X}|\mathbf{Y}) = -\sum_{y\in Y}\sum_{x\in X} Pr_{\mathbf{Y}}(y)Pr(x|y)\log Pr(x|y).$$

From the definition of conditional entropy it is easy to see that

$$H(\mathbf{X}|\mathbf{Y}) \geq 0. \tag{18}$$

The *mutual information* between $\mathbf{X}$ and $\mathbf{Y}$ is defined by $I(\mathbf{X};\mathbf{Y}) = H(\mathbf{X}) - H(\mathbf{X}|\mathbf{Y})$ and enjoys the following properties: $I(\mathbf{X};\mathbf{Y}) = I(\mathbf{Y};\mathbf{X})$ and $I(\mathbf{X};\mathbf{Y}) \geq 0$, from which one gets

$$H(\mathbf{X}) \geq H(\mathbf{X}|\mathbf{Y}). \tag{19}$$

Given three sets X, Y, Z and a joint probability distribution on their cartesian product, the *conditional mutual information* between $\mathbf{X}$ and $\mathbf{Y}$ given $\mathbf{Z}$ is

$$I(\mathbf{X};\mathbf{Y}|\mathbf{Z}) = H(\mathbf{X}|\mathbf{Z}) - H(\mathbf{X}|\mathbf{Z}\mathbf{Y}) \tag{20}$$

and enjoys the following properties: $I(\mathbf{X};\mathbf{Y}|\mathbf{Z}) = I(\mathbf{Y};\mathbf{X}|\mathbf{Z})$ and $I(\mathbf{X};\mathbf{Y}|\mathbf{Z}) \geq 0$. Since the conditional mutual information is always non-negative we get

$$H(\mathbf{X}|\mathbf{Z}) \geq H(\mathbf{X}|\mathbf{Z}\mathbf{Y}). \tag{21}$$

Given any $n \geq 1$ sets, $X_1, \ldots, X_n$ and a joint probability distribution on their cartesian product, it holds that

$$\sum_{i=1}^{n} H(\mathbf{X}_i) \geq H(\mathbf{X}_1 \ldots \mathbf{X}_n). \tag{22}$$

[1] All logarithms in this paper are to the base 2.

Given $n+1$ sets $X_1, \ldots, X_n, Y$ and a joint probability distribution on their cartesian product, the entropy of $\mathbf{X}_1 \ldots \mathbf{X}_n$ given $\mathbf{Y}$ can be expressed as

$$H(\mathbf{X}_1 \ldots \mathbf{X}_n | \mathbf{Y}) = H(\mathbf{X}_1 | \mathbf{Y}) + \sum_{i=2}^{n} H(\mathbf{X}_i | \mathbf{X}_1 \ldots \mathbf{X}_{i-1} \mathbf{Y}) \tag{23}$$

and enjoys the following property:

$$H(\mathbf{X}_1 \ldots \mathbf{X}_n | \mathbf{Y}) \leq \sum_{i=1}^{n} H(\mathbf{X}_i | \mathbf{Y}). \tag{24}$$

ℓ, h	thresholds
n	number of clients
m	number of servers
τ	number of time frames
c	number of corrupt clients
s	number of corrupt servers
$\mathbf{C}_i$	information distributed to client $\mathcal{C}_i$
$\mathbf{C}_{i,j}^t$	visit from client $\mathcal{C}_i$ to server $\mathcal{S}_j$ in time frame t
$B = \{j_1, \ldots, j_\beta\}$	indices of the corrupt servers, $\beta \leq s$
$\mathbf{C}_{i,B}^t$	visits from client $\mathcal{C}_i$ to servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ in time frame t
$\mathbf{X}_{j,(d_j)}^t$	visits from d_j clients to server $\mathcal{S}_j$ in time frame t
$\mathbf{X}_{B,(z)}^t$	visits from z clients to servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ in time frame t
$\mathbf{V}_j^{[t]}$	information collected by server $\mathcal{S}_j$ in time frames $1, \ldots, t$
$\mathbf{V}_B^{[t]}$	information collected by servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$ in time frames $1, \ldots, t$
$\mathbf{P}_{j,f}^t$	f-proof for server $\mathcal{S}_j$, where $f \in \{\ell+1, \ldots, h\}$
$\mathbf{P}_{B,f}^t$	f-proofs for servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$
$L_r = \{\ell+1, \ldots, r\}$	indices of proofs, where $r \in \{\ell+1, \ldots, h\}$
$\mathbf{P}_{j,L_r}^t$	$(\ell+1)$-proof,$\ldots r$-proof for server $\mathcal{S}_j$
$\mathbf{P}_{B,L_r}^t$	$(\ell+1)$-proofs,$\ldots r$-proofs for servers $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_\beta}$

Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms

Oliver Theel

Darmstadt University of Technology
Dept. of Computer Science
Alexanderstr. 10
D-64283 Darmstadt, Germany
theel@informatik.tu-darmstadt.de

Abstract. A particularly suitable design strategy for constructing a robust distributed algorithm is to endow it with a self-stabilization property. Such a property guarantees that the system will always return to and stay within a specified set of legal states within bounded time regardless of its initial state. A self-stabilizing application therefore has the potential of recovering from the effects of arbitrary transient failures. However, to actually verify that an application self-stabilizes can be quite tedious with current proof methodologies and is non-trivial. The self-stabilizing property of distributed algorithms exhibits interesting analogies to stabilizing feedback systems used in various engineering domains. In this paper we would like to show that techniques from control theory, namely Ljapunov's "Second Method," can be used to more easily verify the self-stabilization property of distributed algorithms.

1 Introduction

A very promising design strategy for constructing a robust distributed application is to design it as a *self-stabilizing algorithm* [16]. Informally, an algorithm has the self-stabilization property, if – starting from an illegal state – it is guaranteed to return to a specified set of legal states after a finite period of time (*convergence*). Additionally, the set of legal states must be closed under normal system execution, meaning that the algorithm does not voluntarily switch to any illegal state (*closure*). The definition of legal and illegal states depends on the particular application. Generally, all legal states are specified (e.g., by a state predicate) and illegal states are defined to be those states which are not legal states. Unfortunately, the verification of self-stabilizing algorithms is a complicated task [13]. Consequently, the research community is highly engaged in finding more adequate verification techniques.

The self-stabilization property of a distributed algorithm as described above exhibits interesting analogies to stable feedback systems used in various engineering domains, like electrical and mechanical engineering. Informally, a feedback system is stable, if after a certain finite period of time, the system reaches and remains in a pre-defined state [10]. Contrary to the self-stabilization research

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 209–222, 2000.

domain, which is a rather new area of research in computer science, control theory in the engineering domain has a century-old background and offers a broad theoretical foundation with powerful criteria for reasoning about the stability of feedback systems.

The aim of our research is to narrow the gap between self-stabilization and control theory by adopting criteria originally used for deciding on the stability of feedback systems for proving self-stabilization of distributed algorithms. In [19] we proved the self-stabilization property of a distributed algorithm by modeling it as a discrete *linear* feedback system and subsequently reasoning about the location of the roots of their transfer function. But it must be emphasized that although the transfer function technique was successful for the particular distributed algorithm (a verification of this algorithm based on traditional computer science techniques has not been achieved yet), it cannot be applied for the more general case of non-linear systems.

In this paper, we present a generalization of our verification technique in such a way that it can be adopted even for the non-linear case by the use of Ljapunov's "Second Method" [14]. We would like to draw attention to the fact that the new technique in many cases eases the construction of a proof.

The paper is structured as follows. In the next section, we state the verification problem of self-stabilizing algorithms and describe the verification technique traditionally used in computer science. In Sect. 3, we present an alternative verification approach. We state our underlying system model, describe how distributed algorithms given by guarded commands are mapped to it, and give a criterion for reasoning about stability which forms the heart of Ljapunov's theory. In Sect. 4, we present a sample algorithm whose self-stabilizing property is proven using a new verification technique. Finally, Sect. 5 concludes the paper.

2 Problem Statement and Traditional Verification Technique

A major problem associated with self-stabilizing distributed algorithms is their verification, i.e., the proof that they actually work as required.

Let $\mathbb{C}$ be the set of all possible system states of a system S. Assume that P be a predicate that specifies a subset of $\mathbb{C}$. The sets specified by P are called *legal states* whereas all other states of $\mathbb{C}$ are called *illegal states*. P must be guaranteed through the concept of self-stabilization.

Definition 1 (Self-Stabilization [16]). *A system S is self-stabilizing towards predicate P on $\mathbb{C}$ iff*

(S1) if P holds for $c \in \mathbb{C}$ then P also holds for all subsequent system states, and
(S2) starting from an arbitrary system state, S reaches after a finite number of steps a system state where P holds.

A *step* is an evaluation cycle which leads to a state change. The constraints *(S1)* and *(S2)* are also called *closure* and *convergence*, respectively.

In order to verify that an algorithm self-stabilizes, both constraints must be proven. This is generally done for each constraint separately.

Proving closure: A proof of the closure constraint shows that the stability predicate P is an invariant of the algorithm. Doing so is straightforward: assume P holds at the beginning of a cycle. Then is must be shown that any possible step taken will again result in a system state where P holds.

Proving convergence: A proof of the convergence constraint is much more complicated. Generally, it requires the use of a *variant function* [15] defined on the system state. The values of such a variant function are bounded from below and decrease with every step. From such a proof it follows that there will be a point in time where the variant function reaches a minimum. When the minimum is reached, it must be assured that the system is in a legal state and that switching between legal states does not result in a change of the value of the variant function. The difficulty of this verification strategy lies in the fact that finding such a variant function for a given system requires experience and inspiration since the function must in itself bear the "essence of convergence" of the system. Thus, deriving a variant function for arbitrary systems is regarded as an art rather than a craft.

A famous example for proving self-stabilization is Dijkstra's token ring protocol for mutual exclusion [4]: although the algorithm was presented in 1974 it took 12 year before it was finally proven correct (see [5]). It is not claimed that during this long period of time researchers worked on a successful proof without any interruption, but it strongly indicates that even for a quite simple-looking self-stabilizing algorithm, verification is by no means simple.

It should be noted that advanced techniques have been proposed in order to ease program verification. Among them are *convergence stairs* [8] and *compositions* [17]. These techniques are built upon the traditional verification technique as described above.

3 An Alternative Verification Technique

The basic idea of our alternative verification approach is to take advantage of a technique called *Ljapunov's "Second Method"* [14]. Through this method it is possible to more easily identify a variant-like function as described above. We could observe that in quite a few cases, the technique leads to a quasi automatic identification of this function. As reported by Kalman and Bertram in [11, page 371], the objective of Ljapunov's "Second Method" is *to answer the questions of stability of difference equations, utilizing the given form of the equations but without explicit knowledge of the solutions.* This contrasts Ljapunov's "First Method" where *explicit* representations of those solutions are required. Starting point for the "Second Method" is the observation that finding a solution for a system given as a collection of difference equations is sometimes quite complicated if not impossible. But having a solution, stability or instability can then easily be proven since it is then straightforward to reason about trajectories and equilibrium states of the system in time domain [21]. Through his "Second

Method" Ljapunov circumvents the problem of finding an explicit solution by the following observation: assume the kinetic energy level of a system can be described as a function of the system state. Then, a real-world system (e.g., a physical system with components subject to friction etc.) which is initially started at a certain kinetic energy level will loose kinetic energy as time proceeds. At some point in time, regardless of the initial system state (and thereby regardless of the amount of kinetic energy present in the system), all of the system's kinetic energy will have left. Consequently, the system will enter and remain in a so-called *equilibrium state*. In other words: the system has stabilized. Ljapunov builds upon this basic scheme by generalizing the "notion of kinetic energy" such that even systems without energy loss or without a concept of kinetic energy can be treated.

When exploiting this method originally used for feedback systems of the engineering domain, for self-stabilizing distributed algorithms of the computer science domain, a possible strategy is to model the latter in terms of the former. This approach is pursued in the following.

3.1 System Model

Figure 1 shows a system model for which Ljapunov's method can be adopted. We will show that it allows the modeling of a distributed algorithm which is given by guarded commands and whose self-stabilization property is to be verified. The system model is used to represent *time-discrete variable structure dynamic*

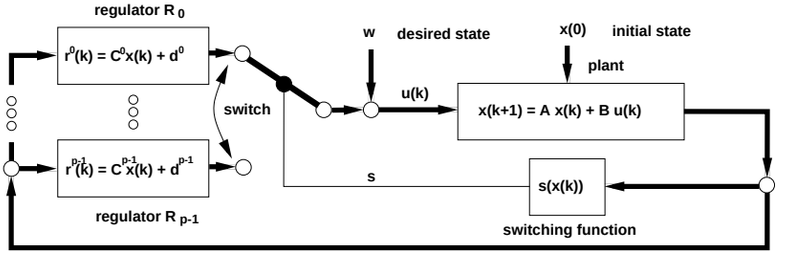


Fig. 1. *System model*

systems [20]. The state of such a system at discrete and abstract time k is given by the n-dimensional *state vector* $\mathsf{x}(k) \in \mathbb{R}^n$. Its i-th component is referred to as $\mathsf{x}_i(k)$, $i = 1, \ldots, n$. The *initial state* is given by $\mathsf{x}(0)$. Depending on a *switching function*, given as a scalar function $s : \mathbb{R}^n \mapsto \mathbb{N}_0$, exactly one of p *regulators* R_i, $i = 0, \ldots, p-1$, is selected at time k and remains active until time $k+1$. $\mathsf{w} \in \mathbb{R}^n$ is an n-dimensional vector representing the *desired state* of the system. A regulator R_i is a sub-system with input $\mathsf{x}(k)$ and output $\mathsf{r}^i(k) = \mathbf{C}^i \cdot \mathsf{x}(k) + \mathsf{d}^i$. $\mathsf{r}^i(k), \mathsf{d}^i \in \mathbb{R}^n$ are n-dimensional vectors which are time-variant and time-invariant, respectively. $\mathbf{C}^i \in \mathbb{M}(n \times n, \mathbb{R})$ is a $n \times n$-dimensional time-invariant

matrix. When selected at time k, regulator R_i maps the state vector $\mathsf{x}(k)$ and the desired state vector w to the *control vector* $\mathsf{u}(k)$, i.e., $\mathsf{u}(k) = \mathsf{w} + \mathsf{r}^i(k)$. The control vector serves as input to the *plant*. The plant is characterized by a system of difference equations. The plant's output at time k is given by the state vector $\mathsf{x}(k)$. Depending on the state vector and the control vector at time k, the plant's output at time $k+1$ evaluates to $\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot \mathsf{u}(k)$. $\mathbf{A}$, $\mathbf{B} \in \mathbb{M}(n \times n, \mathbb{R})$ are time-invariant $n \times n$ matrices. $\mathbf{A}$ is called the *state matrix* and $\mathbf{B}$ is called the *control matrix*.

3.2 Matching Distributed Algorithms to the System Model

We assume that a distributed algorithm is given as a collection of n processes $P_1, \ldots, P_n$ whose program bodies are collections of *guarded commands* [3]. Figure 2 shows a process P_i of such a generic distributed algorithm. The local state of a process is defined by a local variable S_i. The initial, but probably arbitrary local state is given by s_i. The *communication variables* $L_{i,j}$ represent local states of other processes available to process P_i. Communication is achieved via the lookup and modification of these communication variables. While the distributed algorithm executes, all of its processes cyclically evaluate their guards. A guard is a boolean expression over the local state and communication variables. Guards which evaluate to "true" as well as the guarded commands they belong to are called *active*. A global entity, called *central daemon* selects within each evaluation cycle a subset of active guarded commands for execution. The nature of this subset depends on the central daemon's strategy. In the scope of this paper, we assume *serial execution semantics*, i.e., the central daemon *selects exactly one* active guarded command if one or more guarded commands are active. This leads to an atomic execution of the selected guarded command's action. Furthermore, we assume that the central daemon is starvation free and strongly fair in such a way that it selects a guard which is active for an overall infinite number of evaluation cycles infinitely often. In the system model, we implement the set of

process P_i
var S_i **init** s_i {* local state *}
{* $L_{i,1}, \ldots, L_{i,c_i}$ are communication variables *}
begin
$\text{guard}_{i,1} \rightarrow \text{action}_{i,1}$
$[\!]\ \text{guard}_{i,2} \rightarrow \text{action}_{i,2}$
$\vdots$
$[\!]\ \text{guard}_{i,m_i} \rightarrow \text{action}_{i,m_i}$
end

Fig. 2. Process P_i, $i = 1, \ldots, n$, of a distributed algorithm given by guarded commands

local states of all processes (and thereby all communication variables) through the state vector x (the superscript "T" indicates a transposed vector or matrix):

$$\mathsf{x} = [S_1, S_2, \ldots, S_n]^T \tag{1}$$

Its initial value is given by

$$\mathsf{x}(0) = [s_1, s_2, \ldots, s_n]^T \tag{2}$$

While the system is running, vector x may change its value as time proceeds. At time k, x's value is given by $\mathsf{x}(k)$ with $k \in \mathbb{N}_0$.

Assuming serial execution semantics, a particular guarded command – regardless of which process it originally belonged to – is modeled in two parts: the action of the guarded command defines a regulator R_i whereas the guard forms part of the switching function $s(\mathsf{x}(k))$. A generic switching function is

$$s(\mathsf{x}(k)) = \begin{cases} 1 & \text{if guard}_{1,1} \text{ is active at system state } \mathsf{x}(k) \text{ and selected} \\ 2 & \text{if guard}_{1,2} \text{ is active at system state } \mathsf{x}(k) \text{ and selected} \\ \vdots & \\ \sum_{i=1}^{n} m_i & \text{if guard}_{n,m_n} \text{ is active at system state } \mathsf{x}(k) \text{ and selected} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Assume that a certain $\text{guard}_{i,j}$ is associated with value l of the switching function. The two parts of a guarded command cooperate such that whenever $\text{guard}_{i,j}$ evaluates to "true" and is selected by the central daemon then the corresponding regulator R_l becomes active at that time, leading to an atomic modification of the system state which is in correspondence with the guarded command's $\text{action}_{i,j}$. On the contrary, if the guard does not evaluate to "true" then the corresponding regulator can never become active and selected in this evaluation cycle. As already stated, we assume serial execution semantics in the scope of this paper. However, the present model can easily be modified for supporting other execution semantics, like the *maximum parallel execution semantics*[1] as reported in [7].

Depending on the particular distributed algorithm, an additional regulator must be modeled and included in the system: assume a system state which does not activate any of the guards. In this case, the feedback system would block. But since a blocked feedback system cannot be handled by Ljapunov's "Second Method," we transform the system artifically into a non-blocking system by adding a special regulator R_0, called *zero regulator*, which is only triggered if no

[1] According to the *maximum parallel execution semantics*, at most one guarded command *per process* is selected at any time k. Such a behavior is achieved by using the present switching strategy but with a different switching function. The switching function uses additional regulators to implement the parallel execution of several actions within a single step.

other guard becomes active and therefore possibly selected ($\underline{\mathbf{0}}$ and $\underline{0}$ indicate a zero matrix and a zero vector of adequate dimensions):

$$\mathsf{r}^0 := \underline{\mathbf{0}} \cdot \mathsf{x}(k) + \underline{0} \tag{4}$$

Having a desired state of $\mathsf{w} = \underline{0}$ (see below), regulator R_0 does no modification to the system state, thus it will then be executed over and over again, thereby allowing virtual time k to proceed. If a particular distributed algorithm does not exhibit blocking system states (such as e.g., Dijkstra's token ring protocol for mutual exclusion with $k > n$ states [4]) then the zero regulator can be omitted. In (3), the zero regulator R_0 is selected if $s(\mathsf{x}(k))$ evaluates to zero.

The desired state w represents the system state the system is expected to converge to. Consequently, it must bear the characteristics of states specified by the predicate P. Note that the presented system model aims towards self-stabilizing algorithms with only a single such state. A more general system is beyond the scope of this paper but is not complicated: it builds upon a vector function defined on the system state vector x. Comparison between system state and desired states is then done by means of the vector function and the desired state vector which may both be different in dimension than the original state vector. But even when restricted to single desired state, it is often possible to substantially help constructing the overall proof of self-stabilization of a distributed algorithm with several legal states: for Dijkstra's token ring protocol for mutual exclusion with $k > n$ states, for example, a proof can be given whose crucial part shows that a specific legal state is always reached where the bottom process exclusively holds a token. Once in this state, it is very simple to prove that all other processes will also exclusively own a token at certain times and that the set of legal states is never voluntarily left.

When modeling the distributed algorithm in terms of the system model and applying Ljapunov's "Second Method," it must finally be assured that the resulting system exhibits a unique *equilibrium state* $\mathsf{x}_e = \mathsf{w} = \underline{0}$ This can generally be achieved through the transformation

$$\mathsf{x}_{new}(k) := \mathsf{x}_{old}(k) - \mathsf{w}_{old} \tag{5}$$

$$\mathsf{w}_{new} := \underline{0} \tag{6}$$

as well as a corresponding textual modification of the guarded commands (an example will be given in Sect. 4).

The resulting system can then be investigated with respect to system stability through a criterion of Ljapunov.

3.3 Ljapunov's Stability Criterion [12, 14]

The transformed system represents a *discrete-time, free, stationary dynamic system* $\mathsf{x}(k+1) = \mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot \mathsf{u}(k)$ where $\mathbf{A} \cdot \underline{0} + \mathbf{B} \cdot \underline{0} = \underline{0}$. Based on this fact, the following theorem can be adopted.

Theorem 1 (Uniform Asymptotical Stability in the Large [12]). *Suppose there exists a scalar function $V(\mathsf{x}(k))$ such that $V(\underline{0}) = 0$ and*

(L1) $V(\mathsf{x}(k)) > 0$ *when* $\mathsf{x}(k) \neq \underline{0}$ *and*
(L2) $\Delta V(\mathsf{x}(k)) := V(\mathsf{x}(k+1)) - V(\mathsf{x}(k)) < 0$ *when* $\mathsf{x}(k) \neq \underline{0}$ *and*
(L3) $V(\mathsf{x}(k))$ *is continuous in* $\mathsf{x}(k)$ *and*
(L4) $V(\mathsf{x}(k)) \to \infty$ *when* $\|\mathsf{x}(k)\| \to \infty$.

then the equilibrium state $\mathsf{x}_e = \underline{0}$ *is* uniformly asymptotically stable in the large *and* $V : \mathbb{R}^n \mapsto \mathbb{R}$ *with* n *being the dimension of the state space is a so-called* Ljapunov function *of the system.* □

By Ljapunov's theorem, verification of the self-stabilization property of a distributed algorithm is transformed into the identification of a Ljapunov function exhibiting the above properties. If such a function can be identified then "uniform asymptotical stability in the large" [12] is guaranteed. Beyond others, this means that the system *converges* from any point in the state space to the only equilibrium state, namely the origin. *Closure* is also guaranteed: an equilibrium state is – once reached – never left in the absence of disturbances which modify the system state. Thus, self-stabilization is proven [1].

By using Ljapunov's "Second Method," proving self-stabilization basically is transformed into constructing a Ljapunov function. Therefore on first glance, one might come to the conclusion that the problem still has not become any easier. However, the contrary is the case. Various strategies for constructing suitable Ljapunov functions are known from literature (e.g. [9]). Those strategies, sometimes dating back to the beginning of the last century, are waiting to be used. Additionally, through the more formal approach, certain necessary conditions for proving self-stabilization are derived "on-the-fly." In the next section, we will demonstrate the semi-automatic verification of a sample algorithm through our technique.

4 An Example: Stabilizing Maximum

In the following, we present and verify a self-stabilizing distributed algorithm to which we refer to as *Stabilizing Maximum* algorithm. The sample algorithm is quite simplistic but suffices for demonstrating the basic idea and application of the proposed technique. For an example of a more complex algorithm verified through this technique please refer to [18].

We assume a distributed algorithm consisting of n processes P_i, $i = 1, \ldots, n$. Every process P_i can *directly* (e.g., through reading of the corresponding communication variable) communicate with a subset of the other $n-1$ processes but not necessarily with all of them. The only requirement is that every process can somehow communicate with all other processes, i.e., for any pair P_i, P_j, process P_i can either directly communicate with P_j or transitively. Transitive communication assumes that there exists a path from $P_i P_{k_1} \cdots P_{k_q} P_j$ with $k_1, \ldots, k_q \in \{1, \ldots, n\} \setminus \{i, j\}$ and P_i can directly communicate with P_{k_1}, P_{k_q} can directly communicate with P_j and P_{k_l} can directly communicate with $P_{k_{l+1}}$ for $l = 1, \ldots, q-1$. Beyond this, no specific communication topology is assumed.

The sub-algorithm executed by each process P_i is given in Fig. 3. S_i is process P_i's local state. $L_{i,1}, \ldots, L_{i,m_i}$ denote the local states of processes with which process P_i can directly communicate. For ease of description, we call those processes *neighbors* of P_i. For every neighbor, there exists a guarded command in P_i's body: when active and selected, process P_i adjusts its own local state by copying the local state of the particular neighbor into its own local state variable.

process P_i
 var S_i **init** s_i {* local state *}
 {* $L_{i,1}, \ldots, L_{i,m_i}$ are communication variables *}
 begin
 $S_i < L_{i,1} \rightarrow S_i := L_{i,1}$
 $[\!]$ $S_i < L_{i,2} \rightarrow S_i := L_{i,2}$
 $\vdots$
 $[\!]$ $S_i < L_{i,m_i} \rightarrow S_i := L_{i,m_i}$
end

Fig. 3. Process P_i, $i = 1, \ldots, n$, of the sample algorithm

Let a *most recent initial local state* be the local state which a particular process has adopted due to the most recent failure situation and not due the execution of an action. Now, we can formulate the following theorem.

Theorem 2 (Self-Stabilization of the Stabilizing Maximum Algorithm) *The stabilizing maximum algorithm self-stabilizes to a special unison state, namely a system state in which all n processes have an identical local state and that this local state, interpreted as a natural number, is the maximum of all most recent initial local states.* □

Next, we will prepare the corresponding proof.

4.1 Matching the Algorithm to the System Model

We define the state vector x according to (1), i.e., $\mathsf{x}_i(k)$ is the local state of process P_i at time k for all $i = 1, \ldots, n$ and $k \in \mathbb{N}_0$. Thus, the state vector gives the system state. The arbitrary initial state is given according to (2). Let s_{max} be the maximum $\max\{s_1, \ldots, s_n\}$ of all most recent initial local states. $\mathbf{A}$, $\mathbf{B} \in \mathbb{M}(n \times n, \mathbb{R})$ are unity matrices.

The heart of the algorithm consists of $g = \sum_{k=1}^{n} m_k$ guarded commands. As described in Sect. 3.1, every guarded command leads to the definition of a regulator, and all guarded commands together define the switching function. Assume that regulator R_l with $l = j + \sum_{1<k<i} m_k$ implements $\text{action}_{i,j}$. According to

Fig. 1 regulator R_l is given by $\mathsf{r}^l(k) = \mathbf{C}^l \cdot \mathsf{x}(k) + \mathsf{d}^l$. Taking the corresponding guarded command into account then matrix $\mathbf{C}^l$ and vector d^l must be specified as

$$\mathbf{C}^l = (c_{u,v}) \qquad \text{with} \qquad c_{u,v} := \begin{cases} -1 & \text{if } u = i \text{ and } v = i \\ 1 & \text{if } u = i \text{ and } v = j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

and

$$\mathsf{d}^l = \underline{0} \tag{8}$$

Thus, when this particular regulator is selected at time k then the subsequent system state $\mathsf{x}(k+1)$ evaluates to

$$\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot (\mathsf{w} + \mathsf{r}^l) = \mathsf{x}(k) + \mathbf{C}^l \cdot \mathsf{x}(k) + \mathsf{d}^l = \mathsf{x}(k) + \mathbf{C}^l \cdot \mathsf{x}(k)$$

In other words, regulator R_l overwrites the local state of process P_i with P_j's local state. The switching function is given below.

$$s(\mathsf{x}(k)) = \begin{cases} 1 & \text{if } S_1 < L_{1,1} \text{ and selected} \\ 2 & \text{if } S_1 < L_{1,2} \text{ and selected} \\ \vdots & \\ \sum_{l=1}^{n} m_l & \text{if } S_n < L_{n,m_n} \text{ and selected} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Next, we have to define the desired state w. Since we expect the system to stabilize such that every local state will adopt the value s_{max}, w must be set to $[s_{max}, \ldots, s_{max}]^T$. Through the final transformation as given by (5) and (6), we obtain a system which behaves as the untransformed one but with the difference that the desired state lies in the origin of the state space. Note, that no modification of the guarded commands due to the transformation is necessary, because the boolean expressions and actions preserve their original meaning: for example "$S_i < L_{i,j}$" remains valid when subtracting a fixed number from all components of the state vector. The same holds for an action "$S_i := L_{i,j}$." From now on, in order to simplify terminology, x, w etc. always refer to the transformed system.

4.2 Proving Self-Stabilization of the Algorithm

The system obtained in the previous section allows the use of Ljapunov's stability criterion. In order to make the criterion work, we require a suitable Ljapunov function. But how can this function be identified if one lacks any idea of how this function might look like?

A widespread standard starting point for identifying a Ljapunov function in engineering domains is the following (see [9]):

$$V(\mathsf{x}(k)) = \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \tag{10}$$

with $\mathbf{P} = (p_{u,v}) \in \mathbb{M}(n \times n, \mathbb{R})$ being a time-invariant matrix. Based on the expectation that this starting point is general enough for "capturing the dynamic behavior of the system," all what is left to do, is to identify the n^2 matrix elements $p_{u,v}$ such that the constraints *(L1) – (L4)* of Theorem 1 are satisfied. Because the system is specified in a very formal manner, the identification of those matrix elements – if existent – can be assisted and sometimes automatically be solved by mathematical tools, like Matlab. Constraint *(L2)* seems to be most restrictive and is therefore evaluated first.

Proving (L2): Let R_l denote an arbitrary regulator. Then,

$$\begin{aligned}
\Delta V(\mathsf{x}(k)) &= V(\mathsf{x}(k+1)) - V(\mathsf{x}(k)) \;=\; \mathsf{x}^T(k+1) \cdot \mathbf{P} \cdot \mathsf{x}(k+1) - \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \\
&= [\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot \mathsf{u}(k)]^T \cdot \mathbf{P} \cdot [\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot \mathsf{u}(k)] - \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \\
&= \left[\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot (\mathsf{w} + \mathsf{r}^l(k))\right]^T \cdot \mathbf{P} \cdot \left[\mathbf{A} \cdot \mathsf{x}(k) + \mathbf{B} \cdot (\mathsf{w} + \mathsf{r}^l(k))\right] \\
&\quad - \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \\
&= \left[\mathsf{x}(k) + \mathsf{r}^l(k)\right]^T \cdot \mathbf{P} \cdot \left[\mathsf{x}(k) + \mathsf{r}^l(k)\right] - \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \\
&= \left[\mathsf{x}(k) + \mathbf{C}^l \mathsf{x}(k)\right]^T \cdot \mathbf{P} \cdot \left[\mathsf{x}(k) + \mathbf{C}^l \mathsf{x}(k)\right] - \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k) \\
&= \mathsf{x}^T(k) \cdot \left[(\mathbf{C}^l)^T \cdot \mathbf{P} + (\mathbf{C}^l)^T \cdot \mathbf{P} \cdot \mathbf{C}^l + \mathbf{P} \cdot \mathbf{C}^l\right] \cdot \mathsf{x}(k) \qquad (11)
\end{aligned}$$

$\Delta V(\mathsf{x}(k))$ is required to be equal to zero if $l = 0$ and less than zero if $l = 1, \ldots, g$. The former is clearly the case, since the zero regulator does not modify the system state. In order to prove the latter, the fact can be exploited that regulator R_l, $l \neq 0$, is only selected, if its corresponding guard – say $\text{guard}_{i,w}$ – calculates to "true." Thus, we know that in this case $S_i < L_{i,w}$ holds. Expressed in terms of the model this means that $\mathsf{x}_i(k) < \mathsf{x}_j(k)$ for a particular $j \neq i$.

Based on this, the following inequalities must be guaranteed by a suitable choice of $\mathbf{P}$.

$$\mathsf{x}^T(k) \cdot \left[(\mathbf{C}^l)^T \cdot \mathbf{P} + (\mathbf{C}^l)^T \cdot \mathbf{P} \cdot \mathbf{C}^l + \mathbf{P} \cdot \mathbf{C}^l\right] \cdot \mathsf{x}(k) < 0 \text{ if } \mathsf{x}_i(k) < \mathsf{x}_j(k) \qquad (12)$$

with $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$. Through continued direct evaluation, one obtains

$$\mathsf{x}_j(k) \cdot \left(\sum_{1 \leq r \leq n} \mathsf{x}_r(k) \cdot p_{j,r}\right) - \mathsf{x}_i(k) \cdot \left(\sum_{1 \leq r \leq n} \mathsf{x}_r(k) \cdot p_{i,r}\right) < 0 \text{ if } \mathsf{x}_i(k) < \mathsf{x}_j(k) \qquad (13)$$

Let $\mathsf{x}_i(k) + a_{i,j}(k) = \mathsf{x}_j(k)$ with $a_{i,j}(k) > 0$. Then, (13) rewrites to

$$(\mathsf{x}_i(k) + a_{i,j}(k)) \cdot \left(\sum_{1 \leq r \leq n} \mathsf{x}_r(k) \cdot p_{j,r}\right) - \mathsf{x}_i(k) \cdot \left(\sum_{1 \leq r \leq n} \mathsf{x}_r(k) \cdot p_{i,r}\right) < 0 \quad \text{if } \mathsf{x}_i(k) < \mathsf{x}_j(k) \qquad (14)$$

In order to further simplify the inequalities, one can either use the help of mathematical tools or try some matrices which are easy to handle: a matrix $\mathbf{P}$ of diagonal type with $p_{u,v} = 0$ for all $u \neq v$ leads to

$$\begin{aligned}(\mathsf{x}_i(k) + a_{i,j}(k)) \cdot \mathsf{x}_j(k) \cdot p_{j,j} - \mathsf{x}_i(k) \cdot \mathsf{x}_i(k) \cdot p_{i,i} = \\ (\mathsf{x}_i(k) + a_{i,j}(k))^2 \cdot p_{j,j} - \mathsf{x}_i^2(k) \cdot p_{i,i} = \\ \mathsf{x}_i^2(k)(p_{j,j} - p_{i,i}) + 2 \cdot p_{j,j} \cdot a_{i,j}(k) \cdot x_i(k) + p_{j,j} \cdot a_{i,j}^2(k) < 0 \\ \text{if } \mathsf{x}_i(k) < \mathsf{x}_j(k) \end{aligned} \tag{15}$$

Since $a_{i,j}(k) > 0$ and $\mathsf{x}_i(k) < 0$, (15) is satisfied if $p_{j,j} > 0$ and $p_{j,j} = p_{i,i}$. As a result, a possible choice of matrix $\mathbf{P}$ is

$$\mathbf{P} = (p_{i,j}) \qquad \text{with} \qquad p_{i,j} := \begin{cases} a & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

where $a \in \mathbb{R}^+$.

Proving (L1): It is easy to see that $V(\mathsf{x}(k)) = \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k)$ is zero if $\mathsf{x}(k) = \underline{0}$ and greater zero if $\mathsf{x}(k) \neq \underline{0}$. Hence, $V(\mathsf{x}(k))$ is positive definite if $\mathsf{x}(k) \neq \underline{0}$. Actually in this special case, $V(\mathsf{x}(k))$ turns out to be an instance of a generalized Euclidian norm [6]. Consequently, *(L1)* is trivially satisfied.

Proving (L3): $V(\mathsf{x}(k))$ solely consists of continuous functions of $\mathsf{x}(k)$. Thus, $V(\mathsf{x}(k))$ must be continuous in $\mathsf{x}(k)$.

Proving (L4): Finally, it must be guaranteed that if an arbitrary norm of $\mathsf{x}(k)$ approaches infinity, so does $V(\mathsf{x}(k))$. Since in our case, $V(\mathsf{x}(k))$ is itself a norm, the constraint is automatically satisfied.

Because *(L1) – (L4)* hold using $V(\mathsf{x}(k)) = \mathsf{x}^T(k) \cdot \mathbf{P} \cdot \mathsf{x}(k)$ with $\mathbf{P}$ as given by (16), Theorem 1 applies: the system is guaranteed to stabilize from any state to the equilibrium state $\mathsf{w} = \underline{0}$. This corresponds to the system state of the untransformed system in which all local states eventually evaluate to s_{max}. This proves Theorem 2. □

5 Conclusion

In this paper, we have shown that Ljapunov's century-old theory can in fact be used to elegantly prove the self-stabilization property of distributed algorithms. The traditional proof strategy as used in computer science for reasoning about self-stabilization is based on a variant function. Whether one succeeds in finding such a function or not primarily depends on the proof designer's expertise. A successful outcome is by no means guaranteed.

By using Ljapunov's "Second Method," proving self-stabilization is basically reduced to constructing a Ljapunov function. Thus, on first glance, one might come to the conclusion that the problem still has not become any easier. However,

we believe the contrary: many strategies for constructing a suitable Ljapunov function can be found in literature. Some of those strategies are dating back to the beginning of the last century. Through our technique, they could be re-activated for the benefit of program verification. Additionally, we could observe that through the more formal approach, certain necessary conditions for proving self-stabilization are derived "on-the-fly" as it is the case in the sample algorithm given in the paper. Although creative ideas of the proof designer will always ease the construction of a proof, our approach presents a formal framework for the proof designer which helps to focus creative work on crucial hot spots in a very precise setting.

We are currently refining and extending our system model in order to more easily cope with nondeterminism and fairness aspects. For a more general and semi-automatic utilization of the presented verification technique, we are working on a "multi-level layering" of Ljapunov functions. Through this layering, we hope to obtain an identification strategy for "non-scalar" Ljapunov functions. This is expected to cover the cases where a conventional scalar Ljapunov function cannot be identified for proving stability: for instance when proving the Game of Cards [2] with more than two players one is confronted with this problem. For such situations, we think of a second Ljapunov "sub"-function which still manifests convergence, leading to a general proof for all n. We hope to more formally report on this result, which has interesting analogies to lexicographically ordered variant functions, in the future. We hope that our approach helps to turn the goal of verifying self-stabilizing distributed algorithms "into a craft rather than preserving it an art."

Acknowledgments

We would like to thank the anonymous referees for their helpful comments which led to an improved presentation.

References

[1] A. Arora. *Encyclopedia of Distributed Computing*, chapter Stabilization. Kluwer Academic Publishers, 1999.

[2] J. Desel, E. Kindler, T. Vesper, and R. Walter. A simplified proof for a self-stabilizing protocol: A Game of Cards. *Information Processing Letters*, 54:327–328, 1995.

[3] E. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the Association for Computing Machinery*, 18(8):453–457, Aug. 1975.

[4] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[5] E. W. Dijkstra. A belated proof of self stabilization. *Distributed Computing*, 1:5–6, 1986.

[6] O. Föllinger. *Nichtlineare Regelungen I (in German)*. R. Oldenburg Verlag, Munich, Germany, 1998.

[7] M. G. Gouda and T. Herman. Stabilizing Unison. *IPL*, pages 171–175, August 1990.
[8] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, Apr. 1991.
[9] W. Hahn. *Stability of Motion*. Springer-Verlag, 1967.
[10] R. Isermann. *Digitale Regelsysteme, Band I (in German)*. Springer-Verlag, New York, 1988. ISBN 0-387-16596-7.
[11] R. E. Kalman and J. E. Bertram. Control System Analysis and Design Via the "Second Method" of Lyapunov - Part I: Continuous-Time Systems. *Transactions of the ASME, Journal of Basic Engineering*, 82:371–393, June 1960.
[12] R. E. Kalman and J. E. Bertram. Control System Analysis and Design Via the "Second Method" of Lyapunov - Part II: Discrete-Time Systems. *Transactions of the ASME, Journal of Basic Engineering*, 82:394–400, June 1960.
[13] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29:39–42, 1988.
[14] M. A. Ljapunov. Problème général de la stabilité du movement. *Ann. Fac. Sci. Toulouse*, 9:203–474, 1907.
[15] D. Mandrioli and C. Ghezzi. *Theoretical Foundations of Computer Science*. John Wiley and Sons, 1987.
[16] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.
[17] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.
[18] O. Theel. Verification of Dijkstra's Self-Stabilizing Token Ring Algorithm by Means of Non-Linear Control System Analysis. Technical Report TUD–BS–2000–02, Darmstadt University of Technology, Dept. of Computer Science, Institute for System Architecture, D-64283 Darmstadt, Germany, March 2000. Accepted for Publication at the Intern. Symposium on Computational Intelligence (CI'00) being part of the Intern. ICSC Congress on Intelligent Systems and Applications (ISA 2000), Wollongong, Australia, December 12-15, 2000.
[19] O. Theel and F. C. Gärtner. An Exercise in Proving Convergence through Transfer Functions. In *Proc. of the 4th Workshop on Self-Stabilizing Systems (WSS'99), being part of the 19th International Conference on Distributed Computer Systems (ICDCS'99), Austin, TX, U.S.A.*, pages 41–47. IEEE, June 1999.
[20] V. I. Utkin. Variable Structure Systems with Sliding Modes. *IEEE Transactions on Automatic Control*, 22:212–222, 1977.
[21] J. C. Willems. *The Analysis of Feedback Systems*. Number 62 in Research Monograph. The M.I.T. Press, Cambridge, MA, U.S.A., 1971.

Self-Stabilizing Local Mutual Exclusion and Daemon Refinement

Joffroy Beauquier[1], Ajoy K. Datta[2], Maria Gradinariu[1], and Frederic Magniette[1]

[1] Laboratoire de Recherche en Informatique, Université de Paris Sud, France
[2] Department of Computer Science, University of Nevada Las Vegas

Abstract. Refining self-stabilizing algorithms which use tighter scheduling constraints (weaker daemon) into corresponding algorithms for weaker or no scheduling constraints (stronger daemon), while preserving the stabilization property, is useful and challenging. Designing transformation techniques for these refinements has been the subject of serious investigations in recent years. This paper proposes a transformation technique to achieve the above task. The heart of the transformer is a self-stabilizing local mutual exclusion algorithm. The local mutual exclusion problem is to grant a process the privilege to enter the critical section if and only if none of the neighbors of the process has the privilege. The contribution of this paper is twofold. First, we present a bounded-memory self-stabilizing local mutual exclusion algorithm for arbitrary network, assuming any arbitrary daemon. After stabilization, this algorithm maintains a bound on the service time (the delay between two successive executions of the critical section by a particular process). This bound is $\frac{n \times (n-1)}{2}$ where n is the network size. Second, we use the local mutual exclusion algorithm to design two scheduler transformers which convert the algorithms working under a weaker daemon to ones which work under the distributed, arbitrary (or unfair) daemon, both transformers preserving the self-stabilizing property. The first transformer refines algorithms written under the central daemon, while the second transformer refines algorithms designed for the k-fair ($k \geq (n-1)$) daemon.
Keywords: Local mutual exclusion, self-stabilization, transformer, unfair daemon.

1 Introduction

One of the most inclusive approaches to fault-tolerance in distributed systems is *self-stabilization* [Dij74,Dol00]. Introduced by Dijkstra [Dij74], this technique guarantees that, regardless of the initial state, the system will eventually converge to the intended behavior. The correctness of self-stabilizing algorithms is proven assuming some type of scheduler (or daemon) as the adversary. The two most common schedulers are the following: the central scheduler—only one process can execute an atomic step at one time—and the distributed scheduler—any nonempty subset of the enabled processes can execute their atomic steps simultaneously. Although it is easier to prove the stabilization for the algorithms

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 223–237, 2000.

working under the central scheduler, but those working under the distributed scheduler supports more practical implementations. So, it makes sense to design self-stabilizing algorithms under central scheduler, prove its correctness in this model, and then be able to transform the algorithms to their corresponding algorithms under the distributed scheduler preserving the self-stabilization and other desirable properties. One of the main goals of this paper is to design such a transformer. The dining philosophers problem [Dij71] deals with the mutual exclusion among neighboring processes in a ring. The local mutual exclusion problem is the extension of this problem to any arbitrary network. We propose a bounded memory and bounded service time local mutual exclusion algorithm. Then we demonstrate the application of this local mutual exclusion algorithm to design a transformer to transform algorithms working under a weaker daemon to ones which work under the distributed arbitrary daemon, preserving the self-stabilization property.

Related Work. There have been several attempts to develop transformers that transform a program written and proven under the assumption of a weak daemon to a program self-stabilizing under a strong daemon. A special class of systems, called alternator, was introduced in [GH97] for the linear topology and in [GH99] for any arbitrary topology. The idea of an alternator is the following: Every process has an integer state variable which is bounded by $2d-1$, where d is the length of the longest simple cycle in the network. One of the main features of the alternator is that no two enabled neighboring processes can have the state variable equal to $2d-1$ at the same time. The processes do some effective work only when they are in state $2d-1$. This non-interference property is used in [GH99] to design a transformer from the central daemon to the distributed daemon. The transformation idea is to compose the algorithm A (written under the central daemon) with the alternator such that the actions of the algorithm A are executed only when the alternator is in state $2d-1$. Another transformer was proposed in [MN97]. This method uses timestamps to order the actions of any self-stabilizing algorithm. One notable feature of [MN97] is that it achieves the silent stabilization [DGS96], but the algorithm used unbounded variables.

Another approach in designing transformation techniques is to implement the local mutual exclusion among the neighboring processes. Distributed, but non-stabilizing solutions to the local mutual exclusion problem are presented in [CM84] and [AS90]. The alternator [GH99] can also be considered as a solution to the dining philosophers problem. But, the method in [GH99] does not solve the local mutual exclusion problem for the following reason: Only when one and exactly one among the neighboring processes is in state $2d-1$, the process can enter the critical section. But, in this algorithm, there are many configurations where none of the neighboring processes is in state $2d-1$, meaning, none of them is in critical section. The algorithms in [HP89] and [Gou87] propose self-stabilizing solutions to the dining philosophers problem (and hence, to the local mutual exclusion problem). But, both solutions use a central daemon and a distinguished process to implement the token circulation. The process holding the token executes its critical section. Another self-stabilizing solution to the din-

ing philosophers problem has been proposed recently in [Hua00]. The algorithm in [Hua00] works under the read/write atomicity model [DIM93], but makes a very strong assumption—the links of the network are (initially) colored in a special way. A local mutual exclusion algorithm for tree networks is presented in [JADT99]. Another solution on trees is proposed in [AS99]. This solution uses bounded memory and the read/write atomicity model. But, the proposed algorithm does not satisfy the local mutual exclusion property during a short time when the variables are wrapped around to maintain their bounded nature.

Recently, a transformer using the local mutual exclusion has been reported in [AN99]. Their method is focused on the refinement of atomicity—from high to low, and works for the finest atomicity grain, i.e, read/write atomicity [DIM93]. The solution also uses bounded variables. But, since the algorithm uses a weakly fair daemon, although the service time is bounded, the exact bound on the service time cannot be computed (since it depends on the type of daemon).

Our Contributions We first present two solutions to the local mutual exclusion problem. The first solution uses unbounded memory. We then extend the first algorithm to design a bounded memory solution. Both algorithms work in the read/write atomicity model [DIM93]. So, our algorithms use the same model as in [AN99], but, unlike their solution, ours is self-stabilizing under any arbitrary (unfair) distributed daemon. After stabilization, the service time is bounded by $\frac{n(n-1)}{2}$.

Then we use the local mutual exclusion algorithm to design two stabilizing preserving transformers to transform algorithms written using weaker daemon into algorithms which work under the assumption of a stronger daemon. The first transformer refines algorithms written under the central daemon, while the second transformer takes as input algorithms designed for the k-fair ($k \geq (n-1)$) daemon. Both transformers convert the input algorithms to self-stabilizing algorithms which work under the assumption of arbitrary (even unfair) distributed daemon.

Outline of Paper. The rest of the paper is organized as follows: The model and specification of the problem solved in this paper is presented in Section 2. In Section 3, two local mutual exclusion algorithms are given. The two transformers are discussed in Section 4. Section 5 provides some concluding remarks.

2 Model and Specification

A distributed system is a set of state machines called processes. Each process can communicate with a subset of the processes called neighbors. We will use $\mathcal{N}.x$ to denote the set of neighbors of node x, and $|\mathcal{N}.x|$ to represent the number of neighbors of x. The communication among neighboring processes is carried out using the communication registers (called "shared variables" throughout this paper). We consider distributed systems consisting of n processes where every process has a unique identifier. The system's communication graph is drawn

by representing processes as nodes and the neighborhood relationship by edges between the nodes.

Any process in a distributed system executes an algorithm which contains a finite set of guarded actions of the form: $\langle label \rangle :: \langle guard \rangle \longrightarrow \langle statement \rangle$, where each guard is a boolean expression over the local and shared variables.

A *configuration* of a distributed system is an instance of the state of the system processes. A process is *enabled* in a given configuration if at least one of the guards of its algorithm is *true*. We denote the set of enabled processes for a given configuration by E.

A distributed system can be modeled by a transition system. A transition system is a three-tuple $S = (\mathcal{C}, \mathcal{T}, \mathcal{I})$ where $\mathcal{C}$ is the collection of all the configurations, $\mathcal{I}$ is a subset of $\mathcal{C}$ called the set of initial configurations, and $\mathcal{T}$ is a function $\mathcal{T} : \mathcal{C} \longrightarrow \mathcal{C}$. A transition, also called a *computation step*, is a tuple (c_1, c_2) such that $c_2 = \mathcal{T}(c_1)$. A *computation* of an algorithm $\mathcal{P}$ is a *maximal* sequence of computations steps $e = ((c_0, c_1)\ (c_1, c_2)\ \ldots (c_i, c_{i+1}) \ldots)$ such that for $i \geq 0, c_{i+1} = \mathcal{T}(c_i)$ (a single *computation step*) if c_{i+1} exists, or c_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no process is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. A *fragment* of a computation e is a finite sequence of successive computation steps of e.

In a computation, a transition (c_i, c_{i+1}) occurs due to the execution of a nonempty subset of the enabled processes in configuration c_i. We assume the read/write atomicity model [DIM93] with the semantics of [AN99]: In a computation step, a process either reads one of its neighbors' state, or writes its local state, but not both. In every computation step, this subset is chosen by the scheduler or daemon. We refer to the following types of daemon in this paper: *central daemon* — in every computation step, only one of the enabled processes is chosen by the daemon; *k-fair daemon* — a process cannot be selected more than k times by the daemon without choosing another process which has been continuously enabled; *weakly fair daemon* — if a process p is continuously enabled, p will be eventually chosen by the daemon to execute an action; *distributed daemon* — during a computation step, any nonempty subset of the enabled processes is chosen by the daemon.

We refer to the distributed unfair daemon as the *stronger daemon* and all other daemons (defined above) as the *weaker daemons*.

Self-Stabilization. In order to define self-stabilization for a distributed system, we use two types of predicates: the legitimacy predicate—defined on the system configurations and denoted by $\mathcal{L}$—and the problem specification—defined on the system computations and denoted by $\mathcal{SP}$.

Let $\mathcal{P}$ be an algorithm. The set of all computations of the algorithm $\mathcal{P}$ is denoted by $\mathcal{E}_{\mathcal{P}}$. Let $\mathcal{X}$ be a set and $Pred$ be a predicate defined on the set $\mathcal{X}$. The notation $x \vdash Pred$ means that the element x of $\mathcal{X}$ satisfies the predicate $Pred$ defined on the set $\mathcal{X}$.

Definition 1 (Self-Stabilization). *An algorithm $\mathcal{P}$ is self-stabilizing for a specification $\mathcal{SP}$ if and only if the following two properties hold:*
(1) convergence — all computations reach a configuration that satisfies the legitimacy predicate. Formally, $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)(c_1, c_2)\ldots) : \exists n \geq 1, c_n \vdash \mathcal{C}$*;*
(2) correctness — all computations starting in configurations satisfying the legitimacy predicate satisfy the problem specification $\mathcal{SP}$. Formally, $\forall e \in \mathcal{E}_{\mathcal{P}} :: e = ((c_0, c_1)\ (c_1, c_2)\ \ldots) : c_0 \vdash \mathcal{L} \Rightarrow e \vdash \mathcal{SP}$.

Local Mutual Exclusion. The specification of the local mutual exclusion problem ($\mathcal{SP}_{\mathcal{L}}$) is the conjunction of two predicates—safety and liveness—defined in terms of the "privilege to enter the critical section": *safety predicate* — in any configuration, there exists at least one privileged process, and if a process holds a privilege, then none of its neighbors holds the privilege; *liveness predicate* — every process holds the privilege infinitely often.

Definition 2 (Fairness Index). *Let $\mathcal{P}$ be a self-stabilizing mutual exclusion algorithm. $\mathcal{P}$ is considered to have a fairness index of k if in any computation of $\mathcal{P}$ under the assumption of any daemon, between any two consecutive critical section executions of a process, any other process can execute its critical section at most k times.*

Let $\mathcal{P}$ be a self-stabilizing local mutual exclusion algorithm. The *service time* of $\mathcal{P}$ is the maximum number of critical sections executed by other processors between two successive executions of the critical section by any process without any assumption of the daemon.

Virtual Orientation of the Communication Graph. We will use an "adjacency" relation, denoted by $\triangleright$, over the shared variables of the processes to define a virtual orientation of the communication graph. The exact definition (or implementation) of this relation will be specific to the two solutions to the local mutual exclusion problem presented in Section 3.

Definition 3 (Virtual Orientation). *Let $x.p$ and $x.q$ be two shared variables of two neighboring processes p and q. In the communication graph, the edge between p and q is said to be virtually oriented from p to q if and only if $x.p \triangleright x.q$. This edge is an incoming edge for q and an outgoing edge for p.*

Definition 4 (Privileged Process). *A process is said to be privileged in a configuration if in the communication graph, all the edges adjacent to the process are oriented towards it (i.e., incoming edges).*

3 Local Mutual Exclusion

We present two solutions to the local mutual exclusion problem in this section. We first present the unbounded space solution because this would help understand the ideas behind the second solution. Next, we will give the bounded

solution which is our final solution. In the next section, we will use the bounded solution to design two scheduler transformers. Both solutions use the edge reversal mechanism of [BG89] to maintain the acyclic orientation of the communication graph.

3.1 Unbounded Local Mutual Exclusion Algorithm

In this subsection, we propose a self-stabilizing local mutual exclusion algorithm. The key feature of this algorithm is its $(n-1)$-fairness index under any arbitrary distributed daemon.

Constants:
$id.p$: unique integer identifier of p;
$\mathcal{N}.p$: the set of neighbors of process p;
Shared Variable:
$L.p$: unbounded integer;
Local variable:
$L\_copy[|\mathcal{N}.p|]$: array of unbounded integers containing the copy of L of neighbors;
CS : boolean flag used to indicate if a process is in the critical section or not;
Function:
$(\forall q \in \mathcal{N}.p) : p \prec q \equiv L\_copy[q] \rhd L.p$

Actions:
$\mathcal{A}_1 : \forall q \in \mathcal{N}.p, \quad p \prec q \longrightarrow$
$CS = 1;$
execute critical section;
$L.p = max\{L\_copy[q] |\ q \in \mathcal{N}.p\} + 1;$

$\mathcal{CP}_1 : \exists q \in \mathcal{N}.p, \quad q \prec p \wedge L\_copy[q] \neq L.q \longrightarrow$
CS=0;
$L\_copy[q] = L.q;$

Algorithm 3.1: Unbounded Local Mutual Exclusion (ULME) for process p

Algorithm ULME We borrow the definition of the "direction of an edge" from [GK93] to define the adjacency relation ($\rhd$) for Algorithm ULME (shown as Algorithm 3.1).

Definition 5. *For any two neighboring processes p_1 and p_2 executing Algorithm ULME, $L.p_2 \rhd L.p_1$ iff $(L.p_1 < L.p_2) \vee ((L.p_1 = L.p_2) \wedge (id.p_1 < id.p_2))$. We refer to this situation as "p_1 is virtually oriented towards p_2".*

Remark 1. Due to the uniqueness of the process ids, $\rhd$ is a total order (anti-reflexive, anti-symmetric, and transitive) relation.

The virtual orientation is used in Algorithm ULME in the following manner: A process enters its *critical section* if and only if it is privileged (Definition 4), i.e., all its incident edges are oriented towards it. Once the process finishes executing its critical section, it reverses all its incident edges.

Every process p maintains a value from a totally ordered set that records locally a copy of (what it thinks is) the value of each of its neighbors. If p thinks that its own value is the local minimum, then it becomes privileged. After using the privilege, p sets its value to the maximum of the locally recorded neighbors' values plus one (Action $\mathcal{A}_1$). Otherwise, if p thinks that there is a neighbor q (of p) that has a value less than p's, p checks if its locally recorded value is correct. If not, p updates its local record (Action $\mathcal{CP}_1$).

Note 1. The variable CS in Algorithm ULME is not necessary to solve the local mutual exclusion problem. We added this in the code in Algorithm 3.1 because we would need this to design the transformers in Section 4.

Correctness of Algorithm ULME We now give the steps of the proof, providing bounds for the service time and fairness.

Definition 6 (Legitimate Configuration). *A* legitimate configuration *for Algorithm ULME (i.e., a configuration which satisfies the legitimacy predicate $\mathcal{L}_{ULME}$) is a configuration such that: (i) At least one process is privileged and (ii) no two neighbors are privileged.*

Lemma 1. *Let G be the communication graph representing the system executing Algorithm ULME. The graph G is acyclic.*

Lemma 2 (Convergence). *Every computation of Algorithm ULME reaches a configuration satisfying the predicate $\mathcal{L}_{ULME}$.*

Lemma 3. *In any computation of Algorithm ULME, between every two successive instances of a process being privileged, all its neighbors are privileged.*

Lemma 4 (Liveness). *Every process is privileged infinitely often in every computation of Algorithm ULME.*

Lemma 5 (Fairness index). *The fairness index of Algorithm ULME is* $(n-1)$.

Theorem 1. *Algorithm ULME is a $(n-1)$ fairness index, self-stabilizing local mutual exclusion algorithm under any unfair daemon.*

Lemma 6 (Service Time). *The delay between two successive executions of the critical section by a particular process in Algorithm ULME is bounded by $\frac{n(n-1)}{2}$.*

3.2 Refinement to bounded memory

The main drawback of Algorithm ULME is its unbounded memory requirement. In the following, we propose a bounded solution. The new algorithm is self-stabilizing and has the fairness index of $(n-1)$ under any arbitrary daemon.

Algorithm BLME Algorithm BLME is a refined version of Algorithm ULME with bounded variables. This transformation deals with two major related problems — the maintenance of the acyclic orientation of the communication graph and the choice of a bound for the variable L. Once the acyclic orientation is provided, Algorithm BLME (as shown in Algorithm 3.2) is quite similar to Algorithm ULME.

In the following, we redefine the adjacency relation $\rhd$ to deal with the bounded integers. We then provide a bound for the variable L.

Definition 7 (Cyclic Comparison). *Let x and y be two integers bounded by a positive integer $B \geq 2$. We define the relation $\rhd$ as follows:*

- $\forall x \in [0, \frac{B}{2}]$*:*
  1. $y \rhd x$ *iff* $y \in [x+1, x+\frac{B}{2}]$.
  2. $x \rhd y$ *iff* $y \in [\frac{B}{2}+x+1, B-1] \cup [0, x-1]$.
- $\forall x \in [\frac{B}{2}+1, B-1]$*:*
  1. $y \rhd x$ *iff* $y \in [x+1, B-1] \cup [0, x-\frac{B}{2}]$.
  2. $x \rhd y$ *iff* $y \in [x-\frac{B}{2}+1, x-1]$.

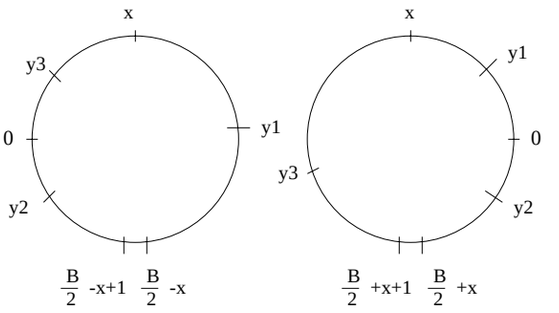


Fig. 1. Cyclic Comparison.

Figure 1 shows the (cyclic) relation between x, y_1, y_2, and y_3. The example on the left in Figure 1 shows the situation $y_1 \rhd x$, $x \rhd y_2$, and $x \rhd y_3$, and the right example indicates that $y_1 \rhd x$, $y_2 \rhd x$, and $x \rhd y_3$.

In the following, based on the relation defined in Definition 7, we redefine the virtual orientation of the communication graph using the variable L bounded by B. This orientation must provide the acyclic nature to the communication graph. We choose the value of B as follows: In order to ensure the acyclic orientation, the values of L of the neighboring processes must be different. In a completely connected graph, every node has $(n-1)$ neighbors. Therefore, the lower bound for the distance between the values of L of two neighboring processes is n. In order to avoid formation of a cycle among the nodes of the communication graph, the sum of n gaps between n processes which form the cycle must be less then

B. The minimum value of B which satisfies the above condition is n^2 when n is even and n^2+1 when n is odd. We give a more formal argument for the above explanation in Lemma 7.

Note that the cyclic comparison is an order when all the compared values are in an interval bounded by n.

Definition 8 (Bounded Virtual Orientation). *Let p_1 and p_2 be two neighboring processes executing Algorithm BLME. p_1 is virtually oriented towards p_2 if they satisfy the properties specified in Definitions 3 and 7 (defined over the variable L for $B = n^2$ if n is even and $B = n^2+1$ if n is odd).*

Note that the variable L of each process executing Algorithm BLME has values in $[0..B-1]$ where B is the constant defined in Definition 8.

As in Algorithm ULME, Algorithm BLME also maintains an acyclic communication graph in all legitimate configurations. To achieve this characteristic of the underlying graph, we force the processes to satisfy the following property, called *balance*. (We will show in Section 3.2 how the balanced processes guarantee the communication graph to be acyclic.)

Definition 9 (Balanced processes). *Two neighboring processes, p_1 and p_2, running Algorithm BLME are said to be* balanced with respect to each other (or p_1 and p_2 are balanced, in short) if and only if $|L.p_1 - L.p_2| < n \wedge ((L.p_1 \neq L.p_2) \vee (L.p_1 = L.p_2 = 0 \wedge id.p_1 < id.p_2))$. When all processes are balanced with respect to all their neighbors, we refer to this situation as a *balanced configuration. We use the notation $p_1 \sim p_2$ to indicate that p_1 and p_2 are two neighboring balanced processes.*
Let p_1 and p_2 be two neighboring processes which are not balanced. We call these processes unbalanced *and denote this condition as $p_1 \not\sim p_2$. If at least two processes are unbalanced, we say that the system is in an* unbalanced configuration.

A process p is enabled to execute its critical section if it is balanced (Definition 9) with its neighbors and if all the edges adjacent to p are oriented towards p (Action $\mathcal{A}_1$). After p exits its critical section, p reverses its incident edges. This allows the neighbors of p to get a chance to execute their critical section. In a self-stabilizing setting, the system may start in an unbalanced configuration (Definition 9). Starting from this unbalanced configuration, Algorithm BLME will eventually take the system into a balanced configuration, and the communication graph becomes acyclic again. From this configuration onwards, Algorithm BLME always moves from one balanced configuration to another, and the communication graph will remain acyclic.

When a process p recognizes that (i) it is unbalanced with respect to at least one of its neighbors, or (ii) it has a neighbor i such that $R_i = 1$ (which implies that some processes are unbalanced), p executes Action $\mathcal{R}_1$ and sets its reset marker R_p to 1. p then waits until all its neighbors also reset their R to 1. At that time, p changes its L variable to 0 (Action $\mathcal{R}_2$). Again, p waits until all its neighbors reset their L to 0. When that happens, p is balanced again with all

Constants:
B : n^2 if n is odd, n^2+1 if n is even.
$\mathcal{N}.p$: the set of neighbors of process p;
Shared Variables:
$L.p \in [0..B-1]$;
$R.p$: boolean; reset flag
Local Variables:
$L\_copy[|\mathcal{N}.p|]$: array of unbounded integers containing the copy of L of neighbors;
$R\_copy[|\mathcal{N}.p|]$: array of boolean containing the copy of R of neighbors;
CS : boolean flag used to indicate if a process is in the critical section or not;
Functions:
$MaxL(p) \equiv L.i$ such that $|L\_copy[i] - L.p| = max(|L\_copy[j] - L.p|, \forall j \in \mathcal{N}.p)$;
$(\forall i \in \mathcal{N}.p) : p \prec i \equiv (L.p \lhd L\_copy[i]) \vee ((L.p = L\_copy[i] = 0) \wedge (id.p < id.i))$;
$(\forall i \in \mathcal{N}.p) : p \sim i \equiv |L.p - L\_copy[i]| < n \wedge (L.p \neq L\_copy[i]) \vee (L.p = L\_copy[i] = 0 \wedge id.p < id.i)$;
$(\forall i \in \mathcal{N}.p) : p \not\sim i \equiv |L.p - L\_copy[i]| \geq n \vee ((L.p = L\_copy[i]) \wedge (L.p \neq 0))$;

Actions:

$\mathcal{A}_1$: $(R.p = 0) \wedge (\forall i \in \mathcal{N}.p,\ p \prec i \wedge p \sim i \wedge R\_copy[i] = 0) \longrightarrow$
CS=1;
execute critical section;
$L.p = MaxL(p) + 1$;

$\mathcal{R}_1$: $(R.p = 0) \wedge (\exists i \in \mathcal{N}.p,\ p \not\sim i \vee R\_copy[i] = 1 \wedge (L\_copy[i] \neq 0 \vee L.p \neq 0)) \longrightarrow$
CS=0;
R.p=1;

$\mathcal{R}_2$: $(R.p = 1) \wedge (L.p \neq 0) \wedge (\forall i \in \mathcal{N}.p,\ R\_copy[i] = 1) \longrightarrow$
CS=0;
$L.p = 0$;

$\mathcal{R}_3$: $(R.p = 1) \wedge (L.p = 0)\ \wedge (\forall i \in \mathcal{N}.p,\ L\_copy[i] = 0) \longrightarrow$
CS=0;
R.p=0;

$\mathcal{CP}_1$: $(R.p = 0) \wedge (\forall j \in \mathcal{N}.p,\ p \sim j \wedge R\_copy[j] = 0) \wedge (\exists i \in \mathcal{N}.p,\ i \prec p \wedge L\_copy[i] \neq L.i) \longrightarrow$
CS=0;
L_copy[i]=L.i;

$\mathcal{CP}_2$: $(R.p = 1) \wedge (L.p \neq 0) \wedge (\exists i \in \mathcal{N}.p,\ R\_copy[i] = 0 \wedge R.i = 1) \longrightarrow$
CS=0;
R_copy[i]=R.i;

$\mathcal{CP}_3$: $(R.p = 1) \wedge (L.p = 0) \wedge (\exists i \in \mathcal{N}.p,\ L\_copy[i] \neq 0 \wedge L.i = 0) \longrightarrow$
CS=0;
L_copy[i]=L.i;

Algorithm 3.2: Bounded Local Mutual Exclusion (BLME) for process p

its neighbors. p now clears R to 0 so that it can eventually execute its critical section (Action $\mathcal{R}_3$).

The role of Actions $\mathcal{CP}_i$, $(i = 1 \ldots 3)$ is to keep the local copies of L and R (i.e., the variables *L_copy* and *R_copy*) up-to-date. These actions are executed every time the value of the local variables changes.

Note 2. The variable CS in Algorithm BLME is not necessary to solve the local mutual exclusion problem. We added this in the code in Algorithm 3.2 because we would need this to design the transformers in Section 4.

Correctness of Algorithm BLME First, we derive a property of the communication graph G, representing the system running Algorithm BLME, from the unbounded virtual orientation (as per Definition 5) and the "balanced" property (Definition 9). This influences the definition of the legitimate configuration (defined in Definition 10). To prove the correctness of Algorithm BLME, we first show that starting from a legitimate configuration, any computation always maintains the legitimacy—the processes execute only Action $\mathcal{A}_1$. Finally, we prove the convergence by defining different possible scenarios in terms of edges of G and using an "edge migration" scheme.

Lemma 7. *Let G be the communication graph representing the system running Algorithm BLME. If the system is balanced, then G is acyclic.*

Property 1. If the system is balanced, then there exists at least one privileged process and no two neighbors are privileged.

Definition 10 (Legitimate Configuration). *A legitimate configuration for Algorithm BLME (i.e.,a configuration which satisfies the legitimacy predicate $\mathcal{L}_{BLME}$) is a configuration such that the following two conditions hold: (i) The system is in a balanced configuration. (ii) The value of the reset marker R of all processes is* 0.

Remark 2. In a legitimate state in Algorithm BLME, the processes execute only Action $\mathcal{A}_1$.

Lemma 8. *Let e be a computation of Algorithm BLME starting from a legitimate configuration c (i.e., c satisfies $\mathcal{L}_{BLME}$). Then any configuration reachable from c in e also satisfies $\mathcal{L}_{BLME}$.*

Now, we want to prove the convergence of Algorithm BLME. We first need to define some covering edge sets of the communication graph G representing the system running Algorithm BLME: *(1)* $M_1 = \{(p,q) \mid p$, and q are balanced and $(R.p = 0 \wedge R.q = 0)\}$; *(2)* $M_2 = \{(p,q) \mid p$, and q are balanced, and $(R.p = 1 \vee R.q = 1)\}$; *(3)* $M_3 = \{(p,q) \mid p$, and q are unbalanced, and $(R.p = 0 \vee R.q = 0)\}$; *(4)* $M_4 = \{(p,q) \mid p$, and q are unbalanced and $(R.p = 1 \wedge R.q = 1)\}$; *(5)* $MT = \bigcup_{i=1}^{4} M_i$; *(6)* $\forall i \in \{2,3,4\} : M_i^0 = \{(p,q) \mid (p,q) \in M_i \wedge (L.p = 0 \vee L.q = 0)\}$; *(7)* $M^{(0,0)} = \{(p,q) \mid L.p = 0 \wedge L.q = 0\}$.

In a legitimate configuration, all the edges are in M_1, i.e., $MT = M_1$, and the only action enabled at any process is $\mathcal{A}_1$ (Remark 2). When the system is in an illegitimate configuration, $MT \neq M_1$, and from the algorithm, at least one of Actions $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ is enabled at some process. So, our obligation now is to show that starting from such a configuration where $MT \neq M_1$, eventually, all edges will become part of M_1 again. We explain this process by using an "edge migration" process.

We will first show that starting from an illegitimate configuration, eventually, one of the processes would be able to execute one of Actions $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$, meaning that the process of convergence would eventually start. We then prove the convergence process using the edge migration process as follows (shown in Figure 2): *(a)* Every edge of M_i eventually becomes a member of $M_i^0 (i \neq 0)$ or $M^{(0,0)}$; *(b)* Every edge of $M_i^0 (i \neq 1)$ eventually moves to $M^{(0,0)}$; *(c)* Every edge of $M^{(0,0)}$ eventually moves to set M_{1r} which is defined as follows: M_{1r} contains all edges (p, q) such that p and q are balanced, $R.p = 0$, $R.q = 0$, the edge (p, q) was originally in a set $M_i (i \neq 1)$, and $(L.p \neq 0 \vee L.q \neq 0)$.
Note that M_{1r} is actually the same as M_1 except that the set M_{1r} is created only due to a computation starting from an illegitimate state. So, once the system is back to a legitimate configuration, M_{1r} becomes the current M_1.

Lemma 9. *Let e be a computation of Algorithm BLME starting from an illegitimate configuration c, i.e., where $MT \neq M_1$. Then eventually one of the actions in $\{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$ will be executed in e.*

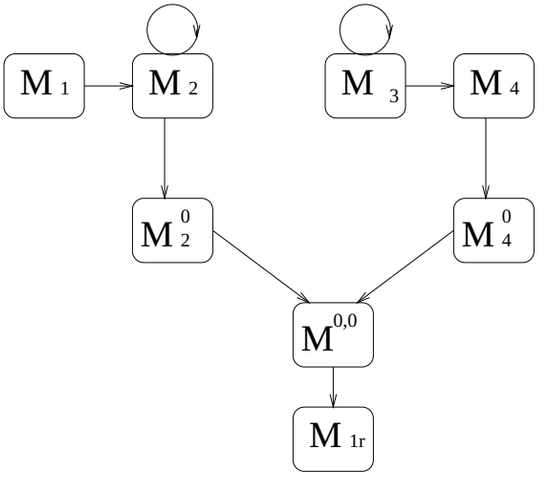


Fig. 2. Edge Migration

Lemma 10. *Let e be a computation of Algorithm BLME starting in a configuration c where $MT \neq M_1$. Then eventually, all edges of G will belong to M_{1r}.*

Remark 3. The liveness (Lemma 4) and $(n-1)$-fairness (Lemma 5) results proven for Algorithm ULME also hold for Algorithm BLME because the orientation of edges is changed in an identical manner in the two algorithms.

Theorem 2. *Algorithm BLME is a self-stabilizing local mutual exclusion algorithm under an unfair daemon having the fairness index $(n-1)$ and the service time $\frac{n\times(n-1)}{2}$.*

4 Daemon Refinement

In this section, we propose an application of the local mutual exclusion algorithms introduced in the previous section. In this section, we denote the algorithms presented in the previous section as Algorithm $\mathcal{A}_{LME}$. We use those algorithms to transform self-stabilizing algorithms proven under some weaker daemon (e.g., the central or k-fair daemon) called weaker algorithms to self-stabilizing algorithms which would work in the presence of the stronger daemon (i.e., the distributed unfair daemon). From now on, we refer to these algorithms as the stronger algorithms.

Transformation Description The transformation technique is based on a particular composition scheme between the actions of the local mutual exclusion algorithm ($\mathcal{A}_{LME}$) and a weaker algorithm ($\mathcal{W}$). Assume that $\mathcal{A}_{LME}$ and ($\mathcal{W}$) algorithms has m and n actions, respectively. Let $a_{lme}g_i$ (respectively, wg_i) and $a_{lme}s_i$ (respectively, ws_i) represent the guard and statement, respectively, of ith action of $\mathcal{A}_{LME}$ (respectively, $\mathcal{W}$) algorithm. The composed algorithm, $\mathcal{S}$, consists of the following actions:

- $\forall i \in [1..m], \forall j \in [1..n]$: $< a_{lme}g_i > \wedge < wg_j > \longrightarrow < a_{lme}s_i >$; if $CS = 1$ then $< ws_j >$
- $\forall i \in [1..m]$: $< a_{lme}g_i > \wedge \neg < wg_1 > \wedge \ldots \wedge \neg < wg_n > \longrightarrow < a_{lme}s_i >$

Note 3. The actions of the weaker algorithm are executed only when $CS = 1$, i.e., when the process is allowed to execute its critical section. The variable CS is used only to design the transformers (see Notes 1 and 2).

Lemma 11. *The composed algorithm is a self-stabilizing local mutual exclusion algorithm according to the specification $\mathcal{SP}_{\mathcal{L}}$ and its fairness index is $(n-1)$.*

4.1 Central Daemon to Distributed Daemon

In this section, we show that we can transform a self-stabilizing algorithm working under a central daemon into a self-stabilizing algorithm under a distributed daemon.

Assume that, in the composition described in Section 4, the weaker algorithm, $\mathcal{W}_C$, is a self-stabilizing algorithm for the specification $\mathcal{SP}_{\mathcal{C}}$ which works under a central daemon, and $\mathcal{S}$ represents the composed algorithm.

Theorem 3. *Algorithm $\mathcal{S}$ is self-stabilizing for the specification $\mathcal{SP}_{\mathcal{C}}$ under a distributed daemon.*

4.2 Fair Daemon to Distributed Daemon

We propose another application for the local mutual exclusion: the transformation of a self-stabilizing algorithm under a k-fair daemon into a self-stabilizing algorithm under a distributed daemon.

Assume that, in the composition described in Section 4, the weaker algorithm, $\mathcal{W}_{kB}$ is a self-stabilizing algorithm for the specification $\mathcal{SP}_{\mathcal{F}}$ under a k-fair daemon ($\forall k \geq (n-1)$), and $\mathcal{S}$ is the composed algorithm.

Theorem 4. *Algorithm $\mathcal{S}$ is self-stabilizing for the specification $\mathcal{SP}_{\mathcal{F}}$ under any distributed daemon.*

5 Conclusions

We presented a transformation technique to transform self-stabilizing algorithms under weak daemons into algorithms which maintain the self-stabilization property and also work under the stronger daemon, like any arbitrary distributed daemon (including the unfair daemon). The key tool in designing the above is a self-stabilizing local mutual exclusion algorithm, which by itself is a major contribution of this work. One of the two local mutual exclusion algorithms presented in this paper is a bounded memory self-stabilizing solution which is proven under the assumption of an unfair daemon in the read/write model [DIM93] with the semantics definition of [AN99]. Another nice feature of our local mutual algorithms is that they achieve a bounded service time.

Since our protocols work under the read/write atomicity model, they can easily be extended to the message-passing environment.

Another possible extension for our bounded solution is a generalization of the unison problem defined in [CFG92]. In Algorithm BLME, the difference between the values of the local variables of any two neighboring processes is bounded. Therefore, each process can start a phase with the number equal to the value of its local variable L. The main features of this extension are that the phase difference between two processes is bounded, and no two neighboring processes are in the same phase.

Acknowledgment The authors would like to thank the referees whose comments helped improve the presentation of the paper.

References

[AN99] A. Arora and M. Nesterenko. Stabilization-preserving atomicity refinement. *DISC'99*, pages 254–268, 1999.

[AS90] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. *31st Annual Symposium on Foundations of Computer Science*, volume I:65–74, october 1990.

[AS99] GH. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-par99, Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.

[BG89] V. Barbosa and E. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *Transactions on Programming Languages and Systems Vol 11 Num 4*, pages 562–584, 1989.

[CFG92] J.M. Couvreur, N. Francez, and M.G. Gouda. Asynchronous unison. *ICDCS92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.

[CM84] M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transations on Programming Languages and Systems*, pages 6(4):632–646, october 1984.

[DGS96] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, pages 115–138, 1971.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *ACM 17*, pages 643–644, 1974.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilizing of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[Dol00] S. Dolev. *Self-Stabilization*. The MIT Press, 2000.

[GH97] M. Gouda and F. Hadix. The linear alternator. *Proceedings of the third workshop on self-stabilizing systems (WSS-97), International Informatics Series 7, Carleton University Press*, pages 31–47, 1997.

[GH99] M. Gouda and F. Hadix. The alternator. *In Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53, 1999.

[GK93] S. Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, pages 7:55–59, 1993.

[Gou87] M. G. Gouda. The stabilizing philosopher: asymmetry by memory and by action. *Tech Rep TR-87-12, Univesity of Texas at Austin*, 1987.

[HP89] D. Hoover and J. Poole. A distributed self-stabilizing solution for the dining philosophers problem. *Information Processing Letter 41*, pages 209–213, 1989.

[Hua00] S.T. Huang. The fuzzy philoshophers. In *Workshop on Advances of Paralleland Distributed Computational Models*, page to appear, May 2000.

[JADT99] C. Johnen, L. O. Alima, A. K. Datta, and S. Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems (ICDCS'99)*, pages 487–494, june 1999.

[MN97] M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letter 66*, pages 285–290, 1997.

More Lower Bounds for Weak Sense of Direction: The Case of Regular Graphs

Paolo Boldi* and Sebastiano Vigna†

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy

Abstract A graph G with n vertices and maximum degree Δ_G cannot be given weak sense of direction using less than Δ_G colours. It is known that n colours are always sufficient, and it was conjectured that just $\Delta_G + 1$ are really needed, that is, one more colour is sufficient. Nonetheless, it has just been shown [2] that for sufficiently large n there are graphs requiring $\omega(n/\log n)$ more colours than Δ_G. In this paper, using recent results in asymptotic graph enumeration, we show not only that (somehow surprisingly) the same bound holds for regular graphs, but also that it can be improved to $\Omega(n \log\log n/\log n)$. We also show that $\Omega\left(d_G\sqrt{\log\log d_G}\right)$ colours are necessary, where d_G is the degree of G.

1 Introduction

Sense of direction and *weak sense of direction* [5] are properties of global consistency of the colouring of a network that can be used to reduce the complexity of many distributed algorithms [4]. Although there are polynomial algorithms for checking whether a given coloured graph has (weak) sense of direction [1], the polynomial bounds are rather high, and, moreover, there are no results (besides the obvious membership to NP) about finding a colouring that is a (weak) sense of direction using the *smallest* number of colours.

The number of vertices n in a graph G is a trivial upper bound for the number of colours, and the maximum degree Δ_G is a trivial lower bound. However, Δ_G was essentially the *only* known lower bound; the difficulty of proving that $\Delta_G + 2$ colours were necessary for some graph prompted for the conjecture that $\Delta_G + 1$ colours were always sufficient [6]; the conjecture is of course of particular interest for regular graphs. Recently the authors proved that there are graphs requiring $\omega(n/\log n)$ additional colours [2] but the proof uses intensively *random graphs* of high degree: therefore, an extension of the proof to regular graphs appears difficult (as the theory of random regular graphs mainly considers fixed or slowly growing degrees).

In this paper, using a recent result in graph asymptotic enumeration [8], we bypass this problem and show that $\Omega(n \log\log n/\log n)$ additional colours are necessary to give weak sense of direction to all regular graphs. This result strongly disproves the original conjecture, even when restricted to regular graphs. We also show

* Partially supported by the Italian MURST (Progetto cofinanziato "Algebra e Teoria dei Tipi nella Specifica e Verifica di Sistemi Complessi").

† Partially supported by the Italian MURST (Finanziamento di iniziative di ricerca "diffusa" condotte da parte di giovani ricercatori).

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 238–252, 2000.

that if the overall number of colours used is dependent on the degree d_G it must be $\Omega\left(d_G\sqrt{\log\log d_G}\right)$.

We remark that even if the main proof of this paper is a rather straightforward counting argument, it is based on an asymptotic estimate of the number of regular graphs enjoying a suitable property, and this estimate requires rather involved computations.

A consequence of our proof is that almost all regular graphs in a certain range of degrees (see Theorem 3) have diameter two. There are presently no published results of this kind in the literature (see [10]), although Krivelevitch, Sudakov, Vu and Wormald are preparing a paper on these issues that covers a wider degree range [9]. However, we believe that the techniques used in the proof can be fruitfully applied to many other properties of random regular graphs.

2 Definitions

A *(directed) graph* G is given by a set $V = [n] = \{0, 1, \ldots, n-1\}$ of n vertices and a set $A \subseteq V \times V$ of arcs (note that the graphs in this paper are *not* considered up to isomorphism—using a common terminology, they are *labelled*). We write $P[x, y] \subseteq A^*$ for the set of paths from vertex x to vertex y. A graph is *symmetric* if $\langle y, x\rangle$ is an arc whenever $\langle x, y\rangle$ is.

In this paper we shall always manipulate symmetric loopless directed graphs, which are really nothing but undirected simple graphs (an edge is identified with a pair of opposite arcs). However, the directed symmetric representation allows us to handle more easily the notion of weak sense of direction and the related proofs. In turn, when using asymptotic enumeration results we shall confuse a symmetric loopless directed graph with its undirected simple counterpart.

The (average) degree d_G of a graph G is $|A|/|V|$ (or, in the undirected interpretation, twice the number of edges divided by the number of vertices). Of course, if G is regular (i.e., all vertices have the same number of incoming and outgoing arcs) then d_G is the (in- and out-)degree of every vertex, and one says that G is d_G-regular.

A *colouring* of a graph G is a function $\lambda : A \to \mathscr{L}$, where $\mathscr{L}$ is a finite set of colours; the map $\lambda^* : A^* \to \mathscr{L}^*$ is defined by $\lambda^*(a_1a_2\cdots a_p) = \lambda(a_1)\lambda(a_2)\cdots\lambda(a_p)$. We write $\mathscr{L}_x = \{\lambda(\langle x, y\rangle) \mid \langle x, y\rangle \in A\}$ for the set of colours that x assigns to its outgoing arcs.

Given a graph G coloured by λ, let

$$L = \bigcup_{\langle x,y\rangle\in V^2} \{\lambda^*(\pi) \mid \pi \in P[x, y]\};$$

be the set of all strings that colour paths of G.

A *local naming* for G is a family of injective functions $\beta = \{\beta_x : V \to \mathscr{S}\}_{x\in V}$, with $\mathscr{S}$ a finite set, called the *name space*. Intuitively, each vertex x of G gives to each other vertex y a name $\beta_x(y)$ taken from the name space.

Given a coloured graph endowed with a local naming, a function $f : L \to \mathscr{S}$ is a *coding function* iff

$$\forall x, y \in V \quad \forall \pi \in P[x, y] \qquad f(\lambda^*(\pi)) = \beta_x(y).$$

A coding function translates the colouring of the path along which two vertices x, y are connected into the name that x gives to y. A colouring λ is a *weak sense of direction* for a graph G iff for some local naming there is a coding function[1]. We shall also say that a coloured graph *has* weak sense of direction, or that λ *gives* weak sense of direction to G.

3 Representing Regular Graphs Using Weak Sense of Direction

A coding function f *represents compactly a great deal of information about a graph*, because f tells whether two paths with the same source have the same target. For instance, suppose that we want to exploit (naively) this property to code compactly a (strongly) connected regular graph G with weak of sense of direction. Assume without loss of generality that $\beta_0(x) = x$ for all vertices x, that is, vertex 0 locally gives to all other vertices their "real names". To code G, first specify for each vertex the set of colours of outgoing arcs. Then, give the values of f on every string of colours having length at most $D + 1$, where D is the diameter of G.

To rebuild G from the above data, we proceed as follows: first of all we compute the targets of the arcs out of 0 using f on strings of length one, thus obtaining the set of coloured paths of length one going out of 0. Then, since we know the colours of the arcs going out of the targets of such paths, we can build the set of coloured paths of length two out of 0, and compute their targets using f on strings of length two, and so on. Thus, we will eventually discover all arcs of G, using just the values of f on paths of length $D + 1$ at most. Unfortunately this naive attempt is too rough, even for $D = 2$, so we shall use a slightly more sophisticated approach.

Let $\mathscr{C}(n, k)$ be the class of all symmetric k-regular graphs with n vertices that enjoy the following property, which we shall call *property* $A_{3/2}$:

> If x_1, x_2, x_3 are three distinct vertices such that x_2 and x_3 are adjacent, and x_1 is not adjacent to x_2 and not adjacent to x_3, then there exists a vertex $z \notin \{x_1, x_2, x_3\}$ that is adjacent to x_1, x_2 and x_3.

Note that property $A_{3/2}$ is weaker than property A_3 of [2]; as we shall see, for suitable d almost all d-regular graphs enjoy property $A_{3/2}$, and nonetheless graphs satisfying $A_{3/2}$ can be coded compactly. Intuitively for regular graphs $A_{3/2}$ is a connectivity property slightly stronger than having diameter two, since given any pair of vertices x, y we can choose a vertex z adjacent to y (the existence of z is ensured by regularity) and apply $A_{3/2}$, getting a vertex that, in particular, is adjacent both to x and to y. The same argument shows also that a regular graph satisfying $A_{3/2}$ is connected.

Lemma 1. *Let G be a graph satisfying $A_{3/2}$. Let λ be a sense of direction for G with name space $\mathscr{S}$, coding function f and local naming β. Assume without loss of generality that $\mathscr{S} \supseteq [n]$ and $\beta_0(x) = x$ for all vertices x. Then, $\langle x, y \rangle$ is an arc iff one of the following holds:*

[1] In [5] a slightly different definition is given, in which the empty string is not part of L. The results of this paper are not affected by this difference.

- $x = 0$ *and* $y = f(a)$ *for some colour* $a \in \mathscr{L}_0$*;*
- $x = f(a)$ *and* $y = f(ab)$ *for some* $a \in \mathscr{L}_0$ *and* $b \in \mathscr{L}_{f(a)}$*;*
- $x = f(ab)$ *and* $y = f(ac)$ *for some* $a \in \mathscr{L}_0$*,* $b, c \in \mathscr{L}_{f(a)}$*, provided that there exists* $d \in \mathscr{L}_{f(ab)}$ *such that* $f(bd) = f(c)$*.*

Proof. If a colours an arc going out of 0, the target of the arc is $f(a)$. Moreover, if there is an arc with colour b going out of $f(a)$, the target of such arc is $f(ab)$. For the third case, ab and ac colour paths going out of 0, whereas bd and c colour paths going out of $f(a)$. Since $f(bd) = f(c)$, the latter paths must have the same target; hence the path from 0 coloured ac has the same target as the path coloured abd. Therefore, there must be an arc, coloured d, from $f(ab)$ to $f(ac)$, as in Fig. 1.

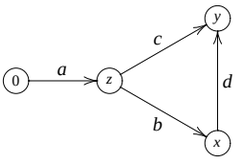


Figure 1. Property $A_{3/2}$ in action.

For the other side of the implication, consider an arc $\langle x, y\rangle$ of G. We have three cases:

- if $x = 0$, then it must correspond to an arc of the form $\langle 0,\ f(a)\rangle$ for some $a \in \mathscr{L}_0$;
- if x is an outneighbour of 0, then $x = f(a)$ for some $a \in \mathscr{L}_0$, and the arc corresponds to an arc of the form $\langle f(a),\ f(ab)\rangle$ for some $b \in \mathscr{L}_{f(a)}$;
- finally, assume that $x, y \neq 0$ and that moreover x and y are not outneighbours of 0. By property $A_{3/2}$, there exists a vertex z adjacent to x, y and 0. Let a be the colour of the arc going from 0 to z, b be the colour of the arc going from z to x, c be the colour of the arc going from z to y, and d be the colour of the arc from x to y. We have $f(ab) = \beta_0(x) = x$, $f(ac) = \beta_0(y) = y$ and $f(bd) = \beta_z(y) = f(c)$ (see Fig. 1). □

Following the line of [2], we can use Lemma 1 to code compactly regular graphs as follows:

Theorem 1. *Let* $c = c(G) \in \mathbf{N}$ *be such that every* k*-regular graph* G *with* n *vertices can be given weak sense direction using no more than* $c(G)$ *colours. Then every graph in* $\mathscr{C}(n, k)$ *can be described*[2] *using* $O\left(cn + c^2 \log n\right)$ *bits.*

Proof. Let $G \in \mathscr{C}(n, k)$ have weak sense of direction with colouring λ, name space $\mathscr{S}$, local naming β and coding function f. Assume without loss of generality that $\mathscr{S} \supseteq [n]$ and $\beta_0(x) = x$ for every vertex x. Describe G as follows:

[2] From now on, we shall sometimes omit the explicit dependence of functions from their argument, when the latter is clear from the context, thus writing c instead of $c(G)$, d instead of $d(n)$ and so on.

1. give the number of colours c;
2. for every vertex x, use c bits to describe the set $\mathscr{L}_x$;
3. give the values of f on every string of length one or two.

The first data require $\lceil \log c \rceil$ bits, the second one cn bits and the third one $\left(c + c^2\right) \lceil 2 \log$ (as we mentioned, $\lceil 2 \log n \rceil$ bits are sufficient to specify a name). From the above description, G can be recovered using Lemma 1. □

4 A Result about Graph Enumeration

The inspiration for this paper came out of a recent breakthrough by McKay and Wormald in asymptotic graph enumeration:

Theorem 2 ([8]). *Let $d = d(n)$ and $\delta_j = \delta_j(n)$, $0 \leq j < n$, be such that $\min\{d, n - d - 1\} > cn/\log n$ for some $c > \frac{2}{3}$, $\sum_{j=0}^{n-1} \delta_j = 0$, $\delta_j = O(1)$ uniformly over j, $d + \delta_j$ is an integer for $0 \leq j < n$ and dn is an even integer. Then the number of graphs with n vertices and local degrees $d + \delta_0$, $d + \delta_1$, $\ldots$, $d + \delta_{n-1}$ is asymptotic to $\alpha M(n, d)$, where*

$$M(n,d) = \left[2\pi n \left(\frac{d}{n-1} \right)^{d+1} \left(1 - \frac{d}{n-1} \right)^{n-d} \right]^{-n/2}$$

and $\gamma_1/n^{\varepsilon} \leq \alpha \leq \gamma_2$ for suitable positive constants ε, γ_1 and γ_2.

In other words, under the given hypotheses the order of magnitude depends essentially on the average degree only, and not on the specific degrees (but note that α in general will depend on n, on d and on the δ_j's). The original result of McKay and Wormald is much more powerful, as it provides a precise asymptotic estimate for much more varied δ_j's, but the simplification above is sufficient for our purposes.

The above theorem has the following consequence, whose (complex) proof is deferred to the last section:

Theorem 3. *Let $d = o(n)$ satisfy the hypotheses of Theorem 2. Then almost all d-regular graphs satisfy $A_{3/2}$.*

The statement "almost all d-regular graphs satisfy P" means that the number of d-regular graph of order n enjoying P divided by the number of all d-regular graphs of order n goes to 1 as $n \to \infty$. Equivalently, if we consider the standard model of d-regular random graphs [3] in which all d-regular graphs of order n are equiprobable, we can say that the probability that a random graph satisfies P goes to 1 as $n \to \infty$.

Since under the given hypotheses the number of all d-regular graphs is asymptotic to the number of d-regular graphs satisfying $A_{3/2}$, we can use Theorem 2 to get an asymptotic estimate of the size of the class $\mathscr{C}(n, d)$, and thus a lower bound on the number of bits that are necessary to describe a graph belonging to it.

Theorem 4. *Let $d = o(n)$ satisfy the hypotheses of Theorem 2. Then, the number of bits required to describe a graph in $\mathscr{C}(n, d)$ is $\Theta(nd \log(n/d))$.*

Proof. Since by Theorem 3 the number of graphs in $\mathscr{C}(n,d)$ is asymptotic to the number of d-regular graphs, Theorem 2 tells us that the number of bits required is asymptotic to $\log[\alpha M(n,d)]$. If we expand the latter expression killing all terms that are $O(nd)$ we obtain

$$\log[\alpha M(n,d)] = -\frac{n}{2}(1+d)\log\frac{d}{n-1} - \frac{n}{2}(n-d)\log\left(1-\frac{d}{n-1}\right) + O(nd)$$
$$= \Theta\left(nd\log\frac{n}{d}\right) + \frac{n}{2}(n-d)\frac{d}{n-1} + O(nd) = \Theta\left(nd\log\frac{n}{d}\right). \square$$

5 The Main Theorem

We finally put together the upper and lower bounds we obtained:

Theorem 5. *If $g(n) = o(n\log\log n/\log n)$, it is impossible to give (weak) sense of direction to all regular graphs using $d_G + g(n)$ colours. Moreover, it is impossible to give (weak) sense of direction to all regular graphs using $o\left(d_G\sqrt{\log\log d_G}\right)$ colours.*[3]

Proof. If $g = O(n/\log n)$, take any $d = \Theta(n/\log n)$ satisfying the hypotheses of Theorem 2 and note that by Theorem 1 $O\left(n^2/\log n\right)$ bits would be sufficient to describe a graph in $\mathscr{C}(n,d)$, but by Theorem 4 $\Theta\left(n^2\log\log n/\log n\right)$ are required. Otherwise, we can write $g = n\,f(n)/\log n$, with $f(n) = o(\log\log n)$, and take $d = g$. In this case $O\left(n^2 f(n)^2/\log n\right) = o\left(n^2 f(n)\log\log n/\log n\right)$ bits would be sufficient, but $\Theta\left(n^2 f(n)\log\log n/\log n\right)$ are required.

Finally, if $h(m) = o\left(m\sqrt{\log\log m}\right)$ as $m \to \infty$ take any $d = \Theta(n/\log n)$; in this case $O\left(h(d)^2/\log n\right) = o\left(n^2\log\log n/\log n\right)$ bits would be sufficient, but again $\Theta\left(n^2\log\log n/\log n\right)$ are necessary. $\square$

6 A Proof of Theorem 3—Part I

To prove that almost all d-regular graphs satisfy $A_{3/2}$, we show that almost no d-regular graph satisfies $\neg A_{3/2}$. The interesting feature of a d-regular graph G with n vertices that does not satisfy $A_{3/2}$ is that it has a rather precise structure, displayed in Fig. 2, where $A_{3/2}$ does not work on x_1, x_2 and x_3 (in Fig. 2 we draw only edges incident on x_1, x_2 and x_3).

The vertices of G are partitioned into seven sets, depending on their adjacency relations with the three vertices on which $A_{3/2}$ does not work. If we strip x_1, x_2 and x_3 we obtain a new "stripped graph" with $n-3$ vertices and a rather precise degree assignment: clearly all vertices in $V_\varnothing$ will have degree d, all vertices in the sets V_i will have degree $d-1$ and all vertices in the sets V_{ij} will have degree $d-2$. The key point is that such a degree structure still falls under the scope of Theorem 2. Since, as we will show, the average degree d' of a stripped graph is independent of the actual cardinalities of the

[3] That is, for every function $h(m) = o\left(m\sqrt{\log\log m}\right)$ there is a graph G such that $h(d_G)$ colours are not sufficient.

V's, we may hope to bound the number of counterexamples to $A_{3/2}$ using $M(n-3, d')$ to bound carefully the number of stripped graphs. To this goal, we work backwards and define a suitable kind of graph that can be enriched with three vertices so to obtain a d-regular counterexample to $A_{3/2}$ of order n.

An (n, d)-*stripped graph* is a graph S with $n-3$ vertices, endowed with a vertex-colouring function $\pi : [n-3] \to 2^{\{1,2,3\}}$, and satisfying the following conditions: let us write V_1 for the set of vertices coloured by $\{1\}$, V_{12} for the set coloured by $\{1, 2\}$ and so on (formally, $V_X = \pi^{-1}(X)$ for $X \subseteq \{1, 2, 3\}$), and finally let $v_X = |V_X|$; we require that

$$\begin{aligned} \deg(x) + |\pi(x)| &= d \text{ for every vertex } x \text{ of } S \\ v_{123} &= 0 \\ v_1 + v_{12} + v_{13} &= d \\ v_2 + v_{12} + v_{23} &= d - 1 \\ v_3 + v_{13} + v_{23} &= d - 1 \end{aligned}$$

The rationale behind the previous equalities is immediate, looking at Fig. 2. Note that, as a consequence,

$$v_1 + v_2 + v_3 + 2(v_{12} + v_{13} + v_{23}) = 3d - 2.$$

We can associate to each (n, d)-stripped graph S a d-regular counterexample to $A_{3/2}$

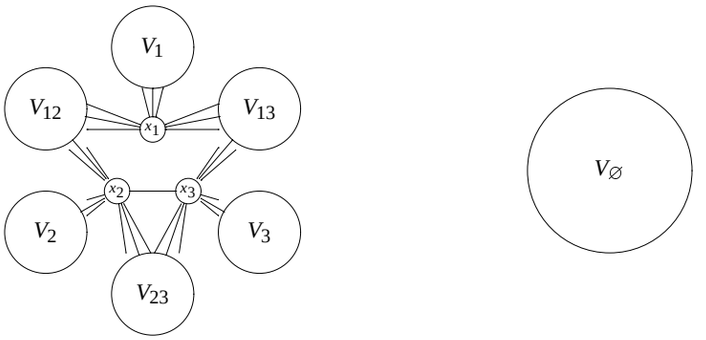


Figure 2. A generic counterexample to $A_{3/2}$.

with n vertices in the following manner: we add three new vertices $n-3$, $n-2$ and $n-1$ to S, and connect vertex $n-y$ to vertex $x < n-3$ if and only if $y \in \pi(x)$. Moreover, vertex $n-2$ is adjacent to vertex $n-3$. It is straightforward to observe that the graph constructed as above is d-regular (because of the condition $\deg(x) + |\pi(x)| = d$). Moreover, vertices $n-3$, $n-2$ and $n-1$ fail to satisfy property $A_{3/2}$; conversely, every d-regular graph of size n whose last three vertices fail to satisfy $A_{3/2}$ may be obtained from a suitable (n, d)-stripped graph using the above construction. Finally, we

remark that we can choose for each counterexample to $A_{3/2}$ a relabelling that exactly exchanges the labels of the (say, lexicographically first) three vertices that break $A_{3/2}$ with those of last three vertices. As a result, we have that at most n^3 counterexamples to $A_{3/2}$ correspond to an (n, d)-stripped graph. Our next goal is thus to bound the number of (n, d)-stripped graphs, and to this purpose we state some simple properties of the variables v that are easily derivable from the linear system above:

Lemma 2. *Let S, π define an (n, d)-stripped graph, $k = n-3-v_\varnothing$ and $s = v_{12}+v_{13}$; then:*

1. *the average degree of S is $d' = d - (3d-2)/(n-3)$;*
2. *the following (in)equalities hold:*

$$v_{12} + v_{13} + v_{23} = 3d - 2 - k \tag{1}$$
$$v_2 = k - 2d + v_{13} + 1 \tag{2}$$
$$v_1 + v_2 = k - d - v_{12} + 1 \tag{3}$$
$$\lceil (3d-2)/2 \rceil \leq k \leq 3d - 2 \tag{4}$$
$$\max(0, 4d - 2k - 2) \leq s \leq \min(d, 3d - 2 - k) \tag{5}$$
$$\max(0, 2d - k - 1) \leq v_{12} \leq \min(s, k - 2d + s + 1). \tag{6}$$

Proof. 1. The average degree of S is:

$$\begin{aligned} d' &= \frac{v_\varnothing d + (v_1 + v_2 + v_3)(d-1) + (v_{12} + v_{13} + v_{23})(d-2)}{n-3} \\ &= d - \frac{v_1 + v_2 + v_3 + 2(v_{12} + v_{13} + v_{23})}{n-3} = d - \frac{3d-2}{n-3}. \end{aligned}$$

2. Equation (1) directly follows from the constraints; hence we have $v_2 = d - v_{12} - v_{23} - 1 = k - 2d + v_{13} + 1$, proving (2). Moreover, $v_1 = d - v_{12} - v_{13}$, hence $v_1 + v_2 = k - d - v_{12} + 1$, which is (3). For proving inequality (4), observe that $k + v_{12} + v_{13} + v_{23} = 3d - 2$ implies $k \leq 3d - 2$; moreover, since $2(v_{12} + v_{13} + v_{23}) = 3d - 2 - (v_1 + v_2 + v_3)$, we have $k = (v_1 + v_2 + v_3 + 3d - 2)/2 \geq (3d-2)/2$. For (5), first recall that $v_2 = k - 2d + v_{13} + 1$, and similarly $v_3 = k - 2d + v_{12} + 1$; the nonnegativity constraints on v_2 and v_3 give $v_{12} \geq 2d - 1 - k$ (which is the only nontrivial lower bound for the remaining pair of inequalities) and $v_{13} \geq 2d - 1 - k$, hence the lower bound on $s = v_{12} + v_{13}$. On the other hand, $s = 3d - 2 - k - v_{23} \leq 3d - 2 - k$ and also $s = v_{12} + v_{13} = d - v_1 \leq d$. Finally, for (6), we have $v_{12} = s - v_{13} \leq s$, and moreover, since $v_{13} \geq 2d - k - 1$, $s = v_{12} + v_{13} \geq 2d - k - 1 + v_{12}$, hence the bound $v_{12} \leq s + k + 1 - 2d$. □

How can we bound the number of (n, d)-stripped graphs? Looking at the linear system above it is clear that once we choose values for k, s and v_{12} within the bounds of Lemma 2 all other v's are uniquely determined, as $v_{13} = s - v_{12}$, $v_{23} = 3d - 2 - k - s$, and the remaining values can always be computed (the system has maximum rank). Thus, the number of vertices to be assigned a certain colour is now fixed: we just have to choose *which* vertices will receive a certain colour. This can be done choosing first k vertices out of $n - 3$ (that is, the set of vertices with degree smaller than d); then

choosing the $3d-2-k$ vertices out of k that will have degree $d-2$; among the latter we must first choose the s vertices that belong to $V_{12} \cup V_{13}$, and out of these the v_{12} vertices of V_{12}; then, among the $k-(3d-2-k) = 2k-3d+2$ vertices of degree $d-1$ we must choose the $k-d-v_{12}+1$ vertices in $V_1 \cup V_2$, and finally out of these the $d-s$ vertices of V_1. Once also this choice is fixed, the bound of McKay and Wormald tells us that the number of graphs with the sequence of degrees given by the choices above is at most $\gamma_2\, M(n-3, d')$. All in all, we obtain the following horrendous-looking triple summation:

$$\sum_{k=\lceil (3d-2)/2 \rceil}^{3d-2} \sum_{s=\max\{0,4d-2k-2\}}^{\min\{d,3d-2-k\}} \sum_{v_{12}=\max\{0,2d-k-1\}}^{\min\{s,k-2d+s+1\}} \binom{n-3}{k} \binom{k}{3d-2-k} \cdot \\ \cdot \binom{3d-2-k}{s} \binom{s}{v_{12}} \binom{2k-3d+2}{k-d-v_{12}+1} \binom{k-d-v_{12}+1}{d-s} \gamma_2\, M(n-3, d')$$

However, things are not as bad as they may seem: the last factor is independent of all summation indices, and the first three binomials are independent of v_{12}, so they can be moved out accordingly. Finally, applying trinomial revision[4] to the last two binomials we remove a dependence on v_{12}, getting to

$$\gamma_2\, M(n-3, d') \sum_{k=\lceil (3d-2)/2 \rceil}^{3d-2} \sum_{s=\max\{0,4d-2k-2\}}^{\min\{d,3d-2-k\}} \binom{n-3}{k} \binom{k}{3d-2-k} \cdot \\ \cdot \binom{3d-2-k}{s} \binom{2k-3d+2}{d-s} \sum_{v_{12}=\max\{0,2d-k-1\}}^{\min\{s,k-2d+s+1\}} \binom{s}{v_{12}} \binom{2k-4d+s+2}{k-2d+s-v_{12}+1}.$$

Since we are interested in an upper bound, we can extend the last summation to $0 \leq v_{12} \leq k-2d+s+1$ and use Vandermonde convolution[5]. The resulting term is a central binomial coefficient of upper index $2k-4d+2s+2$, and can be bounded with $2^{2k-4d+2s+2}$; the part independent of k and s can be moved out, getting to

$$\gamma_2\, M(n-3, d')\, 4^{1-2d} \sum_{k=\lceil (3d-2)/2 \rceil}^{3d-2} \sum_{s=\max\{0,4d-2k-2\}}^{\min\{d,3d-2-k\}} \binom{n-3}{k} \cdot \binom{k}{3d-2-k} \cdot \\ \binom{3d-2-k}{s} \binom{2k-3d+2}{d-s} 4^{k+s}.$$

There is not much more we can do about the summation term. The summation indices k and s appear almost everywhere, so we take a different approach: since the range of summation is extremely small (see Fig. 3) when compared to the summands, we can try to find an upper bound for the latter. To this aim, we study the behaviour of finite differences in k and s over the range of summation. This is a standard technique

[4] The *trinomial revision* theorem states that $\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}$—see, e.g., [7].

[5] *Vandermonde convolution*: $\sum_{k=0}^{n} \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$, *ibid.*.

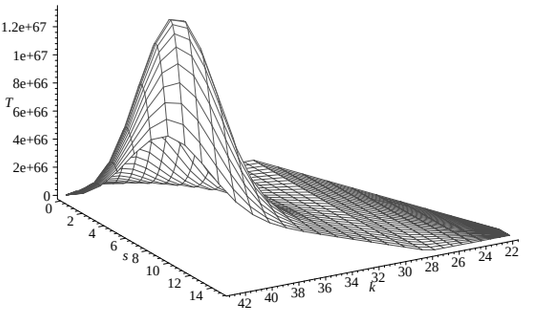


Figure 3. A look at the behaviour of $T_n(s,k)$ for $n = 100$ and $d = \lfloor n/\log n \rfloor$.

used when binomials are involved, as the sign of finite differences usually depends in a simple way on a low-degree polynomial. Indeed, if we let

$$T_n(s,k) = \binom{n-3}{k}\binom{k}{3d-2-k}\binom{3d-2-k}{s}\binom{2k-3d+2}{d-s}4^{k+s}$$

it is immediate to discover that

$$T_n(s,k+1) \geq T_n(s,k) \iff K_n(s,k) \geq 0 \tag{7}$$
$$T_n(s+1,k) \geq T_n(s,k) \iff S_n(s,k) \geq 0, \tag{8}$$

where

$$K_n(s,k) = (4d+6-4n)k - s^2 + (5+8d-4n)s + 12nd - 8n - 8d + 12 - 16d^2$$
$$S_n(s,k) = (2s-2-4d)k + 3s^2 + (-12d+4)s + 12d^2 - 4d - 3.$$

Since both polynomials are linear in k (with ultimately negative coefficient), we can make conditions (7) and (8) explicit, obtaining two rational functions $z_K(s)$ and $z_S(s)$ such that the inequalities

$$k \leq z_K(s) = -\frac{1}{2}\frac{s^2 + (4n-8d-5)s + 8n - 12nd + 8d - 12 + 16d^2}{2n-2d-3}$$
$$k \leq z_S(s) = -\frac{1}{2}\frac{3s^2 + (4-12d)s + 12d^2 - 3 - 4d}{s-1-2d}$$

are equivalent to (7) and (8), respectively.

Armed with the knowledge above, we now try to answer to the following question: which conditions must a pair of integers $\langle s,k\rangle$ satisfy to be a local maximum of T_n? Clearly the strict version of both condition (7) and condition (8) must be false at $\langle s,k\rangle$, for the "next" integers on the plane must feature a smaller or equal value of T_n; similarly, condition (7) must be true at $\langle s,k-1\rangle$ and condition (8) must be true at $\langle s-1,k\rangle$. All

in all, we obtain the following set of constraints:

$$\begin{array}{ll} k \geq z_K(s) & k \geq z_S(s) \\ k \leq z_K(s) + 1 & k \leq z_S(s-1) \end{array}$$

The situation is depicted in Fig. 4 for $n = 10^6$ and $d = \lfloor n/\log n \rfloor$. The thicker curves represent the constraints involving z_K, and the thinner ones the constraints involving z_S. The region satisfying the constraints is the lozenge formed by the four curves (an

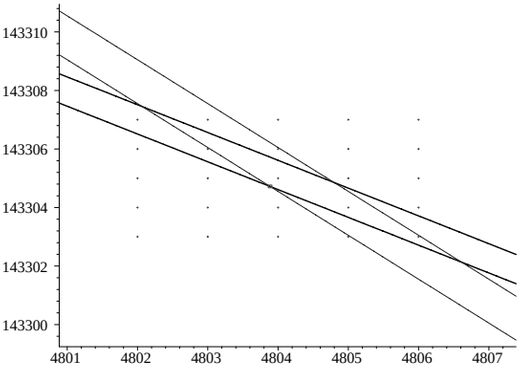


Figure 4. The constraints on the local maxima of T_n

easy check on the values of z_K, z_S and their derivatives on the range of summation shows that indeed this is always the case). The marked point at the intersection of z_S and z_K is the only common zero of K_n and S_n in the range of summation, and can be easily computed with elementary techniques. Its coordinates are

$$\hat{s} = 2\frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right) \qquad \hat{k} = 3d - 3\frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right). \tag{9}$$

Our goal now is to show that knowing $\hat{k}$ and $\hat{s}$ with the precision shown above is sufficient to know with the same precision the location of the global maximum of T_n over the pairs of integers in the summation range. In other words, we just have to show that that all integral points in the lozenge are not too far from $\langle \hat{s}, \hat{k} \rangle$. To this purpose, it is sufficient to give a rough estimate of the size of a rectangle containing the lozenge, for instance the rectangle defined by the upper and lower intersection points, which happen to be also the leftmost and rightmost, respectively. Theoretically it is possible to compute this points exactly, but unfortunately they are the unmanageable roots of two cubic

equations. However, standard algebraic manipulation shows that

$$z_S\left(\hat{s}-\gamma\frac{d^3}{n^2}\right)-z_K\left(\hat{s}-\gamma\frac{d^3}{n^2}\right)-1=\frac{1}{2}(\gamma-2)\frac{d^3}{n^2}-r+o\left(\frac{d^3}{n^2}\right)$$
$$z_K\left(\hat{s}+\gamma\frac{d^3}{n^2}\right)-z_S\left(\hat{s}+\gamma\frac{d^3}{n^2}-1\right)=\frac{1}{2}(\gamma+2)\frac{d^3}{n^2}+r+o\left(\frac{d^3}{n^2}\right),$$

where $r = \hat{s} - 2d^2/n = O(d^3/n^3)$. In other words, we can choose a fixed γ so that ultimately at distance $\gamma d^3/n^2$ to the left of $\hat{s}$ the thinner curves are both over the thicker ones, and conversely at distance $\gamma d^3/n^2$ to the right. This shows that the width of the lozenge is $O(d^3/n^2)$. Finally, it is easy to check that $z_S(\hat{s} - \gamma d^3/n^2) - z_K(\hat{s} + \gamma d^3/n^2) = O(d^3/n^2)$, so the lozenge is included in a rectangle whose sides are both $O(d^3/n^2)$. We conclude the global maximum on integers pair is attained at a point $\langle \bar{s}, \bar{k} \rangle$ satisfying

$$\bar{s}=2\frac{d^2}{n}+O\left(\frac{d^3}{n^2}\right)\qquad\qquad \bar{k}=3d-3\frac{d^2}{n}+O\left(\frac{d^3}{n^2}\right),$$

that is, modulo error terms $O(d^3/n^2)$, the same as (9). As the reader will see, this will be sufficient for our purposes.

We have thus finally reached our goal: our bound on the number of (n, d)-stripped graphs becomes

$$\gamma_2\, M(n-3, d')\, d\lfloor(3d-2)/2\rfloor T_n(\bar{s}, \bar{k}),$$

and the proof that

$$\frac{n^3\gamma_2\, M(n-3, d')\, d\lfloor(3d-2)/2\rfloor T_n(\bar{s}, \bar{k})}{(\gamma_1/n^\varepsilon)M(n, d)} \to 0$$

is now amenable to standard asymptotic techniques. In the next section we provide a full (and rather tedious) proof.

7 A Proof of Theorem 3—Part II

We start with a few obvious considerations: since we have exponentials around, we need to estimate the natural logarithm of our expression. In doing so, we plan in advance to consider only summands that are of order at least d^3/n^2: in particular, the factor $\gamma_2 n^{3+\varepsilon}\lfloor(3d-2)/2\rfloor/\gamma_1$ plays no rôle. We remark the following asymptotic relations between the principal quantities we will have to manage:

$$d=\Omega\left(\frac{n}{\log n}\right)\quad d\sim d'=o(n)\quad d-d'=\frac{3d}{n}+O\left(\frac{1}{n}\right)\quad \frac{d^{t+1}}{n^t}=o\left(\frac{d^t}{n^{t-1}}\right)$$

We start by computing the natural logarithm of $M(n-3,d')/M(n,d)$:

$$\begin{aligned}
\ln\frac{M(n-3,d')}{M(n,d)} &= -\frac{n-3}{2}[\ln(2\pi)+\ln(n-3)-(n-2)\ln(n-4)+\\
&\quad +(d'+1)\ln d'+(n-3-d')\ln(n-4-d')]+\\
&\quad +\frac{n}{2}[\ln(2\pi)+\ln n-(n+1)\ln(n-1)+(d+1)\ln d+(n-d)\ln(n-1-d)]\\
&= -\frac{n}{2}\Bigg[\ln\frac{n-3}{n}-n\ln\frac{n-4}{n-1}+\ln\frac{(n-4)^2(n-1)}{(n-4-d')^3}+(d+1)\ln\frac{d'}{d}-\frac{3d-2}{n-3}\ln d'+\\
&\quad +(n-d)\ln\frac{n-4-d'}{n-1-d}+\frac{3d-2}{n-3}\ln(n-4-d')\Bigg]+\frac{3}{2}[\ln(2\pi)+\\
&\quad +\ln(n-3)-(n-2)\ln(n-4)+(d'+1)\ln d'+(n-3-d')\ln(n-4-d')\big]
\end{aligned}$$

Since we are going to expand asymptotically all logarithms, we notice that

$$\frac{(n-4)^2(n-1)}{(n-4-d')^3}=1+3\frac{d'}{n}+6\frac{d'^2}{n^2}+10\frac{d'^3}{n^3}+O\left(\frac{d'^4}{n^4}\right),$$

so we obtain

$$\begin{aligned}
&= -\frac{n}{2}\Bigg[\ln\left(1-\frac{3}{n}\right)-n\ln\left(1-\frac{3}{n-1}\right)+\ln\left(1+3\frac{d'}{n}+6\frac{d'^2}{n^2}+10\frac{d'^3}{n^3}+O\left(\frac{d'^4}{n^4}\right)\right)+\\
&\quad +(d+1)\ln\left(1-\frac{3d-2}{d(n-3)}\right)-\frac{3d-2}{n-3}\ln d'+(n-d)\ln\left(1+\frac{d-d'-3}{n-1-d}\right)+\\
&\quad +\frac{3d-2}{n-3}\ln(n-4-d')\Bigg]+\frac{3}{2}[\ln(2\pi)+\ln(n-3)-(n-2)\ln(n-4)+\\
&\quad +(d'+1)\ln d'+(n-3-d')\ln(n-4-d')\big].
\end{aligned}$$

Note that since $\ln(1+x)=x-x^2/2+x^3/3+O(x^4)$ for $x\to 0$, if $g=o(f)$ then

$$\begin{aligned}
\ln(f+g) &= \ln f+\frac{g}{f}-\frac{g^2}{2f^2}+\frac{g^3}{3f^3}+O\left(\frac{g^4}{f^4}\right)\\
(f+g)\ln(f+g) &= (f+g)\ln f+g+\frac{g^2}{2f}-\frac{g^3}{6f^2}+O\left(\frac{g^4}{f^3}\right).
\end{aligned}$$

Applying the expansions above and systematically killing all terms that are $o\left(d^3/n^2\right)$ we obtain

$$\begin{aligned}
&\ln \frac{M(n-3,d')}{M(n,d)} \\
&= -\frac{n}{2}\Bigg[\frac{3n}{n-1} + 3\frac{d'}{n} + 6\frac{d'^2}{n^2} + 10\frac{d'^3}{n^3} - \frac{9}{2}\frac{d'^2}{n^2} - 18\frac{d'^3}{n^3} + 9\frac{d'^3}{n^3} - \frac{(d+1)(3d-2)}{d(n-3)} + \\
&\quad - \frac{3d-2}{n-3}\ln d' + \frac{(n-d)(d-d'-3)}{n-1-d} + \frac{3d-2}{n-3}\ln n + \\
&\quad + \frac{3d-2}{n-3}\left(-\frac{4+d'}{n} - \frac{1}{2}\frac{(4+d')^2}{n^2}\right)\Bigg] + \frac{3}{2}\Bigg[-n\ln n + d'\ln d' + (n-d')\ln n + \\
&\quad + (n-d')\left(-\frac{4+d'}{n} - \frac{1}{2}\frac{(4+d')^2}{n^2} - \frac{1}{3}\frac{(4+d')^3}{n^3}\right)\Bigg] + o\left(\frac{d^3}{n^2}\right) \\
&= -3d + \frac{3}{2}\frac{d^2}{n} + \frac{1}{2}\frac{d^3}{n^2} + 3d\ln\frac{d}{n} + o\left(\frac{d^3}{n^2}\right),
\end{aligned}$$

where the last passage is just algebra, once one notes that it is possible to replace d' with d inside logarithms (the resulting error is within our bound).

We now approach the rest of the limit. We want to estimate the behaviour of

$$\begin{aligned}
T_n(\bar{s},\bar{k}) &= \binom{n-3}{\bar{k}}\binom{\bar{k}}{3d-2-\bar{k}}\binom{3d-2-\bar{k}}{s}\binom{2\bar{k}-3d+2}{d-\bar{s}}4^{\bar{k}+\bar{s}} \\
&= \frac{4^{\bar{k}+\bar{s}}(n-3)!}{\bar{s}!(n-3-\bar{k})!(3d-2-\bar{k}-\bar{s})!(d-\bar{s})!(2\bar{k}-4d+2+\bar{s})!}
\end{aligned}$$

using the following asymptotic identity derived from Stirling approximation:

$$\ln[(f+g)!] = (f+g)\ln f - f + \frac{g^2}{2f} - \frac{g^3}{6f^2} + O\left(\frac{g^4}{f^3}\right) + O(\ln(f+g)),$$

which is true when $g = o(f)$, and always keeping in mind that

$$\bar{k} = 3d - 3\frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right) \qquad \bar{s} = 2\frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right)$$

$$3d - \bar{k} - \bar{s} = \frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right) \qquad 2\bar{k} - 4d + \bar{s} = 2d - 4\frac{d^2}{n} + O\left(\frac{d^3}{n^2}\right).$$

Applying systematically the identities above we obtain:

$$\begin{aligned}
\ln T_n\left(\bar{k},\bar{s}\right) &= (\bar{k}+\bar{s}-2d+1)\ln 4+(n-3)\ln n-n-(n-\bar{k}-3)\ln n+n+\\
&\quad -\frac{(\bar{k}+3)^2}{2n}-\frac{(\bar{k}+3)^3}{6n^2}-\bar{s}\ln\frac{2d^2}{n}+2\frac{d^2}{n}-(3d-2-\bar{k}-\bar{s})\ln\frac{d^2}{n}+\frac{d^2}{n}+\\
&\quad -(d-\bar{s})\ln d+d-\frac{\bar{s}^2}{2d}-(2\bar{k}-4d+2+\bar{s})\ln(2d)+2d+\\
&\quad -\frac{1}{4d}\left[-4\frac{d^2}{n}+O\left(\frac{d^3}{n^2}\right)\right]^2+o\left(\frac{d^3}{n^2}\right)\\
&= (\bar{k}+\bar{s}-2d)\ln 4+\bar{k}\ln n-\frac{(\bar{k}+3)^2}{2n}-\frac{(\bar{k}+3)^3}{6n^2}-\bar{s}\ln\frac{2d^2}{n}+3\frac{d^2}{n}+\\
&\quad -(3d-\bar{k}-\bar{s})\ln\frac{d^2}{n}-(d-\bar{s})\ln d-\frac{\bar{s}^2}{2d}-(2\bar{k}-4d+\bar{s})\ln(2d)+\\
&\quad +3d-4\frac{d^3}{n^2}+o\left(\frac{d^3}{n^2}\right)=3d-3d\ln\frac{d}{n}-\frac{3}{2}\frac{d^2}{n}-\frac{3}{2}\frac{d^3}{n^2}+o\left(\frac{d^3}{n^2}\right),
\end{aligned}$$

where again the last passage is just algebra (all logarithms cancel out happily). Finally, we put together everything, getting to

$$-3d+\frac{3}{2}\frac{d^2}{n}+\frac{1}{2}\frac{d^3}{n^2}+3d\ln\frac{d}{n}+3d-3d\ln\frac{d}{n}-\frac{3}{2}\frac{d^2}{n}-\frac{3}{2}\frac{d^3}{n^2}+o\left(\frac{d^3}{n^2}\right)\to-\infty,$$

as required.

References

1. Paolo Boldi and Sebastiano Vigna. Complexity of deciding sense of direction. *SIAM J. Comput.*, 29(3):779–789, 2000.
2. Paolo Boldi and Sebastiano Vigna. Lower bounds for weak sense of direction. In *Structure, Information and Communication Complexity. Proc. 7th Colloquium SIROCCO 2000*, Proceedings In Informatics. Carleton Scientific, 2000.
3. Béla Bollobás. *Random Graphs*. Academic Press, London, 1985.
4. Paola Flocchini, Bernard Mans, and Nicola Santoro. On the impact of sense of direction on message complexity. *Inform. Process. Lett.*, 63(1):23–31, 1997.
5. Paola Flocchini, Bernard Mans, and Nicola Santoro. Sense of direction: Definitions, properties, and classes. *Networks*, 32(3):165–180, 1998.
6. Paola Flocchini, Bernard Mans, and Nicola Santoro. Sense of direction in distributed computing. In *Proc. DISC '98*, volume 1499 of *Lecture Notes in Computer Science*, pages 1–15, 1998.
7. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison–Wesley, second edition, 1994.
8. Brendan D. McKay and Nicholas C. Wormald. Asymptotic enumeration by degree sequence of graphs of high degree. *European J. Combin.*, 11(6):565–580, 1990.
9. Nicholas C. Wormald. Personal electronic communication.
10. Nicholas C. Wormald. Models of random regular graphs. In *Surveys in combinatorics, 1999 (Canterbury)*, pages 239–298. Cambridge Univ. Press, Cambridge, 1999.

Gossip versus Deterministically Constrained Flooding on Small Networks*

Meng-Jang Lin[1], Keith Marzullo[2], Stefano Masini[3]

[1] Somerset Design Center, Motorola, Inc., Austin, TX.
[2] Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA.
[3] Department of Computer Science, University of Bologna, Bologna, Italy.

Abstract. *Rumor mongering* (also known as *gossip*) is an epidemiological protocol that implements broadcasting with a reliability that can be very high. Rumor mongering is attractive because it is generic, scalable, adapts well to failures and recoveries, and has a reliability that gracefully degrades with the number of failures in a run. However, rumor mongering uses random selection for communications. We study the impact of using random selection in this paper. We present a protocol that superficially resembles rumor mongering but is deterministic. We show that this new protocol has most of the same attractions as rumor mongering. The one remaining attraction that rumor mongering has over the determinisitic protocol—namely graceful degradation—comes at a high cost in terms of the number of messages sent. We compare the two approaches both at an abstract level and in terms of how they perform in an Ethernet and small wide area network of Ethernets.

1 Introduction

Consider the problem of designing a protocol that broadcasts messages to all of the processors in a network. One can be interested in different metrics of a broadcast protocol, such as the number of messages it generates, the time needed for the broadcast to complete, or the reliability of the protocol (where *reliability* is the probability that either all or no nonfaulty processors deliver a broadcast message and that all nonfaulty processors deliver the message if the sender is nonfaulty). Having fixed a set of metrics, one then chooses an abstraction for the network. There are two approaches that have been used in choosing such an abstraction for broadcast protocols.

One approach is to build upon the specific physical properties of the network. For example, there are several broadcast protocols that attain very high reliability for Ethernet networks [17] or redundant Ethernets [2, 5, 7]. Such protocols can be very efficient in terms of the chosen metrics because one can leverage

* This research was support in part by DARPA grant N66001-98-8911 and NSF award CCR-9803743. Most of this work was done when Dr. Lin was a graduate student at UT Austin.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 253–267, 2000.

off of the particulars of the network. On the other hand, such protocols are not very portable, since they depend so strongly on the physical properties of the network.

The other approach is to assume a generic network. With this approach, one chooses a set of basic network communication primitives such as sending and receiving a message. If reliability is a concern, then one can adopt a failure model that is generic enough to apply to many different physical networks. There are many examples of reliable broadcast protocols for such generic networks [14]. We consider in this paper broadcast protocols for generic networks.

Unfortunately, many reliable broadcast protocols for generic networks do not scale well to large numbers of processors [4]. One family of protocols for generic networks that are designed to scale are called *epidemiological algorithms* or *gossip protocols*.[4] Gossip protocols are probabilistic in nature: a processor chooses its *partner* processors with which to communicate randomly. They are scalable because each processor sends only a fixed number of messages, independent of the number of processors in the network. In addition, a processor does not wait for acknowledgments nor does it take some recovery action should an acknowledgment not arrive. They achieve fault-tolerance against intermittent link failures and processor crashes because a processor receives copies of a message from different processors. No processor has a specific role to play, and so a failed processor will not prevent other processors from continuing sending messages. Hence, there is no need for failure detection or specific recovery actions.

A drawback of gossip protocols is the number of messages that they send. Indeed, one class of gossip protocols (called *anti-entropy* protocols [9]) send an unbounded number of messages in nonterminating runs. Such protocols seem to be the only practical way that one can implement a gossip protocol that attains a high reliability in an environment in which links can fail for long periods of time [26]. Hence, when gossiping in a large wide-area network, anti-entropy protocols are often used to ensure high reliability. However, for applications that require timely delivery, the notion of reliability provided by anti-entropy may not be strong enough since it is based on the premise of *eventual delivery* of messages.

Another class of gossip protocols is called *rumor mongering* [9]. Unlike anti-entropy, these protocols terminate and so the number of messages that are sent is bounded. The reliability may not be as high as anti-entropy, but one can trade off the number of messages sent with reliability. Rumor mongering by itself is not appropriate for networks that can partition with the prolonged failure of a few links, and so is best applied to small wide-area networks and local area networks.

Consider the undirected clique that has a node for each processor, and let one processor p broadcast a value using rumor mongering. Assume that there are no failures. As the broadcast takes place, processors choose partners at random.

[4] The name *gossip* has been given to different protocols. For example, some authors use the term gossip to mean all-to-all communications, and what we describe here would be called *random broadcasting in the whispering mode* [21].

For each processor q and a partner r it chooses, mark the edge from q to r. At the end of the broadcast, remove the edges that are not marked. The resulting graph is the *communications graph* for that broadcast. This graph should be connected, since otherwise the broadcast did not reach all processors. Further, the *node* and *link* connectivities of the communications graph give a measure of how well this broadcast would have withstood a set of failures. For example, if the communications graph is a tree, and if any processor represented by an internal node had crashed before the initiation of the broadcast, then the broadcast would not have reached all non-crashed processors.

In this paper, we compare rumor mongering with a deterministic version of rumor mongering. This deterministic protocol superimposes a communication graph that has a minimal number of links given a desired connectivity. The connectivity is chosen to attain a desired reliability, and by being minimal link, the broadcast sends a small number of messages. This comparison allows us to ask the question *what value does randomization give to rumor mongering*? We show that the deterministic version does compare favorably with traditional rumor mongering in all but one metric, namely graceful degradation.

We call the communications graphs that we impose *Harary graphs* because the construction we use comes from a paper by Frank Harary [15]. The deterministic protocol that we compare with rumor mongering is a simple flooding protocol over a Harary graph.

The rest of the paper proceeds as follows. We first discuss some related research. We then describe gossip protocols and their properties. Next, we describe Harary graphs and show that some graphs yield higher reliabilities than others given a fixed connectivity. We then compare Harary graph-based flooding with gossip protocols both at an abstract level and using a simple simulation.

Due to space restrictions, no theorems are proven in this paper. Interested readers can find all of the proofs in [20].

2 Related Work

Superimposing a communications graph is a well-known technique for implementing broadcast protocols. Let an undirected graph $G = (V, E)$ represent such a superimposed graph, where each node in V is a processor and each edge in E means that the two nodes incident on the edge can directly send each other messages at the transport level. Two nodes that have an edge between them are called *neighbors*. A simple broadcast protocol has a processor initiate the broadcast of m by sending m to all if its neighbors. Similarly, a node that receives m for the first time sends m to all of its neighbors except for the one which forwarded it m. This technique is commonly called *flooding* [1]. Depending on the superimposed graph structure, a node may be sent more than one copy of m. We call the number of messages sent in the reliable broadcast of a single m the *message overhead* of the broadcast protocol. For flooding, the message overhead is between one and two times the number of edges in the superimposed graph.

The most common graph that is superimposed is a spanning tree (for example, [10, 24]). Spanning trees are attractive when one wishes to minimize the number of messages: in failure-free runs, each processor receives exactly one message per broadcast and so the message overhead is $|V| - 1$. Their drawback is that when failures occur, a new spanning tree needs to be computed and disseminated to the surviving processors. This is because a tree can be disconnected by the removal of any internal (*i.e.* nonleaf) node or any link.

If a graph more richly connected than a tree is superimposed, then not all sets of link and internal node failures will disconnect the graph. Hence, if a detected and persistent failure occurs, any reconfiguration—that is, the computation of a new superimposing graph—can be done while the original superimposed graph is still used to flood messages. Doing so lessens the impact of the failure.

One example of the use of a graph more richly connected than a tree is discussed in [11]. In this work, they show how a hypercube graph can be used instead of a tree to disseminate information for purposes of garbage collection. It turns out that a hypercube is a Harary graph that is three-connected.

A more theoretical example of the use of a more richly connected graph than a tree is given by Liestman [18]. The problem being addressed in this work is, in some ways, similar to the problem we address. Like our work, they are interested in fault-tolerant broadcasting. And, like us, they wish to have a low message overhead. The models, however, are very different. They consider only link failures while we consider both link and node failures. They assume that a fixed unit of time elapses between a message being sent and it being delivered. They are concerned with attaining a minimum broadcast delivery time while we are not. And, the graphs that they superimpose are much more complex to generate as compared to Harary graphs. However, it turns out that some of the graphs they construct are also Harary graph.

Similarly, the previous work discussed in the survey paper [21] on fault-tolerant broadcasting and gossiping is, on the surface, similar to the work reported here. The underlying models and the goals, however, are different from ours. Our work is about what kind of communication graphs to superimpose such that flooding on such a graph masks a certain number of failures while sending the minimum number of messages. We also consider how the reliability degrades with respect to the graph structure when the number of failures exceeds what can be masked. Those earlier work, on the other hand, assumes messages transmitted or exchanged through a series of calls over the graph. The main goal is to compute the minimum time or its upper bound to complete a broadcast in the presence of a fixed number of failures, given different communication modes and failure models. The minimum number of calls that have to take place in order for a broadcast message to reach all processes is not always a metric of interest.

The utility of the graphs described by Harary in [15] for the purposes of reducing the probability of disconnection was originally examined in [25]. This work, however, was concerned with rules for laying out wide-area networks to minimize the cost of installing lines while still maintaining a desired connectivity. It is otherwise unrelated to the work described in this paper.

3 Gossip Protocols

Gossip protocols, which were first developed for replicated database consistency management in the Xerox Corporate Internet [9], have been built to implement not only reliable multicast [4, 12] but also failure detection [23] and garbage collection [13]. Nearly all gossip protocols have been developed assuming the failure model of processor crashes and message links failures that lead to dropped messages. Coordinated failures, such as the failure of a broadcast bus-based local area, are not usually considered. Such failures can only be masked by having redundant local area networks that fail independently (see, for example, [2, 5, 7]). And, they are not usually discussed in the context of synchronous versus asynchronous or timed asynchronous models [8, 14]. Like earlier work in gossip protocols, we do not consider coordinated failures or the question of how synchronous the environment must be to ensure that these protocols terminate in all runs.

Gossip protocols have the following three features:
1. *Scalability* Their performance does not rapidly degrade as the number of processors grow. Each processor sends a fixed number of messages that is independent of the number of processors. And, each processor runs a simple algorithm that is based on slowly-changing information. In general, a processor needs to know the identity of the other processors on its (local-area or small wide-area) network and a few constants. Hence, as long as the stability of the physical network does not degrade as the number of processors grow, then gossip is scalable.
2. *Adaptability* It is not hard to add or remove processors in a network. In both cases, it can be done using the gossip protocol itself.
3. *Graceful Degradation* For many reliable broadcast protocols, there is a value f such that if there are no more than f failures (as defined by a failure model) then the protocol will function correctly. If there are $f + 1$ failures, however, then the protocol may not function correctly. The reliability of the protocol is then equal to the probability that no more than f failures occur. Computing this probability, however, may be hard to do and the computation may be based on values that are hard to measure. Hence, it is advantageous to have a protocol whose probability of functioning correctly does not drop rapidly as the number of failures increases past f. Such a protocol is said to *degrade gracefully*. One can build gossip protocols whose reliability is rather insensitive to f.

There are many variations of gossip protocols within the two approaches mentioned in Section 1. Below is one variation of rumor mongering. This is the protocol that is used in [16] (specifically, $F = 1$ and the number of hops a gossip message can travel is fixed) and is called *blind counter rumor mongering* in [9]:

```
initiate broadcast of m:
   send m to B_initial neighbors
when (p receives a message m from q)
   if (p has received m no more than F times)
      send m to B randomly chosen neighbors that
         p does not know have already seen m;
```

A gossip protocol that runs over a local-area network or a small wide-area network effectively assumes that the communications graph is a clique. Therefore, the neighbors of a processor would be the other processors in the network. A processor knows that another processor q has already received m if it has previously received m from q. Processors can more accurately determine whether a processor has already received m by having m carry a set of processor identifiers. A processor adds its own identifier to the set of identifiers before forwarding m, and the union of the sets it has received on copies of m identify the processors that it knows have already received m. Henceforth in this paper, the gossip protocol we discuss is this specific version of rumor mongering that uses this more accurate method.

Since a processor selects it partners randomly, there is some chance that a message may not reach all processors even when there are no failures. However, in a clique, such probability is small [22]. Therefore, the reliability of gossip protocols is considered to be high. This is not the case when the connectivity of the network is not uniform, though. It has been shown that when the network has a hierarchical structure, a gossip message can fail to spread outside a group of processors [19].

It is often very difficult to obtain an analytical expression to describe the behavior of a gossip protocol. Often, the best one can get are equations that describe asymptotic behavior for simple network topologies. For more complex topologies or protocols, one almost always resorts to simulations. Our simulations show that the reliability does not drop too much for small values of f because a processor can receive messages from many processors. Also, the reliability of gossip rapidly increases with $B \cdot F$. In general, for a given value of $B \cdot F$, higher reliability is obtained by having a larger F (and therefore a smaller B). This is because a processor will have a more accurate idea of which processors already have m the later it forward m.

Our simulation results also show that, as expected, the average number of messages sent per broadcast is bounded by $BF(n-f)$. For a given value of $F \cdot B$, fewer messages are sent for larger F. In this case, some processors learn that most other processors have already received the broadcast by the time they receive m $F-1$ times, and so do not forward m to B processors.

The left hand graph of Figure 1 illustrates how gracefully the measured reliability of gossip degrades as a function of f (the right hand graph is discussed in Section 4.3). This figure was generated using simulation with 10,000 broadcasts done for each value of f and having the f processors crash at the beginning of each run. The number of processors n is 32, $B = 4$, and $F = 3$. The measured reliability is 0.9999 for $f = 3$ crashed processors and is 0.869 for $f = 16$ crashed processors. Notice that the measured reliability is not strictly decreasing with f; when f is close to n, only a few processors need to receive the message for the broadcast to be successful. Indeed, when $f = n - 1$, gossip trivially has a reliability of 1.

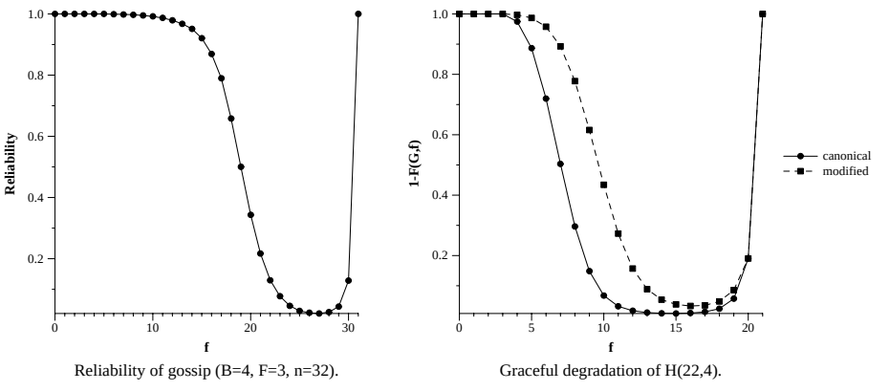


Fig. 1. Reliability of gossip and of Harary Graph flooding.

4 Harary Graphs

In this section, we discuss the approach of imposing a Harary graph of certain connectivity on a network of processors and having each processor flood over that graph. The graph will be connected enough to ensure that the reliability of the flooding protocol will be acceptably large.

4.1 Properties of Harary Graphs

A Harary graph is an $n-$node graph that satisfies the following three properties:

1. It is t-node connected. The removal of any subset of $t-1$ nodes will not disconnect the graph, but there are subsets of t nodes whose removal disconnects the graph.
2. It is t-link connected. The removal of any subset of $t-1$ links will not disconnect the graph, but there are subsets of t links whose removal disconnects the graph.
3. It is link minimal. The removal of any link will reduce the link connectivity (and therefore the node connectivity) of the graph.

Let $H_{n,t}$ denote the set of Harary graphs that contains n nodes and has a link and a node connectivity of t. For example, $H_{n,1}$ is the set of all $n-$node trees, and $H_{n,2}$ is the set of all $n-$node circuits. Figures 2 show one graph in $H_{7,3}$, two graphs in $H_{8,3}$ and one graph in $H_{8,4}$.

Harary gave an algorithm for the construction of a graph in $H_{n,t}$ for any value of n and $t < n$. [15] We denote this graph as $H^c_{n,t}$ and call it the *canonical Harary graph*. The algorithm is as follows:

$H^c_{n,1}$: The tree with edges $\forall i : 0 \leq i < n-1 : (i, i+1)$.

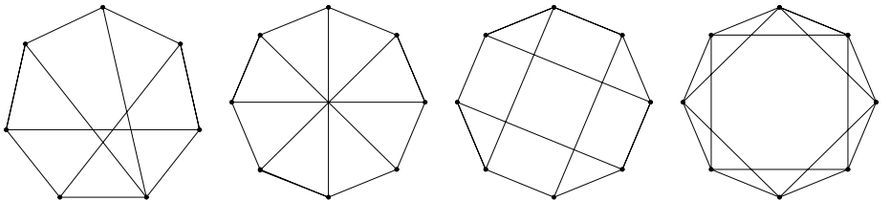


Fig. 2. A graph of $H_{7,3}$ two of $H_{8,3}$, and one of $H_{8,4}$.

$H^c_{n,2}$: The circuit with edges $\forall i : 0 \leq i < n : (i, i+1 \bmod n)$.
$H^c_{n,t}$ for $t > 2$ and even: First construct $H^c_{n,2}$. Then, for each value of $m : 2 \leq m \leq t/2$ add edges (i, j) where $|i - j| = m \bmod n$.
$H^c_{n,t}$ for $t > 2$ and odd: First construct $H^c_{n,t-1}$. Then, connect all pairs (i, j) of nodes such that $j - i = \lfloor n/2 \rfloor$.

For most values of n and t, there are more than one graph in the set of $H_{n,t}$ graphs. All but the third graph in Figure 2 are canonical Harary graphs, while the third graph in Figure 2 is not (it is the unit cube).

It is not hard to see why Harary graphs are link-minimal among all t-connected graphs. For any graph G, the node connectivity $\kappa(G)$, link connectivity $\lambda(G)$, and minimum degree $\delta(G)$ are related:

$$\kappa(G) \leq \lambda(G) \leq \delta(G) \tag{1}$$

Harary showed that $\delta(G)$ is bounded by $\lceil 2\ell/n \rceil$, where ℓ is the number of links [15]. Therefore, to have t node or link connectivity, the number of links has to be at least $\lceil nt/2 \rceil$. If G is a regular graph (that is, all of the nodes have the same degree), then $\kappa(G) = \lambda(G) = \delta(G) = t$. A regular graph with $nt/2$ links is thus link-minimal among all graphs with t node or link connectivity. Harary graphs are such graphs when t is even or when n is even and t is odd.

When both n and t are odd, Harary graphs are not regular graphs because there is no regular graph of odd degree with an odd number of nodes. Rather, there are $n - 1$ nodes of degree t and one node of degree $t + 1$. The number of links is $\lceil nt/2 \rceil$. They are link-minimal because removing any link will result in at least one node having its node degree reduced from t to $t - 1$.

4.2 Overhead and Reliability of Flooding on Harary Graphs

When n or t is even and assuming no failures, the overhead of flooding over a Harary graph is bounded from above by $n(t-1)+1$: the processor that starts the broadcast sends t messages and all the other processors send no more than $t - 1$ messages. When n and t are odd one more message can be sent because the graph is not regular.

As long as the communications graph remains connected, a flooding protocol is guaranteed to be reliable. Hence, we characterize reliability as the probability of $H_{n,t}$ disconnecting. We first consider a failure model that includes the failure of nodes — that is, processor crashes, and the failure of links — that is, message channels that can drop messages due to congestion or physical faults. We then consider only node failures.

For local-area networks and small wide-area networks, one normally assumes that each link has the same probability p_ℓ of failing and each node has the same probability p_n of failing, and that all failures are independent of each other. We do so as well.

Since $H_{n,t}$ is both t-connected and t-link connected, one can compute an upper bound on the probability of $H_{n,t}$ being disconnected: the disconnection of $H_{n,t}$ requires x node failures and y link failures such that $x + y \geq t$. [3] The reliability r is thus bounded from below by the following:

$$r \geq 1 - \sum_{x=0}^{n-2} \binom{n}{x} p_n^x (1-p_n)^{n-x} \sum_{y=\max(t-x,0)}^{\ell} \binom{\ell}{y} p_\ell^y (1-p_\ell)^{\ell-y} \qquad (2)$$

where ℓ is the number of links in $H_{n,t} : \ell = \lceil nt/2 \rceil$. This formula, however, is conservative since it assumes that *all* such failures of nodes and links disconnect the graph. For example, consider $H_{4,2}$ and assume that $p_n = 0.99$ and $p_\ell = 0.999$ (that is, each processor is crashed approximately 14 minutes a day and a link is faulty approximately 1.4 minutes a day). The above formula computes a lower bound on the reliability of 0.9993. If we instead examine all of the failures that disconnect $H_{4,2}$ and sum the probabilities of each of these cases happening, then we obtain an actual reliability of 0.9997.

In general, computing the probability of a graph disconnecting given individual node and link failure probabilities is hard [6], and so using Equation 2 is the only practical method we know of for computing the reliability of flooding on $H_{n,t}$. But, if one assumes that links do not fail then one can compute a more accurate value of the reliability. Note that assuming no link failures may not be an unreasonable assumption. A vast percentage of the link failures in small wide-area networks and local area networks are associated with congestion at either routers or individual processors. Hence, link failures rarely endure and can often be masked by using a simple acknowledgment and retransmission scheme.

Consider the following metric:

Definition 1. Given a graph $G \in H_{n,t}$, the *fragility* $F(G)$ of G is the fraction of subsets of t nodes whose removal disconnects G.

For example, in the left graph of Figure 3, 187 of the 7,315 subsets of 4 nodes are cutsets of the graph, and so the graph has a fragility of 0.0256. The graph on the right, also a member of $H_{22,4}$, has a fragility of $22/7{,}315 = 0.0030$.

Since here we consider node failures only, p_ℓ is zero. We further assume that p_n is small. Thus, the probability of G disconnecting can be estimated as the probability of t nodes failing weighted by $F(G)$: $F(G)p_n^t(1-p_n)^{n-t}$. So,

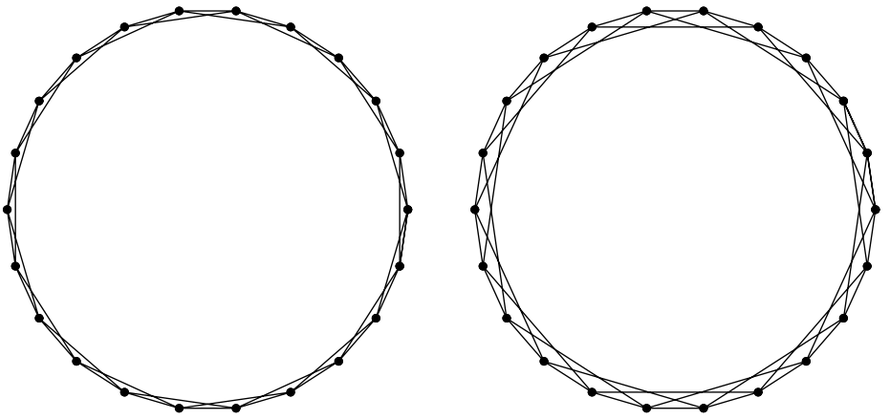


Fig. 3. Two graphs of $H_{22,4}$.

$$r \leq 1 - F(G)p_n^t(1-p_n)^{n-t} \tag{3}$$

This is an upper bound on r because it does not consider disconnections arising from more than t nodes failing. But, when p_n is small the contribution due to more than t nodes failing is small.

Assume that each processor is crashed for five minutes a day, and so $p_n = 0.0035$. From Equations 2 and 3 and setting $p_\ell = 0$ we compute $0.999998989 \leq r \leq 0.999999974$ for flooding in the left graph of Figure 3. In fact, the true reliability r computed by enumerating all the disconnecting cutsets and summing their probabilities of occurring is 0.999999973. Similarly, for the right-hand graph of Figure 3, we compute $0.999998989 \leq r \leq 0.999999997$ and r is 0.999999997.

4.3 Graceful Degradation

Still assuming that $p_\ell = 0$, one can extend the notion of fragility to compute how gracefully reliability degrades in flooding on a Harary graph.

Definition 2. Given a graph $G \in H_{n,t}$, $F(G,f)$ is the fraction of subsets of f nodes whose removal disconnects G.

Hence, $F(G,t) = F(G)$ and $F(G,f) = 0$ for $0 \leq f < t$ or $n-1 \leq f \leq n$. On the condition of any subset of f nodes having failed, the graph will remain connected with the probability of $1 - F(G,f)$. Thus, we can use $1 - F(G,f)$ as a way to characterize the reliability of flooding on a Harary graph G. This is much like the way we measured the reliability of gossip as a function of f as illustrated in Figure 1. There, the reliability was measured after f nodes have crashed.

The graphs of $1 - F(G, f)$ for the two graphs of Figure 3 is shown in the right hand of Figure 1. As can be seen, the less fragile graph also degrades more gracefully than the canonical graph.

4.4 Bounds on Fragility

Since the upper bound on reliability and how gracefully reliability degrades depend on the fragility of a Harary graph, we examine some fragility properties of canonical Harary graphs. We show that canonical Harary graphs can be significantly more fragile than some other families of Harary graphs, and describe how to construct these less fragile graphs.

We define a $t-$cutset of a graph G to be a set of t nodes of G whose removal disconnects G. And, given a subgraph S of G, we define the joint neighbors of S to be those nodes in $G - S$ that are a neighbor of a node in S. The two ideas are related: if S is connected, A are the joint neighbors of S, and $A \cup S \subset G$, then A is a $|A|-$cutset of G.

$H_{n,n-1}$ is an $n-$clique and so $F(H_{n,n-1}, f) = 0$ for all f between 0 and n. Hence, in the following we assume that $t < n - 1$.

Theorem 3. *For even $t > 2$ and $n \geq t + 2$, the number of subsets of t nodes that disconnects $H^c_{n,t}$ is $n(n - t - 1)/2$.*

We can construct a member of $H_{n,t}$ for even t and $n > 2t$ that are less fragile than $H^c_{n,t}$. These graphs, which we call *modified* Harary graphs and denote with $H^m_{n,t}$, have n distinct $t-$cutsets. The right hand graph of Figure 3 is $H^m_{22,4}$, while the left hand graph is $H^c_{22,4}$. An algorithm for constructing $H^m_{n,t}$ is as follows:

$H^m_{n,t}$: First construct $H^c_{n,2}$. Then, for each value of m: $2 \leq m \leq t/2$ add edges (i, j) where $|i - j| = (m + 1) \bmod n$.

Theorem 4. *$H^m_{n,t}$ is in $H_{n,t}$. When $n > 2t$, $H^m_{n,t}$ has n $t-$cutsets.*

It is not hard to find examples of Harary graphs that have fewer than n $t-$cutsets.[20] For arbitrary n and t, however, it remains an open problem of what is the minimum number of cutsets of size t for graphs in $H_{n,t}$.

Canonical Harary graphs have a better fragility when t is odd:, which

Theorem 5. *For $H^c_{n,t}$ with even $n > 6$ and $t = 3$, there are n cutsets of size t.*

Theorem 6. *For $H^c_{n,t}$ with odd $n > 7$ and $t = 3$, there are n cutsets of size t.*

Theorem 7. *For $H^c_{n,t}$ with even $n \geq t + 2$ and odd $t \geq 5$, there are n cutsets of size t. For $H^c_{n,t}$ with odd $n > t + 2$ and odd $t \geq 5$, there are n cutsets of size t.*

When n and t are both odd, only $n - 1$ nodes have t neighbors; one node has $t + 1$ neighbors. Hence, it may be surprising that $H^c_{n,t}$ for odd n and t has n $t-$cutsets. It is not hard to construct graphs in $H_{n,t}$ for odd n and t that have

$n-1$ $t-$cutsets. that has 8 $3-$cutsets. The improvement in fragility for such graphs, however, is small.

For a fixed $t > 2$, the fragility $F(H^c_{n,t})$ decreases with increasing n. This is because the number of cutsets of size t grows either linearly or quadratically with n, while $\binom{n}{t}$ is of $O(n^t)$. Similarly, $F(H^m_{n,t})$ decreases with increasing n. The same observations hold for $F(H^c_{n,t})$ and $F(H^m_{n,t})$ when n is fixed and t is increased.

5 Comparing Harary Graph-Based Flooding and Gossip

A simple comparison of flooding over a Harary graph and gossip based on the following four criteria indicates that each has its own strengths:

1. *Scalability* For a small wide-area network and local area network, the two protocols are equivalent. Both protocols run simple algorithms that are based on slowly-changing information. Both require a processor to know the identity of the other processors on its (small wide-area or local area) network and a small amount of constant information (the rule used to identify neighbors versus the constants B and F). And, in both protocols the number of messages that each processor sends is independent of n.

2. *Adaptability* The flooding protocol requires the processors to use the same Harary graph. Since each processor independently determines its neighbors, it might appear that gossip is more adaptable than flooding over a Harary graph. From a practical point of view, though, we expect that they are similar in adaptability. Given the controlled environment of local-area networks and small wide-area networks, it is not hard to bound from below with equivalent reliabilities the time it takes for each protocol to terminate. Then, one can use reliable broadcast based on the old set of processors to disseminate the new set of processors. In addition, adding or removing a single processor causes only t processors to change their neighbors in Harary graph flooding. If t is small, then a simple (and non-scalable) agreement protocol can be used to change the set of processors.

3. *Graceful Degradation* Gossip degrades more gracefully than Harary graph flooding. Figure 4 illustrates this for $n = 22$. The Harary graphs used here are the $H^c_{22,4}$ and $H^m_{22,4}$, and the gossip protocols use $B = 3, F = 2$ and $B = 2, F = 2$, both with $B_{initial} = B$. We compare the degradation of reliability of these protocols because they have similar message overheads. Note that while the Harary graph flooding yields a *higher* reliability for small f even when $f > t$, both gossip protocols have a higher reliability for $f > 10$.

4. *Message Overhead* Since $H_{n,t}$ has a minimum number of links while remaining t-connected, it is not surprising that Harary graph flooding sends fewer messages than gossip. For example, given $n = 32$, flooding on $H_{32,4}$ and gossip with $B = 4$ and $F = 3$ provide similar reliabilities given processors that are crashed for five minutes a day. Gossip sends roughly $BF/(t-1) = 4$ times as many messages as Harary graph flooding.

To make a more detailed comparison of message overhead, we evaluated the performance of gossip and Harary graph flooding using the *ns* simulator [27].

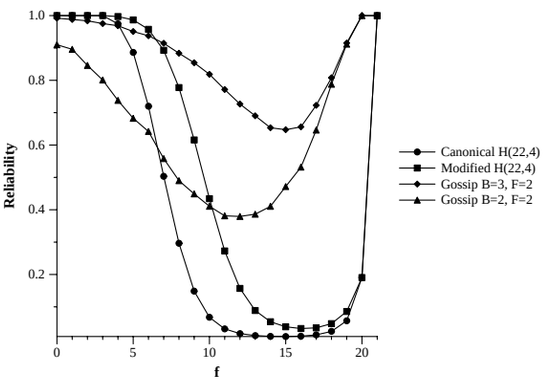


Fig. 4. Graceful degradation of gossip and flooding on $H_{22,4}$

We simulated Ethernet-based networks. One of the networks we considered was a LAN of a single Ethernet, where there are 32 processors. The other was a small WAN of three Ethernets pairwise connected, where each Ethernet has 21 processors, one of them also acting like a router.

For the single LAN we imposed the $H^m_{32,4}$ graph on the processors, and for the small WAN $H^m_{63,4}$ was imposed where processors 0, 21 and 42 were the routers connecting the three LANs. Thus on each LAN, the router has two neighbors on a different LAN and another processor has one neighbor outside of the LAN. We compared flooding on these Harary graphs with gossip where $B = 4, F = 3$, and $B_{initial} = B$.

We obtained the properties of an Ethernet based on those of a common Ethernet for LANs. For the single LAN, we assume a bandwidth of 10 Mbps, and for the small WAN, we assumed that the links between routers of the LANs have a bandwidth of 1 Mbps and a delay of 10 ms. The ns simulator followed the Ethernet specifications and provided the low-level details. Each message was contained in a 1K-byte packet. We did not consider failures in these simulations. All the results were computed as the average of 100 broadcasts.

We found that gossip initially delivers broadcasts faster. However, it takes longer for the last few processors to receive the message. Therefore, it takes longer for gossip to complete a broadcast than for Harary graph flooding. For both protocols, there were a fair amount of collisions because both protocols make intensive use of the network. These collisions increased the completion time for both protocols. We also found that the packt flow of Harary graph flooding diminishes much more quickly than that of gossip.

In the small WAN, Harary graph flooding imposes a much smaller load on the routers than gossip does. We measured the number of packets that were sent across the LANs. With gossip, an average of 169 packets were sent between each pair of LANs. With Harary graph flooding, only 6 packets went across

because processors on each LAN have a total of three neighbors on another LAN. This kind of overloading of routers with redundant messages happens when the network topology is not taken into account [19, 23].

A more detailed discussion of these simulation results can be found in [20].

6 Conclusions

Gossip protocols are often advertised as being attractive because their simplicity and their use of randomization makes them scalable and adaptable. Furthermore, their reliability degrades gracefully with respect to the actual number of failures f. We believe that, while these advertised attractions are valid, Harary graph flooding also provides most of these attractions with a substantially lower message overhead. Furthermore, Harary graph flooding appears to be faster than gossip for broadcast-bus networks. Hence, for local-area networks and small wide-area networks, the only benefit of gossip is that its reliability more gracefully degrades than Harary graph flooding.

This remaining advantage of gossip, however, should be considered carefully. While it is true that gossip more gracefully degrades than Harary graph flooding, the reliability of flooding over $H^m_{n,t}$ decreases only slightly for values of f slightly larger than t. Hence, Harary graph flooding provides some latitude in computing t. And, both suffer from reduced reliability as f increases further. One improves graceful degradation of gossip by increasing F (or, to a slightly less degree, B). Doing so increases the message overhead and network congestion, and therefore the protocol completion time.

References

1. G. R. Andrews. *Concurrent programming: Principles and practice*, Benjamin/Cummings, 1991.
2. Ö. Babaoğlu. On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 5(4):394–416 (November 1987).
3. L. W. Beineke and F. Harary. The connectivity function of a graph. *Mathematika* 14:197–202 (1967).
4. K. Birman *et al.* Bimodal multicast. *ACM Transactions on Computer Systems* 17(2):41–88 (May 1999).
5. M. Clegg and K. Marzullo. A Low-cost processor group membership protocol for a hard real-time distributed system. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997, pp. 90–98.
6. C. J. Colbourn. *The combinatorics of network reliability*, Oxford University Press, 1987.
7. F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems* 2(3):195–212 (September 1990).
8. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems* 10(6):642–657 (June 1999).

9. A. Demers *et al.* Epidemic algorithms for replicated database maintenance. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, 10–12 August 1987, pp. 1–12.
10. S. Floyd *et al.* A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking* 5(6):784–803 (December 1997).
11. R. Friedman, S. Manor and K. Guo. Scalable stability detection using logical hypercube. Technion Department of Computer Science Technical Report 0960, May 1999.
12. R. A. Golding and D. E. Long. The performance of weak-consistency replication protocols. University of California Santa Cruz, Computer Research Laboratory Technical Report UCSC-CRL-92-30, July 1992.
13. K. Guo *et al.* GSGC: an efficient gossip-style garbage collection scheme for scalable reliable multicast. Cornell University, Department of Computer Science Technical Report TR-97-1656, December 3 1997.
14. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems* (S. Mullender, editor), ACM Press, 1993.
15. F. Harary. The maximum connectivity of a graph. In *Proceedings of the National Academy of Sciences*, 48:1142–1146 (1962).
16. M. G. Hayden and K. P. Birman. Probabilistic broadcast. Cornell University, Department of Computer Science Technical Report TR-96-1606, September 1996.
17. T. Abdelzaher, A. Shaikh, F. Jahanian and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium*, 1996, pp.250–259.
18. A. Liestman. Fault-tolerant broadcast graphs. *Networks* 15(2):159–171 (Summer 1985).
19. M. J. Lin and K. Marzullo. Directional gossip: gossip in a wide area network. In *Proceedings of the Third European Dependable Computing Conference*, Prague, Czech Republic, September 1999 (Springer-Verlag LNCS 1667), pp. 364–379.
20. M. J. Lin, K. Marzullo and S. Masini. Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small network. University of California San Diego Department of Computer Science Technical Report CS99-0637, November 1999.
21. A. Pelc. Fault-tolerant broadcast and gossiping in communication networks. *Networks* 28(3):143–156 (October 1996).
22. B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223 (February 1987).
23. R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998, pp. 55-70.
24. Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems* 2(2):145–154 (May 1984).
25. Amitabh Shah. Exploring Trade-offs in the Design of Fault-Tolerant Distributed Databases. Ph.D. dissertation, Cornell University Department of Computer Science, August 1990.
26. D. B. Terry *et al.* Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 3–6.
27. Network Simulator. http://www-mash.cs.berkeley.edu/ns.

Thrifty Generic Broadcast*

Marcos Kawazoe Aguilera[1], Carole Delporte-Gallet[2], Hugues Fauconnier[2], and Sam Toueg[1]

[1] DIX, École Polytechnique, 91128 Palaiseau Cedex, France,
aguilera,sam@lix.polytechnique.fr
[2] LIAFA, Université D. Diderot, 2 Place Jussieu, 75251 Paris Cedex 05, France,
cd,hf@liafa.jussieu.fr

Abstract. We consider the problem of generic broadcast in asynchronous systems with crashes, a problem that was first studied in [12]. Roughly speaking, given a "conflict" relation on the set of messages, generic broadcast ensures that any two messages that conflict are delivered in the same order; messages that do not conflict may be delivered in different order. In this paper, we define what it means for an implementation generic broadcast to be "thrifty", and give corresponding implementations that are optimal in terms of resiliency. We also give an interesting application of our results regarding the implementation of atomic broadcast.

1 Introduction

Atomic broadcast is a well-known building block of fault-tolerant distributed applications (e.g., see [7, 4, 9, 8, 10, 3, 2]). Informally, this communication primitive ensures that *all* messages broadcast are delivered in the same order. In a recent paper, Pedone and Schiper noted that for some applications some messages do not "conflict" with each other, and hence they can be delivered by different processes in different orders [12]. For such applications, the broadcast communication primitive does not need to order all messages; it must order only the conflicting ones. An example given in [12] consists of *read* and *write* messages broadcast to replicated servers, where read messages do not conflict with each other, and hence do not have to be ordered. Intuitively, one may want to avoid ordering the delivery of messages unless it is really necessary: such ordering may be expensive, or even impossible unless one uses oracles such as as failure detectors, and these can be unreliable.

In view of the above, Pedone and Schiper proposed a generalized version of atomic broadcast, called *generic broadcast*. Informally, given any *conflict* relation defined over the set of messages, *if* two messages m and m' conflict, then generic broadcast ensures that they are delivered in the same order.[1] Messages that do not conflict are not required to be ordered. Note that if the conflict relation includes all the pairs of messages, generic broadcast coincides with atomic

* Research partially supported by NSF grants CCR-9711403.

[1] The conflict relation is a parameter of generic broadcast. We assume that it is symmetric and non-reflexive.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 268–282, 2000.

broadcast. On the other hand, if the conflict relation is empty, generic broadcast reduces to reliable broadcast.

How can one implement a generic broadcast primitive? A trivial way is to use atomic broadcast to broadcast *every* message that we want to gbroadcast.[2] This ensures that all messages are ordered, including non-conflicting ones. Such an implementation is unsatisfactory, and goes against the motivation for introducing generic broadcast in the first place. To avoid this trivial implementation, and in order to characterize "good" implementations, Pedone and Schiper introduced the notion of *strictness*. Roughly speaking, an implementation of generic broadcast is strict if it has at least one execution in which two processes deliver two non-conflicting messages in a different order. The notion of strictness is intended to capture the intuitive idea that the total order delivery of messages has a cost, and this cost should be paid only when necessary. As Pedone and Schiper point out in [13], however, the strictness requirement is not sufficient to characterize good implementations of generic broadcast. Intuitively, this is because there is a strict implementation that first orders *all* the messages, including non-conflicting ones, and then selects two non-conflicting messages and delivers them in different orders. Even though such an implementation is strict, it goes against the motivation behind generic broadcast.

In this paper, we reconsider the question of what it means for an implementation of generic broadcast to be good, and we propose new definitions. We first note that in asynchronous systems with crash failures (the systems considered in [12] and here), generic broadcast cannot be implemented without the help of an "oracle" that can be used to order the delivery of messages that conflict. This oracle could be a "box" that solves atomic broadcast or consensus; or it could be a failure detector that can be used to implement such a box. In the first case, this oracle is expensive; in the second case, it can be unreliable and its mistakes can slow down the delivery of messages.[3] In either case, one should avoid the use of the oracle whenever possible. Thus, a good implementation of generic broadcast is one that takes advantage of the fact that only conflicting messages need to be ordered, and uses its oracle only when there are conflicting messages that are actually broadcast.

This leads us to the following definition. Roughly speaking, an implementation of generic broadcast is *non-trivial w.r.t. an oracle*, if it satisfies the following property: if all the messages that are actually broadcast do not conflict with each other, then the oracle is never used. A non-trivial implementation, however, is still unsatisfactory: even in a run where there is *only one* broadcast that conflicts with a previous one, such an implementation is allowed use its oracle an

[2] Henceforth, *gbroadcast* and *gdeliver* are the two primitives associated with generic broadcast. Similarly, *abroadcast* and *adeliver* are associated with atomic broadcast.

[3] Even though one can implement failure detectors that are fairly accurate in practice [14, 6], they may have "bad" periods of time when they make too many mistakes to be useful. For example, from [5] there is an atomic broadcast algorithm that never deliver messages out of order, but message delivery is delayed if/when the algorithm happens to rely on the failure detector during one of its bad periods.

unlimited number of times. This motivates our second definition. An implementation of generic broadcast is *thrifty w.r.t. an oracle* if it is non-trivial and it also satisfies the following property: if there is a time after which the messages broadcast do not conflict with each other, then there is a time after which the oracle is not used. It is easy to see that non-trivial implementations and thrifty ones are necessarily strict in the sense of [12].

In this paper, we consider implementations of generic broadcast that use atomic broadcast as the oracle. Atomic broadcast is a natural oracle for the task of totally ordering conflicting messages. Furthermore, any implementation that is thrifty w.r.t. atomic broadcast can be transformed into an implementation that is thrifty w.r.t. consensus. It can also be transformed into an implementation that is thrifty w.r.t. $\diamond\mathcal{S}$, the weakest failure detector that can be used to solve generic broadcast (this last transformation assumes that a majority of processes is correct).

We present two implementations of generic broadcast: one is non-trivial and the other is thrifty. The non-trivial implementation is simple and illustrates some of our basic techniques; the thrifty implementation is more complex and builds upon the simple implementation. Both implementations work for asynchronous systems with n processes where up to $f < n/2$ may crash, which is optimal. Since both implementations are also strict, this improves on the resiliency of the strict implementation given in [12] which tolerates up to $f < n/3$ crashes.

We continue the paper with an interesting use of thrifty implementations of generic broadcast. Specifically, we show how they can be used to derive "sparing" implementations of atomic broadcast, as we now explain. First note that in asynchronous systems with failures, any implementation of atomic broadcast requires the use of an external oracle, and (just as with generic broadcast) it is better to avoid relying on this oracle whenever possible. For example, if the oracle is a failure detector, relying on this oracle during one of its "bad" period can delay the delivery of messages. So we would like an implementation of atomic broadcast that uses the oracle sparingly. How can we do so?

Suppose a process atomically broadcast m and then m'. No oracle is needed to ensure that m and m' are delivered in the same order everywhere: FIFO order can be easily enforced with sequence numbers assigned by the sender. Similarly, suppose two atomic broadcast messages happen to be causally related[4], e.g., m is adelivered by a process before it abroadcasts m'. Then, we can order the delivery of m and m' without any oracle (this can be done with message piggybacking or "vector clocks"; see for example [10]). Thus, an implementation of atomic broadcast can reduce its reliance on the oracle, by restricting its use to the ordering of broadcast messages that are *concurrent*. We say that an implementation of atomic broadcast is *sparing w.r.t. an oracle*, if it satisfies the following property: If there is a time after which the messages broadcast are pairwise causally related, then there is a time after which the oracle is not used.

[4] We say that two messages are causally related or concurrent, if their broadcast events are causally related or concurrent, respectively, in the sense of [11, 10].

We conclude the paper by showing how to transform *any* implementation of atomic broadcast that uses some oracle, into one that is sparing w.r.t the same oracle. To do so, we use a thrifty implementation of generic broadcast and vector clocks.

As a final remark, note that Pedone and Schiper use message latency as a way to evaluate the efficiency of generic broadcast implementations. In "good" runs with no failures and no conflicting messages, their generic broadcast algorithm ensures that every message is delivered within 2δ (assuming δ is the maximum message delay). In this paper, our focus was not on optimizing the latency of messages in these good runs, but rather on reducing the dependency on the oracle whenever possible. These two goals, however, are not incompatible. In fact, we can modify our thrifty implementations of generic broadcast to also achieve a small message latency in good runs. Specifically, we have an implementation that assumes $f < n/3$ and ensures a message delivery within 2δ in such runs (as in [12]). We also have an implementation that works for $f < n/2$ and ensures message delivery within 3δ in good runs. It is worth noting that *even in runs with failures and conflicting messages*, the message delivery times of 2δ and 3δ, respectively, are *eventually* achieved provided there is a time after which the messages broadcast are not conflicting.

In summary, this paper considers the problem of generic broadcast in asynchronous systems with crashes, a problem that was first studied in [12]. We first propose alternative definitions of "good" implementations of generic broadcast (the previous definition in terms of "strictness" had some drawbacks). Roughly speaking, we consider an implementation to be good if it does not rely on any oracle when the messages that are broadcast do not conflict. We then give two such implementations (with atomic broadcast as its oracle): one does not use the oracle in runs where no messages conflict, and the other one stops using the oracle if conflicting broadcasts cease. Both implementations are optimal in terms of resiliency; they tolerate up to $f < n/2$ process crashes (an improvement over [12]). We then use our results to give "sparing" implementations of atomic broadcast, i.e., implementations that stop using their oracle if concurrent broadcasts cease. Finally, we show how to transform any implementation of atomic broadcast into a sparing one.

In this extended abstract, we omit the proofs (they are given in the full paper [1]).

2 Informal Model

We consider *asynchronous* distributed systems. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range τ of the clock's ticks to be the set of natural numbers $\mathbf{N}$.

The system consists of a set of n *processes*, $\Pi = \{1, 2, \ldots, n\}$ and an oracle. Processes are connected with each other through reliable asynchronous communication channels. Up to f processes can fail by *crashing*. A failure pat-

tern indicates which processes crash, and when, during an execution. Formally, a *failure pattern* F is a function from τ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $crashed(F) = \bigcup_{t \in \tau} F(t)$ and $correct(F) = \Pi - crashed(F)$. If $p \in crashed(F)$ we say p *crashes (in* F*)* and if $p \in correct(F)$ we say p *is correct (in* F*)*.

A distributed algorithm $\mathcal{A}$ is a collection of n deterministic automata (one for each process in the system). The execution of $\mathcal{A}$ occurs in *steps* as follows. For every time $t \in \tau$, at most one process takes a step; moreover, every correct process takes an infinite number of steps. In each step, a process (1) may send a message to a process; (2) queries the oracle (the query may be $\perp$); (3) receives an answer from the oracle (possibly $\perp$); (4) receives a message (possibly $\perp$); and (5) changes state. We say that *a process uses the oracle at time* t if it performs a non-$\perp$ query at time t.

An oracle history H is a sequence of quadruples (p, t, i, o), where p is a process, t is a time (t is monotonically increasing in H), i is the query of p at time t, and o is the answer of the oracle to p at time t. We assume that if no process ever uses the oracle (all queries in H are $\perp$) then the oracle never gives any answer (all answers in H are $\perp$). An oracle $\mathcal{O}$ is function that takes a failure pattern F and returns a set $\mathcal{O}(F)$ of oracle histories[5]. Oracles of interest include failure detectors [5], an atomic broadcast black-box, and a consensus black-box. For example, an atomic broadcast black-box can be modeled as an oracle that accepts "broadcast(m)" queries, and outputs "deliver(m)" answers, where the queries/answers satisfy the usual specification of atomic broadcast (see Section 2.2).

2.1 Reliable broadcast

Intuitively, reliable broadcast ensures that processes agree on the set of messages that they deliver. More precisely, *reliable broadcast* is defined in terms of two primitives: $rbroadcast(m)$ and $rdeliver(m)$. We say that process p *broadcasts message* m if p invokes $rbroadcast(m)$. We assume that every broadcast message m includes the following fields: the identity of its sender, denoted $sender(m)$, and a sequence number, denoted $seq(m)$. These fields make every message unique. We say that q *delivers message* m if q returns from the invocation of $rdeliver(m)$. Primitives *rbroadcast* and *rdeliver* satisfy the following properties:[6]

Validity: If a correct process broadcasts a message m, then it eventually delivers m.

Uniform Agreement: If a process delivers a message m, then all correct processes eventually deliver m.

Uniform Integrity: For every message m, every process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

[5] We assume this set allows any process to make any query at any time.

[6] All the broadcast primitives that we define in this paper are *uniform* [10]. To abbreviate the notation, we drop the word "uniform" from the various broadcast types.

Validity and Uniform Agreement imply that if a correct process broadcasts a message m, then all correct processes eventually deliver m.

2.2 Atomic broadcast

Intuitively, atomic broadcast ensures that processes agree on the order they deliver messages. More precisely, *atomic broadcast* is defined in terms of primitives $abroadcast(m)$ and $adeliver(m)$ that must satisfy the Validity, Uniform Agreement and Uniform Integrity properties above, and the following property:

Uniform Total Order: If some process delivers message m before message m', then a process delivers m' only after it has delivered m.[7]

2.3 Generic broadcast

Generic broadcast is parametrized by a *conflict* relation (denoted $\sim$) defined over the set of messages; this relation is assumed to be symmetric and non-reflexive. Informally, generic broadcast ensures that if two messages m and m' conflict, then they are delivered in the same order. Messages that do not conflict are not required to be ordered. More precisely, *generic broadcast* is defined in terms of the conflict relation (given as a parameter) and two primitives: $gbroadcast(m)$ and $gdeliver(m)$ that must satisfy the Validity, Uniform Agreement and Uniform Integrity properties above, and the following property:

Uniform Generalized Order: If messages m and m' conflict and some process delivers m before m', then a process delivers m' only after it has delivered m.

If the conflict relation includes all the pairs of messages, generic broadcast coincides with atomic broadcast; if the conflict relation is empty, generic broadcast reduces to reliable broadcast.

3 Thrifty implementations

Let $\mathcal{A}$ be an implementation of generic broadcast that can use an oracle X, and let $Runs(\mathcal{A})$ be the set of runs of A. Let $gbcast\_msgs(r)$ be the set of messages gbroadcast in r and $gbcast\_msgs(r, [t, \infty))$ be the set of messages gbroadcast in r at or after time t.

Definition 1. *We say that $\mathcal{A}$ is* non-trivial w.r.t. oracle X *if, when no conflicting messages are* gbroadcast*, X is not used. More precisely:* $\forall r \in Runs(\mathcal{A})$, $[\forall m, m' \in gbcast\_msgs(r), m \not\sim m'] \Rightarrow X$ *is not used in* r.

Definition 2. *We say that $\mathcal{A}$ is* thrifty w.r.t. X *if it is non-trivial w.r.t. X and it guarantees the following property: if there is a time after which messages gbroadcast do not conflict with each other, then eventually X is no longer used. More precisely:* $\forall r \in Runs(\mathcal{A}), [\exists t, \forall m, m' \in gbcast\_msgs(r, [t, \infty)), m \not\sim m'] \Rightarrow \exists t', X$ *is not used in* r *after time* t'.

[7] In [10], Uniform Total Order is a weaker property.

4 A non-trivial implementation of generic broadcast

We now give a non-trivial implementation of generic broadcast for asynchronous systems with a majority of correct processes. The implementation, given in Figure 1, uses atomic broadcast as an oracle, and reliable broadcast as a subroutine (reliable broadcast can be easily implemented in asynchronous systems with process crashes without the use of oracles). In this implementation, $C(m)$ denotes the set $\{m\} \cup \{m' : m' \text{ conflicts with } m\}$.

To gbroadcast a message m, the basic idea is that processes go through two rounds of messages, and then the broadcaster p decides to either rbroadcast m (in which case the oracle is not used) or abroadcast m (in which case the oracle is used). More precisely, to gbroadcast a message m, p sends (m, FIRST) to all processes, where FIRST is a tag to distinguish different types of messages. When a process receives (m, FIRST), it adds m to its set *seen* of messages, and checks if m conflicts with any messages in *seen*. If it does, it sends $(m, \text{BAD}, \text{SECOND})$ to all processes; else, it sends $(m, \text{GOOD}, \text{SECOND})$. When a process receives a message of form $(m, *, \text{SECOND})$ from $n - f$ processes, it adds m to its *seen* set, and then checks if a majority of SECOND messages are GOOD, and if its *seen* set has no messages conflicting with m. If so, the process adds m to its set *possibleRB*, and then sends $(m, \textit{possibleRB} \cap C(m), \text{THIRD})$ to p — the process that gbroadcast m — where $\textit{possibleRB} \cap C(m)$ is the subset of messages in *possibleRB* that either conflict with m or is equal to m (note that $\textit{possibleRB} \cap C(m)$ can be empty, it can contain m, and it can contain messages distinct from m). When p receives messages of the form $(m, \textit{poss}, \text{THIRD})$ from $n-f$ processes, it checks if a majority of them has m it its *poss* set. If so, p rbroadcasts m; else, p abroadcasts m, together with the union of all *poss* sets received. When a process rdelivers m, it gdelivers m if it has not done so previously. When a process adelivers $(m, \textit{prec})$, it gdelivers all messages in *prec* (if it has not done so already), and then gdelivers m.

In this implementation, each process keeps two local variables: *seen* and *possibleRB*. The first one keeps the set of gbroadcast messages that the process has seen so far, and the second keeps the set of gbroadcast messages that are possibly reliably broadcast.

Theorem 1. *Consider an asynchronous system with a majority of correct processes ($n > 2f$). The algorithm in Figure 1 is a non-trivial implementation of generic broadcast that uses atomic broadcast as an oracle.*

Observation: In asynchronous systems with $n \leq 2f$, there are no non-trivial implementations of generic broadcast w.r.t. any oracle X.

5 A thrifty implementation of generic broadcast

We now give a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle. It works in asynchronous systems in which a majority of processes is correct. This implementation is given in Figure 2, and builds

```
1   For every process p:
2       initialization:
3           seen ← ∅; possibleRB ← ∅
4       to gbroadcast(m): send (m, FIRST) to all processes
5       upon receive(m, FIRST) from q:
6           seen ← seen ∪ {m}
7           if seen has no messages conflicting with m
8           then send (m, GOOD, SECOND) to all processes
9           else send (m, BAD, SECOND) to all processes
10      upon receive(m, *, SECOND) from n − f processes for the first time:
11          seen ← seen ∪ {m}
12          good ← {r : received (m, GOOD, SECOND) from r}
13          if |good| > n/2 and seen has no messages conflicting with m
14          then possibleRB ← possibleRB ∪ {m}
15          send (m, possibleRB ∩ C(m), THIRD) to sender(m)
16      upon receive(m, *, THIRD) from n − f processes for the first time:
17          R ← {r : received (m, *, THIRD) from r}
18          for each r ∈ R do poss[r] ← M s.t. received (m, M, THIRD) from r
19          if |r : m ∈ poss[r]| > n/2 then rbroadcast(m)
20          else abroadcast(m, ∪_{r∈R} poss[r])
21      upon rdeliver(m): if m not gdelivered then gdeliver(m)
22      upon adeliver(m, prec):
23          for each m' ∈ prec do
24              if m' not gdelivered then gdeliver(m')
25          if m not gdelivered then gdeliver(m)
```

Fig. 1. Non-trivial implementation of generic broadcast with an atomic broadcast oracle

upon the non-trivial implementation given in Section 4. In this implementation, $C(M) = M \cup \{m' : m' \text{ conflicts with some } m \in M\}$, and $C(m) = \{m\} \cup \{m' : m' \text{ conflicts with } m\}$.

Each process p keeps four variables: *seen*, *possibleRB*, *stable* and *adel*. *seen* is the set of gbroadcast messages that p has seen but has not yet adelivered or rdelivered. *possibleRB* is the set of gbroadcast messages that can be rbroadcast, but were not yet adelivered or rdelivered. *adel* is the set of messages that p has adelivered. *stable* is a set of pairs (m, B), where m is a message and B is a set of messages. Intuitively, $(m, B) \in$ *stable* means that p has adelivered or rdelivered m, and p must gdeliver all messages in B before it gdelivers m. We denote by π_1 the projection on the first component of a tuple or of a set of tuples. That is, $\pi_1((m, B))$ is m and $\pi_1(stable)$ is the set of m such that $(m, B) \in stable$, for some B.

To gbroadcast a message m, a process p sends (m, FIRST) to all processes. Upon receipt of such a message, a process q adds m to its *seen* set, if m is not in $\pi_1(stable)$. Then q sends to all processes a SECOND message containing m, together with *seen*, and those elements of *stable* whose first component either conflicts with some message in $seen \cup \{m\}$ or belongs to $seen \cup \{m\}$. When a process q receives $(m, s, st, \text{SECOND})$, it adds to *seen* those elements in s that are not in $\pi_1(stable)$, and it adds st to *stable*. When q collects SECOND messages from $n - f$ processes, it checks if *seen* contains m and no messages conflicting with m,

and if so, q adds m to *possibleRB*. Then, q sends to p — the gbroadcaster of m — a THIRD message containing m, *seen*, *possibleRB* and those elements in *stable* whose first component either conflicts with some message in $seen \cup \{m\}$ or belongs to $seen \cup \{m\}$. When p receives a THIRD message for m from $n - f$ processes, it checks if a majority of them have m in their third components and if m is not in $\pi_1(stable)$. If so, p rbroadcasts m, together with those messages in $\pi_1(stable)$ that conflict with m. Else, p abroadcasts m, together with (1) the so-called *flush* set, which contains those messages that are in the *seen* sets of a majority of processes, (2) the so-called *prec* set, which contains those messages that are in the *possibleRB* set of some process and that either conflict with a message in $flush \cup \{m\}$ or belong to $flush \cup \{m\}$, (3) those messages in $\pi_1(stable)$ that either conflict with a message in $flush \cup prec \cup \{m\}$ or belong to $flush \cup prec \cup \{m\}$. We assume that, before p abroadcasts $(m, flush, prec, \ldots)$, p chooses some arbitrary ordering for the messages in *flush* and *prec*, which will be known to any process that adelivers $(m, flush, prec, \ldots)$.

When a process q rdelivers $(m, before)$, it removes m from *possibleRB* and from *seen*, and adds $(m, before)$ to *stable*. Then, q looks for elements (m', B) in *stable* such that m' has not been gdelivered and all messages in B have been gdelivered. If q finds such an element, q gdelivers m'.

When q adelivers $(m, flush, prec, before)$, it removes $\{m\} \cup prec \cup flush$ from *possibleRB* and from *seen*, and adds *adel* to *before*. Then, q iterates over the ordered elements of *prec*. For each element m' of *prec*, q adds to *stable* the tuple (m', B), where B is the elements in *before* that conflict with m'. The intuition here is that $(m', B) \in stable$ means that p must gdeliver m' *after* p gdelivers all elements in B. Then, in a similar fashion, q iterates over the ordered elements of *flush*, to add each of them to *stable*. Next, q adds m to *stable* and adds $\{m\} \cup flush \cup prec$ to *adel*. Finally, q looks for elements (m', B) in *stable* such that m' has not been gdelivered and all messages in B have been gdelivered. If q finds such an element, q gdelivers m'.

Theorem 2. *Consider an asynchronous system with a majority of correct processes ($n > 2f$). The algorithm in Figure 2 is a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle.*

6 A sparing implementation of atomic broadcast

As we explained in the introduction, we would like to solve atomic broadcast with an algorithm that does not rely on an oracle whenever possible. Since no oracle is needed to order the delivery of causally related messages, we would like the atomic broadcast algorithm to stop using the oracle when messages are causally related.

More precisely, we say that *message m immediately causally precedes message m' and denote $m \rightarrow^1 m'$* if either (1) some process p abroadcasts m and then abroadcasts m' or (2) some process p adelivers m and then abroadcasts m'. Thus, $\rightarrow^1$ is a relation on the set of messages. Let $\rightarrow$ be the transitive closure

```
1   For every process p:
2       initialization:
3           seen ← ∅; possibleRB ← ∅; stable ← ∅; adel ← ∅
4       to gbroadcast(m): send (m, FIRST) to all processes
5       upon receive(m, FIRST) from q:
6           if m ∉ π1(stable) then seen ← seen ∪ {m}
7           send (m, seen, {X ∈ stable : π1(X) ∈ C(seen ∪ {m})}, SECOND) to all processes
8       upon receive(m, s, st, SECOND) from q:
9           seen ← seen ∪ (s \ π1(stable)); stable ← stable ∪ st
10          if received messages of the form (m, *, *, SECOND) from n − f processes for the first time then
11              if m ∉ π1(stable) then seen ← seen ∪ {m}
12              if seen ∩ C(m) = {m} then possibleRB ← possibleRB ∪ {m}
13              send (m, seen, possibleRB, {X ∈ stable : π1(X) ∈ C(seen ∪ {m})}, THIRD) to sender(m)
14      upon receive(m, *, *, *, THIRD) from n − f processes for the first time:
15          R ← {r : received (m, *, *, *, THIRD) from r}
16          for each r ∈ R do
17              s[r] ← M, where M is the set such that p received (m, M, *, *, THIRD) from r
18              poss[r] ← M, where M is the set such that p received (m, *, M, *, THIRD) from r
19              stable ← stable ∪ M, where M is the set such that p received (m, *, *, M, THIRD) from r
20          if |r : m ∈ poss[r]| > n/2 and m ∉ π1(stable) then rbroadcast(m, π1(stable) ∩ C(m))
21          else if m ∉ π1(stable) then
22              flush ← {m′ : m′ ≠ m ∧ |q : m′ ∈ s[q]| > n/2}
23              prec ← ∪r∈R poss[r] ∩ C(flush ∪ {m})
24              abroadcast(m, flush, prec, π1(stable) ∩ C(flush ∪ prec ∪ {m}))
                /* in the abroadcast message above, sets flush and prec are ordered, arbitrarily */
25      upon rdeliver(m, before):
26          possibleRB ← possibleRB \ {m}; seen ← seen \ {m}
27          stable ← stable ∪ {(m, before)}
28          while ∃(m′, B) ∈ stable s.t. m′ not gdelivered and all messages in B have been gdelivered
29          do gdeliver(m′)
30      upon adeliver(m, flush, prec, before):
31          possibleRB ← possibleRB \ ({m} ∪ prec ∪ flush); seen ← seen \ ({m} ∪ prec ∪ flush)
32          before ← before ∪ adel
33          for each m′ ∈ prec do stable ← stable ∪ {(m′, C(m′) ∩ before)}; before ← before ∪ {m′}
34          for each m′ ∈ flush do stable ← stable ∪ {(m′, C(m′) ∩ before)}; before ← before ∪ {m′}
            /* the for each loops above iterate in the order of the ordered sets prec and flush */
35          stable ← stable ∪ {(m, C(m) ∩ before)}; adel ← adel ∪ {m} ∪ flush ∪ prec
36          while ∃(m′, B) ∈ stable s.t. m′ not gdelivered and all messages in B have been gdelivered
37          do gdeliver(m′)
```

Fig. 2. Thrifty implementation of generic broadcast with an atomic broadcast oracle

of $\rightarrow$[1]. We say that m *is causally related to* m' if either $m \rightarrow m'$ or $m' \rightarrow m$. If m and m' are not causally related, we say that m *and* m' *are concurrent*. These definitions are based on [11].

Let $\mathcal{A}^X$ be an implementation of atomic broadcast that uses an oracle X.

Definition 3. *We say that* $\mathcal{A}^X$ *is* sparing w.r.t. oracle X *if it guarantees the following property: if there is a time after which messages abroadcast are pairwise causally related, then eventually* X *is no longer used. More precisely:* $\forall r \in Runs(\mathcal{A}^X), [\exists t, \forall m, m' \in gbcast\_msgs(r, [t, \infty)), m \rightarrow m' \vee m' \rightarrow m] \Rightarrow \exists t', X$ *is not used in* r *after time* t'.

In this section, we show how to transform *any* implementation of atomic broadcast that uses some oracle X, into an implementation that is sparing w.r.t to X. As a first step, we show how to transform any implementation of generic broadcast that is thrifty w.r.t. oracle X, into an implementation of atomic broadcast that is sparing w.r.t. X. This is achieved through the algorithm in Figure 3.

In this algorithm, *seq* denotes the number of messages that p has abroadcast so far, while $ndel[q]$ is the number of messages from q that p has adelivered so far. Intuitively, *ts* is a vector timestamp for messages such that if *ts* is the timestamp of m, then $ts[j]$ is the number of messages from process j that causally precede m. We can show that if m causally precedes m' and their timestamps are *ts* and *ts'*, respectively, then $ts \leq ts'$.

To abroadcast a message m, process p first obtains a new vector timestamp *ts* for m, by copying the vector *ndel* to *ts*, and then changing $ts[p]$ to a new sequence number. Then p gbroadcasts m with its timestamp *ts*. Upon gdeliver of (m, ts), a process q copies *ts* to *prec*, and changes $prec[sender(m)]$ to $ts[sender(m)] - 1$. Intuitively, *prec* represents the number of messages from each process that q must adeliver before q can adeliver m. Then q appends $(m, prec)$ to L, and then searches for the first message $(m', prec')$ in L with $prec' \leq ndel$.[8] If it finds such a message, it adelivers m', increments $ndel[sender(m')]$ by one, and removes $(m', prec')$ from L.

Theorem 3. *Consider an asynchronous system with at least one correct process. If we plug-in an implementation of generic broadcast that is thrifty w.r.t. oracle* X *into the algorithm in Figure 3, then we obtain an implementation of atomic broadcast that is sparing w.r.t. oracle* X.

As we now explain, we can use this result to transform any implementation $\mathcal{A}^X$ of atomic broadcast that uses an oracle X, into an implementation $\mathcal{A}^X_{sparing}$ that is sparing w.r.t. X. To do so, we first replace the atomic broadcast oracle in Figure 2 with $\mathcal{A}^X$, and thus obtain an implementation $\mathcal{G}^X_{thrifty}$ of generic broadcast that is thrifty w.r.t. X. We then use the transformation in Figure 3 to transform $\mathcal{G}^X_{thrifty}$ to $\mathcal{A}^X_{sparing}$ — an implementation of atomic broadcast that is sparing w.r.t. X (by Theorem 3).

Theorem 4. *Given any implementation of atomic broadcast that uses some oracle* X*, we can transform it to one that is sparing w.r.t.* X.

[8] We say that a vector $v_1 \leq v_2$ if for every $q \in \Pi$, $v_1[q] \leq v_2[q]$.

```
1  For every process p:

2    initialization:
3      seq ← 0        /* # of messages abroadcast by p */
4      L ← ∅          /* ordered set with message to adeliver */
5      for each q ∈ Π do ndel[q] ← 0
                      /* ndel[q] = # of messages from q that p has adelivered */
6      define (m, ts) ~ (m', ts') iff ts ⋦ ts' and ts' ⋦ ts
                      /* conflict relation for generic broadcast */

7    to abroadcast(m):
8      seq ← seq + 1; ts ← ndel; ts[p] ← seq      /* get new timestamp */
9      gbroadcast(m, ts)      /* with ~ as the conflict relation */

10   upon gdeliver(m, ts):
11     prec ← ts; prec[sender(m)] ← ts[sender(m)] − 1
12     L ← L · (m, prec)      /* append (m, prec) to L */
13     while ∃(m', prec') ∈ L such that prec' ≤ ndel do
14       (m', prec') ← first element in L such that prec' ≤ ndel
15       adeliver(m')
16       ndel[sender(m')] ← ndel[sender(m')] + 1
17       L ← L \ (m', prec')
```

Fig. 3. Transforming thrifty generic broadcast into sparing atomic broadcast

7 Low-latency thrifty implementations of generic broadcast

It is easy to see that the generic broadcast implementations in Figures 1 and 2 guarantee that in "good" runs with no failures and no conflicting messages, every message is delivered within 4δ, where δ is the maximum network message delay.[9] It turns out that we can decrease this latency to 3δ with some simple modifications to the algorithms. Moreover, if we assume that $n > 3f$ (i.e., more than two-thirds of the processes are correct) then we can further reduce the latency to 2δ. With the thrifty implementation, this latency is eventually achieved even in runs with failures and conflicting messages, provided that there is a time after which the messages gbroadcast are not conflicting.

Reducing the message latency to 3δ. To achieve a latency of 3δ in good runs, we modify the implementation in Figure 1 as follows: (1) processes should send the THIRD message to all processes in line 15, (2) instead of rbroadcasting a message m in line 19, a process p sends a message telling all processes to "deliver m", and then p gdelivers m, and (3) upon the receipt of a "deliver m" message for the first time, a process relays this "deliver m" message to all processes and gdelivers m. With this modification, it is easy to see that in good runs, every gbroadcast message is gdelivered within 3δ.

Theorem 5. *With the modifications above, the algorithm in Figure 1 ensures that, in runs with no failures and no conflicting messages, every gbroadcast message is gdelivered within 3δ, where δ is the maximum network message delay.*

[9] This assumes a reasonable implementation of reliable broadcast, which is used as a subroutine in these implementations.

We can modify the thrifty implementation in Figure 2 in a similar manner: (1) processes send the THIRD message to all processes in line 13, (2) instead of rbroadcasting a message in line 20, a process sends a "deliver $(m, \pi_1(stable) \cap C(m))$" message to all processes, sets variable *before* to $\pi_1(stable) \cap C(m)$, and then executes the code in lines 26–29, and (3) upon the receipt of message "deliver $(m, before)$" for the first time, a process relays this "deliver" message to all processes, and executes the code in lines 26–29.

Theorem 6. *With the modifications above, the algorithm in Figure 2 ensures that if there is a time after which the messages gbroadcast are not conflicting, eventually every gbroadcast message is gdelivered within 3δ, where δ is the maximum network message delay.*

Reducing the message latency to 2δ when $n > 3f$. To achieve a latency of 2δ in good runs, we assume that $n > 3f$ (instead of $n > 2f$). With this assumption, Figure 4 gives a non-trivial implementation of generic broadcast. The implementation is a simplification of the one in Figure 1, and uses atomic broadcast as the oracle.

```
1   For every process p:
2       initialization:
3           seen ← ∅; good ← ∅
4       to gbroadcast(m): send (m, FIRST) to all processes
5       upon receive(m, FIRST) from q:
6           seen ← seen ∪ {m}
7           if seen ∩ C(m) = {m} then good ← good ∪ {m}
8           send (m, good ∩ C(m), SECOND) to all processes
9       upon receive(m, *, SECOND) from n − f processes for the first time:
10          R ← {r : received (m, *, SECOND) from r}
11          for each r ∈ R do g[r] ← M s.t. received (m, M, SECOND) from r
12          if |{r : g[r] = {m}}| > 2n/3 then send (m, DELIVER) to all processes; gdeliver(m)
13          else if p = sender(m) then
14              poss ← {m' ≠ m : |{r : m' ∈ g[r]}| > n/3}
15              abroadcast(m, poss)
16      upon receive(m, DELIVER) from some process:
17          if m not gdelivered then send (m, DELIVER) to all processes; gdeliver(m)
18      upon adeliver(m, prec):
19          for each m' ∈ prec do
20              if m' not gdelivered then gdeliver(m')
21          if m not gdelivered then gdeliver(m)
```

Fig. 4. Low-latency non-trivial implementation of generic broadcast

Theorem 7. *Consider an asynchronous system with $n > 3f$. The algorithm in Figure 4 is a non-trivial implementation of generic broadcast that uses atomic broadcast as an oracle. In runs with no failures and no conflicting messages, every gbroadcast message is gdelivered within 2δ, where δ is the maximum network message delay.*

Figure 5 gives a thrifty implementation of generic broadcast with a latency of 2δ in good runs. The implementation is a simplification of the one in Figure 2, and uses atomic broadcast as the oracle.

Note that, in line 18, process p sends a message to itself. We did this to avoid repetition of code; p should not really send a message to itself, but rather execute the code in lines 24–28.

```
1  For every process p:
2    initialization:
3      seen ← ∅; good ← ∅; stable ← ∅; adel ← ∅
4    to gbroadcast(m): send (m, FIRST) to all processes
5    upon receive(m, FIRST) from q:
6      if m ∉ π1(stable) then seen ← seen ∪ {m}
7      if m ∉ π1(stable) and seen ∩ C(m) = {m} then good ← good ∪ {m}
8      send (m, seen, good, {X ∈ stable : π1(X) ∈ C(seen ∪ {m})}, SECOND) to all processes
9    upon receive(m, ss, *, st, SECOND) from q:
10     seen ← seen ∪ (ss \ π1(stable)); stable ← stable ∪ st
11     if received messages of the form (m, *, *, *, SECOND) from n − f processes for the first time then
12       R ← {r : received (m, *, *, *, SECOND) from r}
13       for each r ∈ R do
14         s[r] ← M, where M is the set such that p received (m, M, *, *, SECOND) from r
15         g[r] ← M, where M is the set such that p received (m, *, M, *, SECOND) from r
16       if m ∉ π1(stable) then seen ← seen ∪ {m}
17       if |r : m ∈ g[r]| > 2n/3 and m ∉ π1(stable)
18       then send (m, π1(stable) ∩ C(m), DELIVER) to p
19       else if m ∉ π1(stable) and p = sender(m) then
20         flush ← {m' : m' ≠ m ∧ |q : m' ∈ s[q]| > 2n/3}
21         prec ← {m' : m' ≠ m ∧ |q : m' ∈ g[q]| > n/3}
22         abroadcast(m, flush, prec, π1(stable) ∩ C(flush ∪ prec ∪ {m}))
           /* in the abroadcast message above, sets flush and prec are ordered, arbitrarily */
23   upon receive(m, before, DELIVER) from some process for the first time:
24     send (m, before, DELIVER) to all processes
25     good ← good \ {m}; seen ← seen \ {m}
26     stable ← stable ∪ {(m, before)}
27     while ∃(m', B) ∈ stable s.t. m' not gdelivered and all messages in B have been gdelivered
28     do gdeliver(m')
29   upon adeliver(m, flush, prec, before):
30     good ← good \ ({m} ∪ prec ∪ flush); seen ← seen \ ({m} ∪ prec ∪ flush)
31     before ← before ∪ adel
32     for each m' ∈ prec do stable ← stable ∪ {(m', C(m') ∩ before)}; before ← before ∪ {m'}
33     for each m' ∈ flush do stable ← stable ∪ {(m', C(m') ∩ before)}; before ← before ∪ {m'}
       /* the for each loops above iterate in the order of the ordered sets prec and flush */
34     stable ← stable ∪ {(m, C(m) ∩ before)}; adel ← adel ∪ {m} ∪ flush ∪ prec
35     while ∃(m', B) ∈ stable s.t. m' not gdelivered and all messages in B have been gdelivered
36     do gdeliver(m')
```

Fig. 5. Low-latency thrifty implementation of generic broadcast with an atomic broadcast oracle

Theorem 8. *Consider an asynchronous system with $n > 3f$. The algorithm in Figure 5 is a thrifty implementation of generic broadcast that uses atomic broadcast as an oracle. If there is a time after which the messages gbroadcast are not conflicting, eventually every gbroadcast message is gdelivered within 2δ, where δ is the maximum network message delay.*

References

1. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. Technical Report (to appear), DIX, École Polytechnique, Palaiseau, France, 2000.
2. G. Alvarez, F. Cristian, and S. Mishra. On-demand fault-tolerant atomic broadcast protocol. In *Proceedings of the Fifth IFIP International Conference on Dependable Computing for Critical Applications*, Sept. 1995.
3. Y. Amir, P. Moser, L.E. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, Nov. 95.
4. K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, Feb. 1987.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *J. ACM*, 43(2):225–267, Mar. 1996.
6. W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proceedings of the First International Conference on Dependable Systems and Networks (also FTCS-30)*, June 2000.
7. F. Cristian, H. Aghili, H. R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985.
8. D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environment. In *IEEE Proceedings of the 23th International Symp on Fault-tolerant computing (FTCS-23)*, pages 544–553, June 1993.
9. A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast (extended abstract). In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 297–309, Aug. 1990.
10. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, May 1994.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
12. F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, Sept. 1999.
13. F. Pedone and A. Schiper. Generic broadcast. Technical Report SSC/1999/012, École Polytechnique Fédérale de Lausanne, Switzerland, Apr. 1999.
14. R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of Middleware'98*, Sept. 1998.

Locating Information with Uncertainty in Fully Interconnected Networks

Lefteris M. Kirousis[1⋆] Evangelos Kranakis[2⋆⋆] Danny Krizanc[3⋆⋆⋆] and Yannis C. Stamatiou[4†]

[1] University of Patras, Department of Computer Engineering and Informatics, Rio 26500, Patras, Greece. e-mail: kirousis@ceid.upatras.gr

[2] Carleton University School of Computer Science, Ottawa, ON, K1S 5B6, Canada. email: kranakis@scs.carleton.ca

[3] Wesleyan University, Department of Mathematics, Middletown, CT 06459. e-mail: dkrizanc@wesleyan.edu

[4] University of Patras, Department of Computer Engineering and Informatics, Rio 26500, Patras, Greece and Computer Technology Institute, Kolokotroni 3, GR-262 21 Patras, Greece. e-mail: stamatiu@ceid.upatras.gr

Abstract. We consider the problem of searching for a piece of information in a fully interconnected computer network (or *clique*) by exploiting advice about its location from the network nodes. Each node contains a database that "knows" what kind of documents or information are stored in other nodes (e.g. a node could be a Web server that answers queries about documents stored on the Web). The databases in each node, when queried, provide a pointer that leads to the node that contains the information. However, this information is up-to-date (or correct) with some bounded probability. While, in principle, one may always locate the information by simply visiting the network nodes in some prescribed ordering, this requires a time complexity in the order of the number of nodes of the network. In this paper, we provide algorithms for locating an information node in the complete communication network, that take advantage of *advice* given from network nodes. The nodes may either give correct advice, by pointing directly to the information node, or give wrong advice by pointing elsewhere. We show that, on the average, if the probability p that a node provides correct advice is asymptotically larger than $1/n$, where n is the number of the computer nodes, then the average time complexity for locating the information node is, asymptotically, $1/p$ or $2/p$ depending on the available memory. The probability p may, in general,

⋆ Partially supported by the Computer Technology Institute.

⋆⋆ Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant and by MITACS project CANCCOM (Complex Adaptive Networks for Computing and Communication).

⋆⋆⋆ Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant and by MITACS project CANCCOM (Complex Adaptive Networks for Computing and Communication).

† Supported by the Greek Ministry of National Economy through a NATO scholarship (contract number 106384/ΔΟΟ 1222/2-7-98) and by the MITACS project CANCCOM. Also partially supported by the Computer Technology Institute.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 283–296, 2000.

be a function of the number of network nodes n. On the lower bounds side, we prove that no fixed memory deterministic algorithm can locate the information node in finite expected number of steps. We also prove a lower bound of $\frac{1}{p}$ for the expected number of steps of any algorithm that locates the information node in the complete network.

Key Words and Phrases: Search, Information Retrieval, Complete Network, Uncertainty, Random Walks.

1 Introduction

Suppose that we have a network of computers interconnected in some particular topology (e.g. ring, mesh, clique etc.) and one of the computers possesses a piece of information. The objective is to design a *software agent* that is able to travel along the communication links of the network and locate the information as fast as possible. An additional element to this picture, that differentiates the search from the usual random search, is that some of the computers, when queried, will respond with the name of the computer that holds the information. However, some other faulty computers, when queried, will give consistently wrong advice as to which computer has the information. (We do not consider intermittent faults that may appear and then disappear in an unpredictable fashion.) In this paper we address the problem of locating a piece of information in the *complete network* using, possibly incorrect, advice from the nodes, where each computer may communicate directly with any other computer.

This variant of *searching with uncertainty*, was introduced in [12], where the network topologies were the *ring* and the *torus*. Models with faulty information in the nodes have been considered before for the problem of routing (see [1, 3, 6, 8, 14]). However, in this problem it is assumed that the identity of the node that contains the information is known, and what is required is to reach this node following the best possible route. Search problems in graphs, where the identity of the node that contains the information sought is not known, have been considered in the past. We have the *deterministic search games*, where a fugitive that possesses some properties (it is agile or inert) hides in the nodes or edges of a graph and the aim is to locate it using as few searchers as possible (for definitions and relevant theory see [5, 11, 16]). Also, the problem of exploring an *unknown* graph has been considered (see [2, 13, 17, 18], for example). Closer to the spirit of our work are the search problems that are defined and studied in [4] where the authors consider problems of locating points in the plane using incomplete knowledge about their position. On another front, a number of algorithms have appeared that search a graph relying on some sort of *random walk* along its edges (for a comprehensive and very readable survey, see [15]). We have a very interesting class of games called *stochastic games* in which opponents' strategies incorporate some sort of probability transition matrix (see [21]). Also, in [22] one may find a detailed treatment of *Geometric Games* where the aim is to locate elements of a hidden set.

In this paper, we show how to locate a piece of information on the complete network of n nodes using some *advice* they give when queried. The advice is correct, i.e. it directs to the information node, with some bounded probability $p(n)$. The class of algorithms we consider can execute one of the following operations: (i) query a node of the complete network about the location of the information node (ii) follow the advice given by a queried node and (iii) select the next node to visit using some probability distribution function on the network nodes (that may be, for example, a function of the number of previously seen nodes or the number of steps up to now).

The algorithms we consider may be either *memoryless*, have *limited memory* or have *unlimited memory*. An algorithm is memoryless, if it may not store any information before it moves from a node to the next one. It is of limited memory, if it can only store a *fixed* amount of information (i.e. independent from the size of the complete network). Finally, an algorithm is of unlimited memory, if it may store any amount of information, usually the identities of the nodes encountered in the past. In addition, an algorithm may use randomization or be deterministic.

Table 1 summarizes our results for various types of algorithms. We observe

	memoryless	limited (fixed) memory	unlimited memory
randomized	$n-1$	$\frac{2}{p}$	$\frac{1}{p}$
deterministic	∞	∞	$\frac{1}{p}$

Table 1. Expected number of steps for various classes of algorithms

that no deterministic algorithm with no memory at all or with fixed amount of memory can locate the information node in a finite expected number of steps. The reason for this is that it may fall into cycles in the space of its states, for sufficiently large cliques. However, there exists a deterministic algorithm with unlimited memory ($O(n \log n)$ bits always suffice), that achieves an expected number of steps asymptotically equal to $\frac{1}{p}$. Moreover, randomized algorithms can achieve expectation $\frac{2}{p}$ even with a limited amount of memory. We give an algorithm that simply remembers if in the previous step it followed the advice of the queried node or not, thus requiring only one bit of memory. Moreover, when unlimited memory is available, the expectation falls to $\frac{1}{p}$.

In what follows, we first describe the network model we adopted and how the faulty and non-faulty nodes are determined. We then prove a lower bound of $\frac{1}{p}$ (we assume that p is not 1, since this trivial case is easily handled) for the expected number of steps required by any algorithm that locates the information node in the complete network. We also give an argument for the impossibility of achieving finite expectation in the number of steps with memoryless deterministic algorithms. We then describe some algorithms for searching for the information node and analyze their complexity.

2 The computer network model

In order to present and analyse the various algorithmic approaches to the problem of searching for a piece of information in the complete network, we will define a model of the computer network that captures the fact that some computers give a valid advice (i.e. they point to the node containing the information) while some others give faulty advice by pointing elsewhere.

The computer network is modelled as a clique on n nodes. There are direct links between any pair of nodes. Each node contains the following information:

- *Node identity number.* This is a number within the range $1 \ldots n$ that uniquely identifies each node.
- *Advice.* This is also a number within the range $1 \ldots n$ and it is interpreted as the advice of the node as to which node of the network contains the information.

In order to model the fact that the advice of a node may be incorrect with some probability as well as the fact that the information may reside in any node in a random fashion, we introduce randomness to the model in the following way:

- A node is randomly and uniformly selected from among the n nodes that contains the information and its identity, say s, becomes known to the other nodes. Node s sets its *advice* to be equal to itself, i.e. to contain s. In this way, any algorithm may distinguish a node that contains the information from one that does not.
- Each node, except s, flips a coin that shows heads with probability p (a varying parameter of the model) and if head shows-up, then the node sets its *advice* equal to s. These nodes give correct advice about the location of the information. If tails appears, then the node randomly and uniformly selects a number from within the set $\{1 \ldots n\} - \{s, \mathrm{id}\}$ where *id* is the node's identity number. These nodes are the *faulty* nodes.

After this procedure is carried out, *no changes* are permitted to any piece of information that was determined by the procedure. That is, the sets of faulty and non-faulty nodes remain the same and the wrong advice given by the faulty nodes never changes.

3 Some lower bounds for searching in the clique

Let $K_n = (V_n, E_n)$ be the clique graph on n vertices. In this section, we will prove that if $p = \omega(1/n)$, then no algorithm that searches for a piece of information in K_n (under the model we have defined) can do, asymptotically, better than executing $1/p$ steps on the average. To this end, we will use the fact that given a random set of node $S \subset V_n$ of cardinality k an *adaptive* random walk that starts from a node in $V_n - S$ will hit some node in S in expected time that converges

to $\Theta(\frac{n}{k})$, as n tends to infinity. An adaptive random walk is simply a random walk in which after each step, the currently visited node, along with its edges, is deleted from the clique.

Our aim is to show that the problem of locating the information node is at least as difficult as the problem of locating one of k non faulty nodes using a random walk that starts from a randomly chosen initial node. The following proposition holds:

Proposition 1. *For a randomly selected set S of non-faulty nodes, locating the information node $t \in S$ is at least as hard, on the average, as hitting the random set S.*

It is easy to see why this proposition is valid by observing that *any* algorithm that locates the information node in some number of steps may also be used to hit the random set to which the information node belongs in the same number of steps.

Now given a node v in V_n, we denote by $r_v(u)$ the probability of the adaptive random walk choosing vertex $u \neq v$ when it leaves from node v. In the uniform adaptive random walk on K_n, $r_v(u) = \frac{1}{n-1}$ for every v, u. The following can be proved:

Lemma 2. *The uniform adaptive random walk is an optimal algorithm for hitting a randomly selected set of k clique nodes.*

PROOF Suppose that an algorithm uses a non-uniform distribution function $r_v(u)$ (that may also be a function of the current step) and that it is about to select the next vertex to visit. Then the probability of failure to select at step i a node from the random set of size k is

$$\begin{aligned} P_v &= \sum_{u \text{ adjacent to } v} r_v(u) \left(1 - \frac{k}{n-i}\right) \\ &= \left(1 - \frac{k}{n-i}\right) \sum_{u \text{ adjacent to } v} r_v(u) = \left(1 - \frac{k}{n-i}\right), \end{aligned}$$

since nothing has been revealed about the initial random set of k nodes (it is still random after the deletion of i clique nodes).

Therefore, we see that the probability of failure from any node is independent of the probability distribution chosen for the node by the algorithm. Therefore, the uniform distribution is an optimal one.

Now no algorithm can have at the ith step a probability of failure less than the probability of failure of the adaptive random walk. This is because for such an algorithm the set of the k selected nodes is still random and the clique is essentially the clique K_{n-i}. Therefore, the probability of not hitting this set, using *any* distribution on the outgoing edges of the currently visited node, is $\frac{k}{n-i}$. ■

According to the lemma above, the number of steps taken by the adaptive random walk is a lower bound for the number of steps required by any algorithm to locate one of the randomly chosen nodes. The expected number of steps required by the uniform adaptive random walk is the following:

$$\sum_{i=1}^{n-k+1} i \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) \frac{k}{n-i+1}.$$

As n tends to infinity, the above expression is bounded from below, asymptotically, by the expectation of the geometric distribution with probability of success $\frac{k}{n}$. The following lemma will help us establish this fact:

Lemma 3. *The following inequality holds:*

$$\frac{k}{(n-k)^k} \int_k^n \frac{(n+1-z)(z-k)^k}{z} dz + \frac{n-k+1}{k+1^{n-k}} \leq \sum_{i=1}^{n-k+1} i \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) \frac{k}{n-i+1}.$$

Proof We write

$$\sum_{i=1}^{n-k+1} i \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) \frac{k}{n-i+1} =$$
$$\sum_{i=1}^{n-k} i \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) \frac{k}{n-i+1} + \frac{n-k+1}{(k+1)^{n-k}}.$$

Then for the product it holds that

$$\begin{aligned} \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) &= \exp\left(\sum_{l=1}^{i-1} \ln(1 - \frac{k}{n-l+1})\right) \\ &\geq \exp\left(\int_1^i \ln(1 - \frac{k}{n-l+1}\right) dl \\ &= \left(1 - \frac{k}{n}\right)^n \frac{1}{(n-k)^k} \frac{(n-i+1)^{n-i+1}}{(n-i+1-k)^{n-i+1-k}}. \end{aligned}$$

Therefore, using again approximations of sums with integrals, we obtain the following:

$$\sum_{i=1}^{n-k+1} i \prod_{l=1}^{i-1} \left(1 - \frac{k}{n-l+1}\right) \frac{k}{n-i+1}$$
$$\geq \left(1 - \frac{k}{n}\right)^n \frac{k}{(n-k)^k} \sum_{i=1}^{n-k} i \frac{(n-i+1)^{n-i}}{(n-i+1-k)^{n-i+1-k}} + \frac{n-k+1}{(k+1)^{n-k}}$$

$$= \left(1-\frac{k}{n}\right)^n \frac{k}{(n-k)^k} \sum_{z=k+1}^{n} \frac{n+1-z}{(z-k)^{z-k}} z^{z-1} + \frac{n-k+1}{(k+1)^{n-k}}$$

$$= \left(1-\frac{k}{n}\right)^n \frac{k}{(n-k)^k} \sum_{z=k+1}^{n} \frac{(n+1-z)(z-k)^k}{(1-\frac{k}{n})^n z} + \frac{n-k+1}{(k+1)^{n-k}}$$

$$\geq \frac{k}{(n-k)^k} \int_k^n \frac{(n+1-z)(z-k)^k}{z} dz + \frac{n-k+1}{(k+1)^{n-k}} \quad (1)$$

as claimed. ■

Setting the denominator of the integrand in (1) equal to n, we have the following:

Corollary 4. *The following inequality holds:*

$$\sum_{i=1}^{n-k+1} i \prod_{l=1}^{i-1} \left(1-\frac{k}{n-l+1}\right) \frac{k}{n-i+1} \geq \frac{k}{k+1}\frac{n+2}{k+2}\left(1-\frac{k}{n}\right) + \frac{n-k+1}{(k+1)^{n-k}}.$$

From the corollary, we see that the uniform adaptive random walk cannot do better, asymptotically, than achieving an expected number of steps $\frac{n}{k}$, if $k = o(n)$. If $\frac{k}{n} = c < 1$, then the lower bound becomes, asymptotically, $\frac{1}{c} - 1$ but it is of no better "order" than $\frac{1}{c}$.

In order to use the corollary in the context of searching with uncertainty in the complete network, we observe that if the probability of a node being non-faulty is p, then, on the average, there are pn non-faulty nodes in the network. The remaining nodes are faulty and they give incorrect advice. Therefore, any algorithm that searches for the information node can, at the very best, locate one of these randomly chosen pn nodes and follow its advice. This can be done in expected number of steps $\frac{n}{k} = \frac{n}{pn} = \frac{1}{p}$.

We will now give an argument for showing that no bounded memory deterministic algorithm can achieve a finite expected number of steps. In other words, randomization is *necessary* if memory is bounded. Let A be a deterministic algorithm with a fixed amount of memory, that locates the information node in cliques of any size. A deterministic algorithm that performs one of the operations we described, can be thought of as a function $f : (S, V) \mapsto (S, V \cup \{0\})$, where S is the finite set of different states A may assume, $V = \{v_1, \ldots, v_n\}$ is the set of nodes and the number 0 means that the algorithm follows the advice of the currently visited clique node. By *states of the algorithm*, we mean the different possible contents of its memory combined with its internal states. The set of its internal states is necessarily finite.

It is easy to see, that there can be no deterministic algorithm that locates the information node in cliques of any size, that does not follow the advice of at least one of the nodes it encounters. For let K_n be one of the cliques that the algorithm handles correctly. Then the set of pairs (s, v) with s a state of

the algorithm and $v \in V_n$ a vertex of the clique with n nodes is finite since the algorithm has finite memory. If n' is an integer larger than $|S \times V_n|$, then consider the operation of the algorithm on $K_{n'}$ (in what follows, we consider K_n as a subgraph of $K_{n'}$). Since the information node is chosen randomly, there is a nonzero probability that it will be one of the nodes in $V_{n'} - V_n$, say in v'. Since the algorithm is deterministic, there is no pair (s, v), with $v \in K_n$, that is mapped onto (s', v') for some state s'. Then the algorithm fails to locate the information node, which contradicts our assumption that it works correctly for cliques of any size. Therefore, there must be some state/node pair that is mapped onto $(s', 0)$ by the algorithm, for some state s' and, moreover, the algorithm must encounter at least one of these pairs, say (s, v), at some point of its operation on K_n (otherwise it would, again, not operate properly with $K_{n'}$). Now there is a nonzero probability that the following events occur simultaneously:

- The node v is faulty.
- The (wrong) advice it gives sends the algorithm to a previously encountered node (but, possibly, the algorithm is now in another state).

The algorithm must again follow the advice of some node u in K_n at some point for otherwise there exists again a sufficiently large clique that cannot be handled correctly. Again, the two events above occur simultaneously for u with nonzero probability. Since the advice of the faulty nodes never changes, the event that the algorithm will be constantly directed to previously seen state/node pairs (for K_n) is fixed and nonzero. Since the state/node pairs are finite, the algorithm will eventually reach a previously encountered pair (s, v), with some fixed nonzero probability. If this occurs, then the algorithm will repeat infinitely the same steps without ever locating the information node in K_n. Therefore, the expected number of steps required by the algorithm cannot be finite.

The essence of the above argument is that a finite memory deterministic algorithm should always follow the advice of the nodes it encounters. However, in doing this, it will eventually enter a cycle in its state space with fixed and nonzero probability. It can also be proved that if we alter our model so that when a faulty node is queried again it determines its wrong advice at random and does not necessarily give the same wrong advice, then the "always follow the advice" policy results in an algorithm with expected number of steps equal to $\frac{1}{p}$.

4 Fixed memory and randomization: one bit of memory helps

In this section, we will show that just one bit of memory suffices in order to reduce the expected number of steps to locate the information node from $n-1$ (random walk) to $\frac{2}{p}$, which is $o(n)$. In contrast, when the search algorithm is not allowed to have any memory, then an optimal way to search for the information node is to perform a random walk on the clique nodes. Such a random walk will hit the information node in expected number of steps $n-1$ (see [15]).

The algorithm given below, simply alternates between following the advice of the currently visited node and selecting a random node as the next node to visit. It only needs to remember if at the last step it followed the advice or not. Although one bit that it is complemented at each step suffices, the algorithm is stated for convenience as if it knew the number of steps it has taken, checking at each step if this number is odd or even.

Algorithm: Fixed Memory Search
Input : A clique (V, E) with a node designated as the information holder
Aim: Find the node containing the information

```
1.  begin
2.     current = RANDOM(V)
3.     l ← 1
4.     while current(information) ≠ true
5.        l ← l + 1
6.        if l mod 2 = 0
7.           current = current(advice)
8.        else
9.           current ← RANDOM(V − current)
10.    end while
11. end
```

Let us estimate the average number of steps that are required by the algorithm in order to locate the piece of information.

At step l, failure can occur in one of the following ways:

- If $l = 1$, the algorithm fails if it randomly selects one node other than the node containing the information. The probability for this happening is $q_1 = (1 - \frac{1}{n})$.
- If l is even, the algorithm fails if the currently visited node is faulty. The probability of this event is $q_l = q = 1 - p$.
- If l is odd larger than 1, the algorithm fails if it randomly selects a node other than the currently visited one, that does not contain the information. The probability of this event is $q_l = (1 - \frac{1}{n-1})$.

Then the expected number of steps (expectation of the random variable l) is the following (where $q_l = 1 - p_l$):

$$\begin{aligned} E[\#\text{ of steps}] &= \sum_{l=1}^{\infty} l \cdot \Pr[\text{first success in } l\text{-th step}] \\ &= \sum_{l=1}^{\infty} l \cdot q_1 \cdots q_{l-1} p_l \\ &= \left(1 - \frac{1}{n}\right) \sum_{l \geq 2,\ \text{even}} l q^{\frac{l}{2}-1} \left(1 - \frac{1}{n-1}\right)^{\frac{l}{2}-1} (1-q) \end{aligned}$$

$$+\left(1-\frac{1}{n}\right)\sum_{l\geq 3,\ \text{odd}} lq^{\frac{l+1}{2}-1}(1-\frac{1}{n-1})^{\frac{l-1}{2}-1}\frac{1}{n-1}+\frac{1}{n}$$

$$=\left(1-\frac{1}{n}\right)\sum_{l\geq 0}2(l+1)q^l(1-\frac{1}{n-1})^l(1-q)$$

$$+\left(1-\frac{1}{n}\right)\sum_{l\geq 0}(2(l+1)+1)q^{l+1}\left(1-\frac{1}{n-1}\right)^l\frac{1}{n-1}+\frac{1}{n}.$$

After some tedious algebraic manipulations, it can be shown that

$$E[\#\text{ of steps}]=1+\frac{(n-1)^2(1+q)}{n(n-qn+2q-1)}=1+\left(1-\frac{1}{n}\right)^2\frac{2-p}{p+\frac{1-2p}{n}}$$

$$\sim\frac{2+\frac{1-2p}{n}}{p+\frac{1-2p}{n}}.$$

If $pn\to\infty$, i.e. $p=\omega(\frac{1}{n})$, then the expression above converges to $\frac{2}{p}$. However, if $pn\to 0$, i.e. $p=o(\frac{1}{n})$, the exppression for the expectation tends to $2n$. If $p=\Theta(\frac{1}{n})$ so that $pn\to c$, then the expectation is asymptotically equal to $\frac{2n}{c}$ which again tends to $\frac{2}{p}$.

5 Unlimited memory: randomized and deterministic algorithms

We will now give an unlimited memory randomized algorithm for locating the information node in a clique. More specifically, the algorithm can store the previously visited nodes using $O(n\log n)$ memory bits and use this knowledge in order to decide its next move. In this way, the algorithm avoids visiting again previously visited nodes. Such an algorithm always terminates within $n-1$ steps in the worst case.

Algorithm: Unlimited Memory Rendomized Search
Input : A clique (V,E) with a node designated as the information holder
Aim: Find the node containing the information

```
1.  begin
2.    l ← 1
3.    current = RANDOM(V)
4.    M ← {current} // M holds the up to now visited nodes
5.    while current(information) ≠ true
6.      read(current(advice))
7.      if current ∉ M
8.        M ← M ∪ {current}
9.        current = advice
10.     end if
```

```
11.        else
12.          current ← RANDOM(V − M)
13.        l ← l + 1
14.    end while
15. end
```

We can now prove the following:

Theorem 5. *If $p = \omega(\frac{1}{n})$, then the expected number of steps required by the algorithm in order to locate the information is, asymptotically, equal to $1/p$.*

PROOF The algorithm fails at step $l \geq 1$, due to one of the following events:

1. Event A_1: the answer v to the query is one of the previously visited values.
2. Event A_2: the answer v is a node that lies outside the previously visited set but the piece of information is not there.

Let E_l be the event that at steps $1 \ldots l$ the algorithm failed to find the information. Then the probability of failure at the lth step, given that the algorithm has failed at steps $1, \ldots, l-1$, is equal to $q_1 q_2 \cdots q_l$, where

$$\begin{aligned} q_l &= \Pr[\text{failure in } l\text{-th step} | E_{l-1}] = \Pr[A_1 | E_{l-1}] + \Pr[A_2 | E_{l-1}] \\ &= q \frac{l}{n-2} \cdot \frac{n-l-2}{n-l-1} + q \frac{n-l-2}{n-2} \\ &= q \frac{n-l-2}{n-2} \left(1 + \frac{l}{n-l-1} \right) = \frac{n-l-2}{n-l-1} \cdot \frac{n-1}{n-2}. \end{aligned}$$

It follows that the expected number of steps of the above algorithm is given by the formula

$$\begin{aligned} E[\#\text{ of steps}] &= \sum_{l=1}^{n-2} l \cdot \Pr[\text{first success in } l\text{-th step}] = \sum_{l=1}^{n-2} l \cdot q_0 q_1 \cdots q_{l-1} p_l \\ &= \sum_{l=1}^{n-2} l \cdot q_0 q_1 \cdots q_{l-1} - \sum_{l=1}^{n-2} l \cdot q_0 q_1 \cdots q_{l-1} p_l \\ &= \sum_{l=1}^{n-2} q_0 q_1 \cdots q_{l-1} - (n-2) q_0 q_1 \cdots q_{n-2} \\ &= \sum_{l=1}^{n-2} q_0 q_1 \cdots q_{l-1} = \sum_{l=1}^{n-2} q^l \left(\frac{n-1}{n-2} \right)^l \frac{n-l-1}{n-1} \\ &= \frac{1}{n-1} \sum_{l=1}^{n-2} q^l \left(\frac{n-1}{n-2} \right)^l (n-l-1) \\ &= \frac{n}{n-1} \sum_{l=1}^{n-2} \left(q \cdot \frac{n-1}{n-2} \right)^l - \frac{1}{n-1} \sum_{l=1}^{n-2} (l+1) \left(q \cdot \frac{n-1}{n-2} \right)^l \\ &= \frac{n}{n-1} \frac{1-a^{n-1}}{1-a} - \frac{1}{n-1} \frac{-a^{n+1} + n a^n - n a^{n-1} + a}{(1-a)^2}, \end{aligned}$$

where $a = q\frac{n-1}{n-2}$. To obtain the last formula we used the well-known expansion of the geometric progression. It follows that asymptotically in n,

$$\frac{1-a^{n-1}}{1-a} \sim \frac{1-q^{n-1}e}{p}$$
$$\frac{-a^{n+1}+na^n-na^{n-1}+a}{(1-a)^2} \sim \frac{1}{n-1}\left(\frac{-q^{n+1}e+nq^ne-nq^{n-1}e+q}{p^2}\right).$$

Substituting this in the previous formula we obtain that

$$E[\#\text{ of steps}] = \frac{1}{p} - \frac{q^{n-1}e}{p} - \frac{1}{n-1}\left(\frac{-q^{n+1}e+nq^ne-nq^{n-1}e+q}{p^2}\right).$$

As $\frac{1}{p} = o(n)$ (from the hypothesis that $p = \omega(1/n)$), we obtain that, asymptotically, $E[\#\text{ of steps}] = 1/p$, which completes the proof of the theorem. ■

As we have proved, no fixed memory deterministic algorithm has a finite expectation in the number of steps to locate the information node. However, the unlimited memory randomized algorithm we described in this section can be converted easily into an unlimited memory deterministic one by only changing line 12 to be as follows:

$$\text{current} = \text{the lexicographically smallest node in } V - M.$$

This change does not affect the analysis of the randomized algorithm because the probability that the chosen node is faulty or not, or that it is the information node, is the same as if the next node had been chosen at random.

6 Conclusions

In this paper we have considered the problem of searching for an information node in the complete network using information about its location obtained from the nodes of the network, where the information is correct with some bounded probability p. This problem was introduced in [12] where the ring and toroidal interconnection networks were considered.

For the complete network, we proved that there is no algorithm that can locate the information node in expected number of steps less than $\frac{1}{p}$. We also proved that there is no fixed memory deterministic algorithm that achieves a finite expectation in the number of steps. We also gave various search algorithms and analyzed their expected number of steps. It is interesting to consider the same problem for general graphs. One complication that immediately arises is that a node giving correct advice does not point to the information node directly, since it may not be adjacent to it, but it points to a node that lies on a shortest path to that node (see [12]). It also appears that deriving useful (i.e. as functions of the number of nodes) lower bounds for the expected number of steps to locate the information node must be difficult. Our proof of the $\frac{1}{p}$ lower bound was based on the analysis of the expected number of steps required by a random

walk on the complete network before *hitting* a randomly chosen set of nodes that give *correct* advice. We believe that doing the same for other classes of graphs is an interesting line of research. However, tackling more general classes of graphs using the random walks technique, seems to be difficult, in view of the discussion in [15] about the analysis of the expected hitting times in random walks on various classes of graphs. In [15] it is stated that hitting (or access) times can remain bounded, i.e. independent of the number of nodes of the graph, even for regular graphs. In the same paper, it is also stated that the only graphs for which a nonconstant lower bound on the expected hitting time can be proved are the graphs with *transitive* automorphism group. Therefore, deriving lower bounds for the problem of searching with uncertainty in more general classes of graphs, may require more complex lower bound techniques.

References

1. Y. Afek, E. Gafni, and M. Ricklin, Upper and lower bounds for routing schemes in dynamic networks, *in: Proc. 30th Symposium on Foundations of Computer Science*, (1989), 370–375.
2. S. Albers and M. Henzinger, Exploring unknown environments, *in Proc. 29th Symposium on Theory of Computing*, (1999), 416–425.
3. B. Awerbuch, B. Patt-Shamir, and G. Varghese, Self-stabilizing end-to-end communication, *Journal of High Speed Networks* **5** (1996), 365–381.
4. R.A. Baeza-Yates, J.C. Culberson, and G.J.E. Rawlins, Searching in the plane, *Information and Computation* **106(2)** (1993), 234–252.
5. D. Bienstock and P. Seymour, Monotonicity in graph searching, *Journal of Algorithms* **12** (1991), 239–245.
6. R. Cole, B. Maggs, and R. Sitaraman, Routing on butterfly networks with random faults, *in: Proc. 36th Symposium on Foundations of Computer Science*, (1995), 558–570.
7. X. Deng and C. Papadimitriou, Exploring an unknown graph, *in Proc: 31st Symposium on Foundations of Computer Science*, 1990, 356–361.
8. S. Dolev, E. Kranakis, D. Krizanc, and D. Peleg, Bubbles: Adaptive routing scheme for high-speed networks, *SIAM Journal on Computing*, to appear.
9. W. Evans and N. Pippenger, Lower bounds for noisy boolean decision trees, *in Proc. 28th Symposium on Theory of Computing*, (1996), 620–628.
10. U. Fiege, D. Peleg, P. Raghavan, and E. Upfal, Computing with uncertainty, *in Proc. 22nd Symposium on Theory of Computing*, (1990), 128–137.
11. L. Kirousis and C. Papadimitriou, Interval graphs and searching, *Discrete Mathematics* **55** (1985), 181–184.
12. E. Kranakis and D Krizanc, Searching with uncertainty, *in: Proc. 6th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, (1999), C. Gavoille, J.-C, Bermond, and A. Raspaud, eds., pp, 194-203, Carleton Scientific, 1999.
13. E. Kushilevitz and Y. Mansour, Computation in noisy radio networks, *in Proc. 9th Symposium on Discrete Algorithms*, 1998, 236–243.
14. T. Leighton and B. Maggs, Expanders might be practical, *in: 30th Proc. Symposium on Foundations of Computer Science*, (1989), 384–389.
15. L. Lovász, Random Walks on Graphs: A Survey, *Combinatorics* **2** (1993), 1–46.

16. N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriou, The complexity of searching a graph, *Journal of the ACM* **35** (1988), 18–44.
17. P. Panaite and A. Pelc, Exploring unknown undirected graphs, *in: Proc. 9th Symposium on Discrete Algorithms*, (1998), 316–322.
18. C.H. Papadimitriou and M. Yannakakis, Shortest paths without a map, *Theoretical Computer Science* **84(1)** (1991), 127–150.
19. N. Pippenger, On networks of noisy gates, *in: Proc. 26th Symposium on Foundations of Computer Science*, (1985), 30–36.
20. P. Raghavan, Robust algorithms for packet routing in a mesh, *in: Proc. 1st Symposium on Parallel Algorithms and Architectures*, (1989), 344–350.
21. T.E.S. Raghavan, T.S. Ferguson, T. Parthasapathy, and O.J. Vrieze eds., *Stochastic games and related topics.* Kluwer Academic Publishers, 1991.
22. W.H. Ruckle, *Geometric games and their applications.* Research Notes in Mathematics, Pitman Publishing Inc., 1983.

Optimistic Replication for Internet Data Services

Yasushi Saito and Henry M. Levy

Department of Computer Science and Engineering,
University of Washington, Seattle, WA 98195, U.S.A.
{yasushi,levy}@cs.washington.edu,
http://porcupine.cs.washington.edu

Abstract. We present a new replication algorithm that supports replication of a large number of objects on a diverse set of nodes. The algorithm allows replica sets to be changed dynamically on a per-object basis. It tolerates most types of failures, including multiple node failures, network partitions, and sudden node retirements. These advantages make the algorithm particularly attractive in large cluster-based data services that experience frequent failures and configuration changes. We prove the correctness of the algorithm and show that its performance is near-optimal.

1 Introduction

We present a new lightweight replication algorithm designed for PC-based Internet data services, such as WWW, FTP, and email. Our algorithm is unique in many aspects. First, our algorithm lets copies of an object, or *replicas*, be created or deleted dynamically and yet guarantees that the state of all the replicas eventually converges. Second, it is designed specifically to support many small replicated objects, which is typical in web-based environments; in particular, it has low space and computation overhead and handles object deletion efficiently. Third, the algorithm tolerates most failure types, including multiple node failures and sudden node retirements, and network partitions. Finally, it is non-blocking and decentralized, that is, it lets any node issue an update, and it makes no single node permanently responsible for maintaining replica consistency. We achieve this efficiency and versatility by being *optimistic*, that is, we guarantee that replicas become consistent only eventually. This optimism precludes the use of our scheme in applications that demand high data reliability, such as banking, but it poses little problem in typical Internet services, because these services have simple update semantics and weak consistency requirements.

1.1 Background

Our work derives from the Porcupine cluster-based mail server project [19]. Porcupine connects up to several hundred off-the-shelf computers to serve billions of mail messages per day. Porcupine's architecture is fully *dynamic*; any node can

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 297–314, 2000.

potentially manage any user's profile and store any user's email messages. For each incoming message, Porcupine chooses a set of nodes on which to replicate and store the message (called a *replica set*), based on node load and message affinity. A cluster membership service and a distributed naming service keep track of the locations of users' messages. The dynamic message placement yields many benefits: performance improvement via balanced load and flexible support for system configuration changes by message migration.

From the viewpoint of a replication service, Porcupine presents an environment very different from traditional database systems. Following are the key characteristics of Porcupine and a discussion of how they define the goals of our replication service:

Frequent failures: With hundreds of nodes in the cluster, a part of the system is always down. First, the algorithm must provide ***strong fault tolerance*** by maintaining replica consistency even when some of the nodes are down, sometimes permanently. Second, the algorithm must be ***non-blocking***, that is, it must allow reads and writes to any replica any time regardless of whether peer replicas are reachable.

Changing replica sets: Porcupine needs to change email message replica sets automatically to react to node additions and removals. We must support ***dynamic addition and removal of replicas*** while allowing contents updates to the object.

Small object size: The unit of replication in Porcupine is an email message whose average size is 5K bytes, as opposed to many gigabytes typical in database systems. We need to ***minimize the space overhead*** of per-object data structures used to maintain replica consistency.

Selective replication: Porcupine stores billions of email messages, each of which is in its own replica set. Our algorithm needs to be ***quiescent***, that is, it should incur no computational and space overhead when no update is in progress. Moreover, email messages are deleted frequently. Thus, our system should support ***quick object deletion*** without leaving any data structures behind.

Weak consistency requirements: Services such as email do not demand strict replica consistency because the possibility of inconsistent data is inherent in the environment; for example, unreliable network transport can cause delivery delay or duplicate messages. Thus, we only need to support ***eventual consistency*** of replica state.

These service requirements are not unique to email — in fact, they are shared by many other Internet applications, including Usenet [20], Internet-based BBS services (e.g., slashdot.org or delphi.com), naming services [15, 13], and wide-area mirroring of Web or FTP data [17]. All these services are potential applications of our replication algorithm.

1.2 Related Work

Data replication has been studied and deployed widely. However, previous work in this area has addressed the goals of our algorithm only in a piecemeal fashion and has not solved all of the problems that we face in our intended environment.

Traditional replication algorithms include primary-copy algorithms used widely in commercial database systems [3], quorum consensus algorithms [7, 9, 10], and atomic broadcast protocols [4, 1]. They try to achieve single-copy semantics, that is, they give users an illusion of having a single, highly available copy of an object. Although generic, these algorithms fail to address problems we face – frequent failures and frequent changes – because they sacrifice availability by prohibiting accesses to a replica when data is not provably up to date.

Mobile replicated database systems (e.g., Bayou [16], and Roam [18]) share some of our goals: elimination of a single point of failure, handling frequent failures, and dynamic replica addition and deletion. The main difference between their solutions and ours is that these systems focus on minimizing the communication overhead, whereas we focus on minimizing the space and the computation overhead. For example, the techniques used by these systems, including on-demand polling and a semantic log for describing updates, reduce the communication cost but increase both the computation and the storage overhead.

Our algorithm is most closely related to multi-master wide-area replicated services, including Active Directory [13], Xerox Clearinghouse [6], and Usenet [20]. These systems provide non-blocking accesses, support replication of many small objects, and propagate updates efficiently over unreliable links. However, they do not support replica set changes and provide only a weak fault tolerance; for example, one failed node can stall the update propagation of the entire system. Moreover, the existing systems do not support quick object deletion and require storing update records (often called "death certificates") for an indefinitely long period.

Several systems allow dynamically changing the placement of replicas using reference-monitoring mechanisms to balance the system load [17, 23]. They update replicas by gossiping changes along a spanning tree and are unable to achieve replica consistency even under a single node failure. Our work complements them by proposing a robust mechanism that can tolerate a wider variety of failures.

1.3 Overview of the Algorithm

Our algorithm is based on three principles: *state transfer*, update resolution using *Thomas write rule*, and update retirement using *synchronized clocks*.

In its basic form, our algorithm is similar to systems such as Active Directory [13] and Usenet [20]. Any replica (or any node for a newly created object) can issue an update any time. A coordinator, usually the issuer of the update, propagates the update by pushing the new object state to others in background. Conflicting updates are resolved by Thomas write rule [21], that is, by attaching timestamps to them and accepting only the newest update.

Our algorithm is unique in its uniform handling of replica set updates and contents updates — in fact, an update is actually a tuple consisting of the new object contents and the new replica set. For an update that changes the replica set, the coordinator pushes the update to the union of the old and the new replica sets (called the *targets* of the update). A node receiving the update either modifies, creates, or deletes a replica depending on whether or not it appears in the new replica set. Thomas write rule is again used to resolve conflicts among replica set changes; the older updates are canceled by forwarding the newest update to their targets and letting them be rolled back. Overriding the older updates requires the coordinator of the newer update to discover the older updates' targets. This *node discovery* problem is similar to the distributed resource discovery problem [8], and the solution is also similar: the nodes that receive the update send back to the coordinator the sets of nodes they know and let the coordinator expand its target node set transitively.

We apply at-most-once messaging technique using synchronized clocks [12] to retire updates. After the coordinator completes update propagation, it sends out retirement notices to the target nodes. Upon receiving a retirement notice, a node deletes the update from disk after waiting for a fixed period; the wait ensures that the node will never apply a stale update in the future.

This combination of state transfer, Thomas write rule, and quick update retirement allows our algorithm to solve our goals effectively as follows:

- Our algorithm's basic design directly achieves the three goals — non-blocking access, dynamic replica set changes, and eventual consistency.
- We achieve strong fault tolerance in two ways. First, our algorithm is fully decentralized — in particular, it lets any node take over the task of the coordinator any time. Second, its node discovery process eliminates a single point of failure quickly and lets the system tolerate a sudden node retirement without compromising replica consistency.
- Our algorithm's space overhead is quite small for two reasons. First, the state transfer architecture minimizes the size of the update record by omitting the new object contents – the object contents are usually read from the replica directly. Second, our algorithm quickly reclaims the space occupied by update records by retiring them as soon as they finish propagation.

1.4 Structure of the Paper

The rest of the paper is structured as follows. We describe our algorithm in detail in Section 2 and show two examples in Section 3 to elucidate its behavior, in particular, the resolution of concurrent updates. Section 4 proves the correctness of our algorithm. In Section 5, we discuss several extensions to the basic algorithm to address issues that arise in practice: e.g., optimizations to make the algorithm work efficiently and the handling of long-term failures. We briefly discuss the computational and the space overhead of the algorithm in Section 6 and conclude in Section 7.

2 The Replication Algorithm

Although our algorithm is designed to support many objects replicated on a diverse set of nodes, it is first presented in the context of a single object. We describe a straightforward extension of the basic algorithm to support multiple objects in Section 5.1.

2.1 System Model and Assumptions

We make the following assumptions about the environment. First, the nodes in the system can communicate through a fully connected network. Second, nodes and network links may crash, and messages may be reordered, delayed, or lost, but Byzantine failures will not occur. Finally, the nodes have loosely synchronized clocks; clock synchronization algorithms are well known and are deployed widely [14].

Our algorithm only propagates updates. Locating and reading replicas and choosing the replica placement are outside the scope of this algorithm. For locating replicas, any weakly consistent naming service can be used. For replication policy, an object could be assigned to a random set of nodes [19] or to a set determined by reference-monitoring mechanisms [17, 23].

2.2 Notational Conventions

All global variables are stored on stable storage and survive node crashes. Procedures marked **public** run as transactions that are non-preemptive and update global variables atomically. $\langle val_1, val_2 \rangle$ is a tuple of two values. Send(*node*, *proc*, *args*...) sends a message to *node* and requests calling *proc* with *args*.... Send does not wait for *proc* to finish; it merely queues the message. Texts after '|' are comments.

2.3 Data Structures

Fig. 1 shows the types used in the algorithm. Loosely synchronized clocks order updates [14]. Other types of clocks, e.g., logical clocks [11], may be used without affecting the correctness of the algorithm, but wall clocks best suit our purpose because they can order logically unrelated events (e.g., a user contacting two nodes in a cluster serially). The procedure Now() returns the current local clock value. We assume the clock resolution to be fine enough that successive calls to Now() always return different values; this assumption lets us use timestamps to identify updates.

```
type Timestamp = record
  time: Clock    | wall-clock time.
  nid: NodeID    | a tie-breaker.
end
type Update = record
  state: {ACTIVE, RETIRING,
          RETIRED, SUSPENDED}
  ts: Timestamp
  target, done, peer: NodeSet
end
```

Fig. 1. Data structures used by the algorithm.

An update to the object is represented by the **Update** record. Its *state* field indicates the update's status. A new update starts as being ACTIVE. An update is RETIRING on the coordinator while retirement notifications are sent, and it is RETIRED after the reception of a retirement notice. An update is SUSPENDED when it is found to conflict with a newer update, and it stays dormant until the newer update arrives and supersedes it. *Target*, *done*, and *peer* fields specify the set of nodes that should receive the update, have acknowledged the update, and should replicate the object, respectively. Thus, *done* and *peer* are always subsets of *target*. The update propagation finishes when *done* = *target*.

Five persistent variables are stored per replica on a node (Fig. 2). Two of them, *gData* and *gPeers*, are visible to the application and the rest are used internally by the replication algorithm. *GData* stores the actual contents of the object — we are not concerned about the object's internal structure in this paper. *GPeer* shows, to the best of the node's knowledge, the replica set of the object. *GU* remembers the newest update applied on the object. Notice the absence of the new object contents in *gU* — the contents are propagated to other nodes by reading from *gData* directly most of the time. The exception is when *gData* is deleted by an update, but the object contents still need to be propagated to other nodes (this happens, for example, when the object is moved from one node to another). *GSavedData* is used to save the new object contents in such cases, and it is otherwise null. That *gSavedData* is usually null contributes to reducing the space overhead of the algorithm, because all other data structures used by the algorithm are of small and fixed size. *GRetireTime* is used to delete a retired update and is discussed further in Section 2.7.

```
var
  gData: Contents←NULL
  gPeer: NodeSet←φ
  gU: Update←NULL
  gRetireTime: Clock
  gSavedData: Contents
```

Fig. 2. Per-node, per-object global variables used by the algorithm.

2.4 Application Programming Interface

One procedure, UpdateObject (Fig. 3), is called by the application. It takes two parameters, the new replica set (*peer*) and the new object contents (*data*). Passing an empty set to *peer* will delete the object entirely from the system. The caller of this procedure must ensure that the node stores a replica already, except when the object is being newly created. This restriction, also discussed in Section 4.2, is to prevent creating an orphan replica that is disconnected from others and is not found by the node discovery process. The implicit variable *me* shows the name of the node itself.

```
public proc UpdateObject(peer, data)
  u←Update(ts← Timestamp(Now(), me),
           state←ACTIVE, done←φ,
           peer←peer, target←peer)
  ApplyUpdate(u, data)
```

Fig. 3. UpdateObject is called by the application to create a new object, modify the object contents, add or remove the replica set, or delete the object.

2.5 Update Application

```
proc ApplyUpdate(u, data): bool
  | Expand knowledge. ⟨1⟩
  u.target ← u.target ∪ gPeer
  if gU ≠ NULL
      ∧ gU.state ⊈ {RETIRING,RETIRED}
    u.target ← u.target ∪ gU.target
    gU.target ← u.target
  | Reject if u is stale. ⟨2⟩
  if gU ≠ NULL ∧ gU.ts > u.ts
    return false
  | Log the update. ⟨3⟩
  gU ← u
  gSavedData ← NULL
  | Modify the replica.
  if me ∈ gU.peer
    ⟨gData, gPeer⟩ ← ⟨data, gU.peer⟩
  else
    ⟨gData, gPeer⟩ ← ⟨NULL, φ⟩
    if gU.peer ≠ φ
      | Save data; not needed when peer is
      | φ since everyone deletes the replica.
      gSavedData ← data
  gU.done ← gU.done ∪ {me}
  return true
```

Fig. 4. Local update application. This procedure logs and applies the update to the local replica. It returns whether the update was successfully applied or not.

The procedure ApplyUpdate (Fig. 4), called both from the local application and from remote nodes, logs and applies the update to the replica and prepares for update propagation. It first merges the target sets of both u and the current update, gU ⟨1⟩[†]. This must be done even when u is to be discarded ⟨2⟩ so that the participants of both updates can eventually receive the newer update.

2.6 Update Propagation

An update is pushed to other nodes periodically by PushUpdate (Fig. 5). The target node set expands as replies come back from the target nodes ⟨6⟩. The propagation finishes when all the target nodes reply ⟨4⟩.

The function IAmCoordinator tells whether the node is designated to coordinate a particular update. For now, it just returns true, meaning that any node can be a coordinator, and that an update is flooded among all the target nodes. Having multiple coordinators does not affect the correctness of the algorithm, but it surely wastes the network bandwidth — we improve this design in Section 5.2.

2.7 Deleting Retired Updates

While each update record occupies only a small space, it is stored even when the replica itself is removed. We need to delete updates in a timely manner; otherwise, the update records of deleted replicas will accumulate and eventually fill the disk up. An update is deleted from disk in two steps. The first step, performed periodically by PushRetire (Fig. 6), is the update *retirement*; the coordinator informs the target nodes that update propagation is complete. The second step is the update *removal* in which we apply the at-most-once messaging

[†] Markers such as ⟨1⟩ refer to lines in the program listings.

```
public proc PushUpdate()
  if gU≠NULL ∧ gU.state = ACTIVE
    if gU.done=gU.target |Update done ⟨4⟩
      gU.state ← RETIRING
      gU.done ← {me}
      gSavedData ← NULL
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.target - gU.done)
        if node ∈ gU.peers
          |Node will delete the replica.
          data ← NULL
        elif gData ≠ NULL
          |I store valid contents.
          data ← gData
        else
          data ← gSavedData
        Send(node, UpdateRequest,
             me, gU, data)

public proc UpdateRequest(caller, u, data)
  ok ← ApplyUpdate(u, data)
  Send(caller, UpdateReply,
       u.ts, ok, gU.done, gU.target)

public proc UpdateReply(ts, ok,
       done, target)
  if ¬UpdateOverwritten(ts) | ⟨5⟩
    if ¬ ok
      gU.state ← SUSPENDED;
      return
    gU.done ← gU.done ∪ done
    gU.target ← gU.target ∪ target | ⟨6⟩

proc UpdateOverwritten(ts): bool
  return gU=NULL |The update retired
         ∨ gU.ts > ts |A new update arrived

proc IAmCoordinator(u): bool
  return true
```

Fig. 5. Update propagation. PushUpdate is called periodically to push the newest update to participants. UpdateRequest is executed on remote nodes in response to PushUpdate. UpdateReply is called on the coordinator to handle replies from UpdateRequest.

algorithm [12] to remove retired updates without being confused by out-of-order update messages. Here, the node simply waits for *MAXDELAY* seconds before deleting a retired update (Fig. 6, RemoveUpdate). *MAXDELAY* is the sum of the maximum clock skew among nodes and the message lifetime, an interval long enough that almost all the messages will arrive at the destination within the interval.

This update removal scheme additionally requires each node to discard stale incoming network messages (Fig. 6 MessageArrived). Here, each network message is stamped with the sender's clock value and is accepted by the receiver only when its timestamp is no older than *MAXDELAY* on the receiver's clock.

3 Examples

We show two examples to illustrate the behavior of the algorithm. The first example is a simple contents update. The second example demonstrates the node discovery process used to resolve conflicting replica set changes.

Fig. 7 shows a sequence of steps performed to update an object replicated on nodes A, B, and C.

(1) A issues an update U:⟨target=peer={A,B,C}⟩ and modifies its replica.
(2) A pushes U to B and C.
(3) B applies U and returns ⟨*Uts*, true, {A,B}, {A,B,C}⟩ to A. C applies U and returns ⟨*Uts*, true, {A,C}, {A,B,C}⟩ to A.

```
public proc PushRetire()
  if gU≠NULL ∧ gU.state = RETIRING
    if gU.done=gU.target
      gRetireTime ← Now()
      gU.state ← RETIRED
    elif IAmCoordinator(gU)
      foreach node ∈ (gU.target - gU.done)
        Send(node, RetireRequest, me, gU.ts)
public proc RetireRequest(caller, ts)
  if ¬UpdateOverwritten(ts)
    gRetireTime ← Now()
    gU.state ← RETIRED
    gSavedData ← NULL
  Send(caller, RetireReply, me, ts)
public proc RetireReply(node, ts)
  if ¬UpdateOverwritten(ts)
    gU.done ← gU.done ∪ {node}
public proc RemoveUpdate()
  if gU≠NULL ∧ gU.state=RETIRED
     ∧ Now() >gRetireTime+MAXDELAY
    gU←NULL | Delete the update. ⟨7⟩
public proc MessageArrived(msg)
  if msg.ts < CurClock() - MAXDELAY
    return | message too old. just ignore it
  dispatch msg
```

Fig. 6. Update retirement. PushRetire is called periodically to push retirement notices to participants. RetireRequest is executed on remote node in response to PushRetire. RetireReply is called on the coordinator to handle replies from RetireRequest. RemoveRequest is called periodically to remove retired updates. MessageArrived is called for every incoming message to discard messages that are too old.

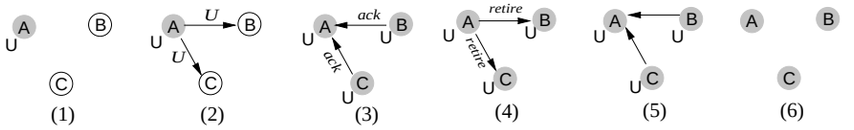


Fig. 7. An object replicated on nodes A, B, and C is updated. Gray circles indicate that nodes that have applied the update. The letter "U" indicates that the update is logged on the node.

(4) A receives the replies from B and C and changes *U.state* to RETIRING. A sends U's retirement to B and C.
(5) B and C change *U.state* to RETIRED and reply to A. A receives the replies from B and C and changes *U.state* to RETIRED.
(6) *MAXDELAY* seconds later, A, B, and C erase U from gU.

Fig. 8 shows a scenario in which an object replicated on nodes A and B is updated concurrently, first by A that adds C to the replica set, and next by B that adds D to the replica set. We assume that B's update is newer than A's.

(1) A issues U^A:⟨target=peer={A,B,C}⟩ and modifies its *gPeer*. Simultaneously, B issues U^B: ⟨target=peer={A,B,D}⟩ and modifies its *gPeer*.
(2) A pushes U^A to B and C. B pushes U^B to A and D. Now, for the sake of explanation, suppose C and D receive the updates before B and A do.
(3) C creates a replica and replies ⟨U^A.*ts*, true, {A,C}, {A,B,C}⟩ to A. D creates a replica and replies ⟨U^B.*ts*, true, {B,D}, {A,B,D}⟩ to B.
(4) B receives U^A from A. Because $U^A.ts < U^B.ts$, B rejects U^A and replies ⟨U^A.*ts*, false, {A,B} {A,B,C,D}⟩ to A. B's U^B.*target* becomes {A,B,C,D} (Fig. 4 ⟨1⟩). On receiving the reply from B, A changes U^A.*state* to SUSPENDED.

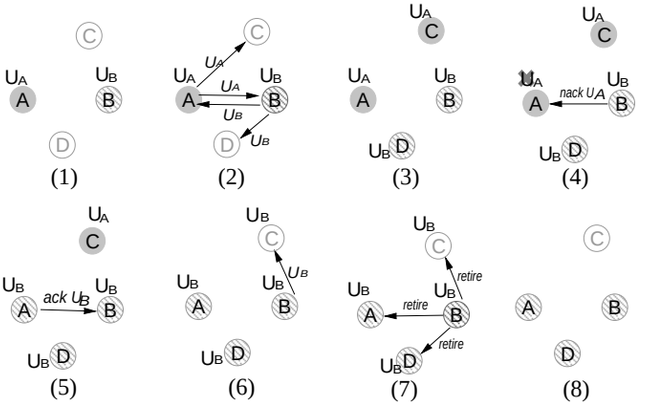


Fig. 8. Conflicting updates involving replica addition. Gray circles are nodes that applied U^A, and diagonally shaded circles are nodes that applied U^B.

(5) A receives U^B from B. Because $U^B.ts > U^A.ts$, A modifies *gPeer*, replaces *gU* with U^B, and replies $\langle U^B.ts$, true, {A,B}, {A,B,C,D}$\rangle$ to B. Later, A may receive a reply for U^A from C, but A ignores the reply (Fig. 5 $\langle 5 \rangle$).

(6) B receives the reply from A and pushes U^B to C, the only target not yet contacted. On reception of U^B, C recognizes that it is not a part of the new replica set, removes its replica, and replies $\langle U^B$, true, {A,B,C}, {A,B,C,D}$\rangle$ to B.

(7) B receives the replies from C and D. B changes $U^B.state$ to RETIRING and pushes U^B's retirement to A, C, and D. A, C, and D acknowledge the retirement.

(8) *MAXDELAY* seconds later, A, B, C, and D erase U^B from *gU*. In the end, A, B, and D store replicas. C created a replica and later deleted it.

4 Correctness Proof

While being simple, our algorithm contains several subtleties, especially regarding replica additions and deletions. For example, how does it guarantee that all replicas receive an update, when another update is adding replicas concurrently? In this section, we prove two main safety properties of the algorithm: all nodes receive the newest update at the end of propagation, and no stale updates are accepted by nodes regardless of concurrent updates. We also argue the liveness of the algorithm, i.e., all the replicas will receive the newest update, by applying our safety arguments.

The state of the system can be viewed as a directed graph, called the *knowledge graph*, in which the vertices represent nodes and the edges represent the nodes' knowledge of others through *gPeer* and *gU.targets*. The graph is usually complete (i.e., no update is issued and the value of *gPeer* is identical on all

Symbol	Meaning
n:*var*	The value of variable *var* on node n.
n:$Utarget,T$	The value of *Utarget* on node n just before T.
n:$Udone,T$	The value of *Udone* on node n just before T.
$n_1 \overset{G_T}{\rightarrow} n_2$	n_1 knows n_2 in G_T, i.e., $(n_1 \rightarrow n_2) \in E_{G_T}$.
$n_1 \overset{G_T}{\rightsquigarrow} n_2$	A path from n_1 to n_2 exists, i.e., $n_1 \overset{G_T}{\rightarrow} n_2 \vee \exists n', n_1 \overset{G_T}{\rightarrow} n' \wedge n' \overset{G_T}{\rightsquigarrow} n_2$.

Table 1. Notational conventions used in the proof

the replicas), but it becomes incomplete during replica addition or deletion. At a high level, our proof shows that the algorithm ripples the newest update through the graph, adding edges to the graph along the way to cover all the nodes and to restore the completeness of the graph eventually.

Following are notations used in the ensuing proof. Other symbols are summarized in Table 1.

- A node "*stores a replica*" when $gData \neq$NULL.
- A node "*has an update*" when its $gU.state$ is ACTIVE or RETIRING.
- A node "*retires an update*" when it sets $gU.state$ to RETIRED.
- A node n_1 "*knows*" another node n_2 when either n_1 has an update and $n_2 \in n_1$:$gU.target$, or $n_2 \in n_1$:$gPeer$.
- $G_T \equiv \langle V_{G_T}, E_{G_T} \rangle$, a *knowledge graph* for the object at time T [‡], is defined as follows.
  V_{G_T} = Nodes that store a replica or have an update.
  $E_{G_T} = \{v_1 \rightarrow v_2 \mid \{v_1, v_2\} \subseteq V_{G_T} \wedge v_1 \text{ knows } v_2\}$
- $S_T \equiv \langle V_{S_T}, E_{S_T} \rangle$, an *induced subgraph* of G_T, excludes from G_T vertices that correspond to failed nodes and the associated edges. S_T shows the knowledge graph in the presence of failure.

4.1 Correctness Criteria

Ideally, we want to prove that the algorithm keeps all the live replicas consistent regardless of types of failures. Such a guarantee, however, is impossible when nodes or links fail in a way that makes the corresponding induced subgraph disconnected. For example, suppose two nodes fail simultaneously after both have created two new replicas, as illustrated in Fig. 9. After such a failure, any update issued on the two new replicas will not reach each other, and the new replicas will remain inconsistent until the original two nodes recover. Therefore, we define the correctness only under the condition that a knowledge subgraph is at least weakly connected.

[‡] All times mentioned in the proof are hypothetical global times observed by an external agent.

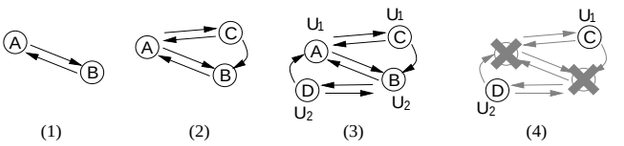


Fig. 9. A scenario that disconnects a graph. The edges show knowledge among nodes. (1) Replicas A and B initially know each other. (2) A issues U_1 and creates a replica C. (3) B issues U_2 and creates a replica D. (4) A crashes before U_1 is propagated to B or D. B crashes before U_2 is propagated to A or C. In the end, two components, {C} and {D}, are both live but disconnected.

Correctness criteria: *Suppose S_{T_s} is weakly connected, and no node or link fails and no new update is issued during a long enough period (T_s, T_e). Let U be the newest update generated before T_e. The algorithm is correct if the following conditions hold.*

(1) Every node $n \in U.peer$ applies U before T_e.
(2) No node $n \notin U.peer$ stores a replica at T_e.
(3) No update U' older than U (i.e., $U'.ts < U.ts$) is applied on any node after U is applied.

Notice that these criteria demand, in case of a graph disconnection, a replica consistency within each partition, and that as soon as the partitions re-integrate, all the replicas converge onto the globally newest state. Indeed, this set of criteria is as strong as any non-blocking replication algorithm can guarantee.

4.2 Graph Invariants

Theorem 1. *If S_T is strongly connected, then $\forall T' > T$, $S_{T'}$ is also strongly connected if no node or link fails during the period (T, T').*

Theorem 2. *If S_T is weakly connected, then $\forall T' > T$, $S_{T'}$ is weakly connected if no node or link fails during the period (T, T').*

Proof sketch. We show by induction that no transition on the subgraph can disconnect the subgraph. First, a replica creation will not disconnect the graph because a new replica is always created by "stemming out" from an existing replica (Section 2.4). Next, replica deletion does not change the graph shape because the update record is still stored on the same node. Finally, update retirement will not disconnect the graph because an update retires only after all the peer replicas have spanned the edges to one another. Theorem 2 can be proved exactly the same way. ■

Theorem 3. *G_T is strongly connected for all T.*

Proof. The graph is clearly connected in the base case in which no replica exists. Thus, G_T is connected for all T from Theorem 1. ■

4.3 All Replicas Receive the Newest Update

Now, we prove that all the nodes receive the newest update by distinguishing two cases. Theorem 4 proves that if an update retires, all the nodes must have received the update. Theorem 5 proves that when an update is unable to retire because some of its targets are dead, all the remaining nodes still receive the update. These two theorems together prove the correctness criteria (1) and (2).

Theorem 4. *Suppose S_{T_s} is weakly connected, and no node or link fails and no new update is issued during a long enough period (T_s, T_e). Let U be the newest update generated before T_e. If a coordinator c begins the retirement of U at time T $(T < T_e)$, then $V_{S_{T_e}} \subseteq c{:}Udone,_T$; that is, all the nodes in $V_{S_{T_e}}$ have received and applied U.*

Proof. We only need to prove that $V_{S_T} \subseteq c{:}Udone,_T$; if $V_{S_T} \subseteq c{:}Udone,_T$, then $V_{S_{T_e}} \subseteq V_{S_T}$ because no newer update is generated after T and no node possesses a stale update after T.

For the sake of contradiction, suppose $V_{S_T} \not\subseteq c{:}Udone,_T$.

Because S_T is weakly connected from Theorem 2, we can pick a node $p \in c{:}Udone,_T$ such that $\exists n \in V_{S_T} - c{:}Udone,_T$ and that either $n \overset{S_T}{\rightarrow} p$ or $p \overset{S_T}{\rightarrow} n$. Below, we show that no such pair of p and n can exist.

First, suppose a pair (p, n) with an edge $p \overset{S_T}{\rightarrow} n$ exists (Fig. 10 (a)). Let T_p be the time c propagated U to p $(T_p < T)$. First, the edge $p \overset{S_T}{\rightarrow} n$ must have been created at or before T_p, because U is the newest update and any other update that could have created $p \rightarrow n$ would have been rejected by p after T_p. On the other hand, $p \overset{S_T}{\rightarrow} n$ must have been created after T_p; otherwise, the edge $c \overset{S_T}{\rightarrow} n$ must be in the graph (Fig. 5 ⟨6⟩). Because of this contradiction, this pair (p, n) cannot exist. Second, suppose only a pair (p, n') with an edge $n' \overset{S_T}{\rightarrow} p$ exists (Fig. 10 (b)). From Theorem 3, a path $c \overset{G_T}{\rightsquigarrow} n'$ exists in the full graph G_T (remember, G_T may include dead nodes). Therefore, there exists a dead or uncommunicative node $q \in c{:}Utarget,_T$ along the path $c \overset{G_T}{\rightsquigarrow} n'$, and q makes U unable to retire in the first place. Therefore, this pair nodes (p, n') cannot exist as well. Therefore, for U to retire, $V_{S_T} \subseteq c{:}Udone,_T$. ■

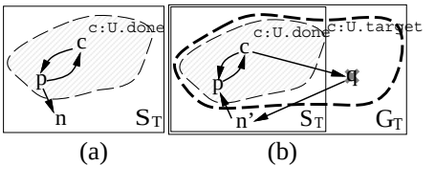


Fig. 10. A coordinator c may fail to contact a node n in two situations.

Theorem 5. *Suppose S_{T_s} is weakly connected, and no node or link fails and no new update is issued during a long enough period (T_s, T_e). Let U be the newest update generated before T_e. If $(c{:}Utarget,_{T_e} - c{:}Udone,_{T_e}) \cap V_{S_{T_e}} = \phi$ on a coordinator node c, then $V_{S_{T_e}} \subseteq c{:}Udone,_{T_e}$.*

Proof. For the sake of contradiction, suppose $V_{S_{T_e}} \not\subseteq c{:}Udone,_{T_e}$. Using the argument that appeared in the previous proof, we can pick a node $p \in c{:}Udone,_{T_e}$ such that $\exists n \in V_{S_{T_e}} - c{:}Udone,_T$ and either $n \overset{S_{T_e}}{\rightarrow} p$ or $p \overset{S_{T_e}}{\rightarrow} n$, and we can show that a pair (p, n) with an edge $p \overset{S_T}{\rightarrow} n$ cannot exist.

Now, suppose only a pair (p, n') with an edge $n' \overset{S_T}{\rightarrow} p$ exists. For the moment, lets assume the following lemma holds for any pair of nodes, n_1 and n_2.

Lemma 1. *If $n_1 \overset{S_T}{\rightarrow} n_2$ but not $n_2 \overset{S_T}{\rightarrow} n_1$, then n_1 has an update.*

From this lemma, n' has an update, say, U'. Thus, n' contacts p, which in turn causes n' to discover c (Fig. 5 ⟨6⟩), which in turn cause n' to propagate U' to c, thereby letting c discover n' before T_e (Fig. 4 ⟨1⟩). Therefore, a pair (p, n) with an edge $n \overset{S_T}{\rightarrow} p$ cannot exist as well. Thus, $V_{S_{T_e}} \subseteq c{:}U.done_{,T_e}$. ■

We sketch the proof of Lemma 1. Edges can disappear only when an update retires. For an update to retire, all the target nodes need to reply, that is, edges must span between any pair of the target nodes. Thus, when $n_1 \overset{S_T}{\rightarrow} n_2$ but not $n_2 \overset{S_T}{\rightarrow} n_1$, then n_1 must have an update. ■

4.4 No Node Receives a Stale Update

Theorem 6. *Suppose no update is generated for a long period ending at T_e. Let U be the newest update issued before T_e. After U retires, no update older than U is applied on any node.*

Proof. A node may receive an older update after it retires U for two potential reasons: (1) another node that has not received U propagates a stale update, or (2) a network delay causes a message containing a stale update to be received after U retires. Theorem 4 prevents the case (1). The use of synchronized clocks, described in Section 2.7, prevents the case (2) (refer to [12] for the full proof).

4.5 Liveness

Liveness is derived immediately from the work of the algorithm. The algorithm sends update or retire messages to the nodes in *gU.target* until it receives the replies from all the nodes. *GU.target* may grow as a result of reply processing (Fig. 4 ⟨1⟩), but because its size is finite, the coordinator will eventually push the update to all the nodes it can communicate with[§].

5 Extensions

5.1 Supporting Multiple Objects

All the discussions so far have focused on a single object, but in fact, the basic algorithm can be extended easily to support multiple objects. To support multiple objects, instead of the variables *gU*, *gSavedData*, and *gRetireTime*, we now have a persistent table that partially maps an object ID to an update in progress for the object. An update is added to the table when an object is going to be modified (Fig. 4 ⟨3⟩), and is deleted from the table when it is removed (Fig. 6 ⟨7⟩) or is superseded by a newer update for the same object (Fig. 4 ⟨3⟩).

[§] Here, we assume a bounded message transmission delay. Otherwise, no algorithm can ensure liveness.

5.2 Designated Coordinator

```
var members: NodeSet;
proc IAmCoordinator(u): bool
  return u.ts.nid = me
    ∨ min(members ∩ u.target) = me
```

Fig. 11. Designated coordinator selection. A membership service stores the set of presumed live nodes in *members*.

The basic algorithm presented so far is inefficient because it floods an update among all the target nodes in PushUpdate, causing an update to be sent as many as $(N-1)^2$ times (N is the number of replicas). By combining the use of a group membership service [5, 22] and a simple change to the function IAmCoordinator (Figures 5 and 11), however, we can reduce the cost to $N-1$ in the common case. In the new implementation, a node pushes or retires an update only when it is the issuer of the update or when it is designated to take over the failed issuer. Notice that because the membership service is shared by all the objects hosted on a node, its cost is amortized over many runs of the algorithm and becomes negligible.

5.3 Delaying Update Retirements

The algorithm is further optimized by delaying and aggregating calls to PushRetire for different objects to the same node. Delaying calls to PushRetire does not affect the replica consistency; it merely delays the deletion of the update record and increases the size of the update table.

5.4 Optimistic Deltas

Instead of pushing the entire object state every time, we can send *optimistic deltas* [2] to save the network and the computational cost. Here, a coordinator simply pretends that all the replicas for the object were consistent before the update and pushes only the difference between the old and the new contents (called the optimistic delta) along with the *fingerprint* of the old replica contents. On the receiver side, a node applies the update when its replica's fingerprint matches the update's; otherwise, the node requests a full contents transfer from the coordinator. This technique can reduce the cost of the algorithm, especially during replica set changes, in the common case without concurrent updates.

Fingerprint is any short bit-string that summarizes the replica contents. Applying a collision-resistant hash function (e.g., MD5) on the replica contents is one way to compute a fingerprint. A faster, more accurate, but slightly more space-consuming alternative is to store along with each replica a timestamp (Figures 1 and 4) that shows the last time the replica was modified, and to use the timestamp as a fingerprint.

5.5 Handling Long-term Failures

In the real world, computers often crash and never recover. Such nodes create an unbounded amount of backlog of updates that eventually fill up the disks on other nodes. Our algorithm handles such a situation automatically by purging nodes that remain down for too long.

When a node finds another node dead for more than a predefined *purge period* (e.g., one week), it pretends that it received affirmative replies from the dead node for all its jammed updates. The node then purges the dead node simply by removing the dead node's name from the replica sets of all the replicas stored on the node. To avoid having inconsistent data, when a node recovers after being down for the purge period or longer, it clears its disk contents and rejoins the cluster with a new node ID.

The only remaining problem is when nodes or links fail in such a way as to make the knowledge graph disconnected, and they remain failed until the purge deadline. In such case, the aforementioned scheme may make replicas permanently inconsistent. We argue below that such a scenario is highly unlikely to happen in practice.

How can a graph become disconnected? One cause of a graph disconnection is link failures (i.e., network partitioning). Another cause, which may happen without link failures, is multiple node failures combined with concurrent replica additions, illustrated in Fig. 9. Now, can a graph disconnection last until the purge period? The answer is no, for all practical purposes. First, a network partitioning would never last long because it is repaired simply by installing replacement parts. Second, the latter failure scenario would not happen in practice because it requires a combination of simultaneous replica creations and coincidental sudden long-term failures of multiple nodes — the window of vulnerability is very narrow for both.

6 Performance

6.1 Networking and Computational Overhead

With the optimizations described in Section 5.2, our algorithm pushes an update to N replicas in the common case and aggregates retirement notices into one batch notice. In total, the algorithm sends $2(1+\frac{1}{G})N$ messages per update, where G is the average aggregation factor for retirement notices. In Porcupine, the value of G is around 20 under heavy load, and the networking and the processing costs of our algorithm is close to $2N$ per update, which is the optimal number for an algorithm that does not batch (and thus delay) update propagation — $N + N$ messages are always needed to propagate and acknowledge an update to N nodes.

6.2 Space Overhead

This algorithm stores two types of data structures per replica in addition to the contents: the replica set (*gPeer* in Fig. 1), and the update record (*gU*, *gSavedData*, and *gRetireTime* in Fig. 1).

The replica set information consumes small space – typically a few bytes per replica – and it is stored only when the replica itself is present.

An update record is stored on disk only while the update is in progress. The space consumed by update records on a node is $(S + \alpha M/R)UD$. Here, $(S + \alpha M/R)$ shows the average space overhead of an update record on a node: S

is the size of gU and $gRetireTime$, α is the proportion of updates that shrink the replica set size, M is the average object size, and R is the average replication factor for objects — thus, $\alpha M/R$ shows the time-averaged space overhead of $gSavedData$. U is the average number of objects updated per second and D is the average update lifetime, including the deletion wait period (Section 2.7) and delays introduced by retirement-aggregation (Section 5.2). In Porcupine, $S \approx 60$ bytes, $\alpha \approx 1/50$, $M \approx 5000$ bytes, $R \approx 2$, $U \approx 30$, and $D \approx 120$. Thus, total amount of stable storage used is about four hundred kilobytes.

7 Conclusions

We have described a new decentralized replication algorithm designed for Internet servers in this paper. Following are key features of our algorithm.

- Eventual consistency under most failure types, e.g., node and link node failures and sudden node retirements.
- Any replica can issue updates any time.
- Support for dynamic replica addition and deletion.
- Minimal space overhead, especially, efficient object deletion.
- Minimal computational and networking overhead in the common case.

As future work, we plan to investigate the space, time, and computation complexities of the algorithm under update conflicts. In addition, we are studying the implementation of semantically richer operations, e.g., multi-object transactions, on top of our algorithm.

Acknowledgements

We thank Robert Grimm, Mike Swift, Brian Bershad, and the anonymous reviewers for giving us valuable comments that improved the quality of the paper immensely. This work was supported in part by DARPA grant F30602-97-2-0226 and NSF grant EIA-9870740.

References

1. Yair Amir. *Replication using group communication over a partitioned network.* PhD thesis, Hebrew University of Jerusalem, 1995.
2. Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *USENIX Annual Technical Conference*, 1997.
3. P. Bernstein and E. Newcomer. *Principles of Tranasction Processing.* Morgan Kaufmann, 1997.
4. K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems*, 5(1):272–314, February 1987.
5. Flaviu Christian and Frank Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, UC San Diego, 1995.

6. A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *ACM Symp. on Princ. of Distr. Computing*, pages 1–12, 1987.
7. David K. Gifford. Weighted voting for replicated data. In *7th Symp. on Operating Systems Principles*, pages 150–162, 1979.
8. M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *ACM Symp. on Princ. of Distr. Computing*, pages 229–237, 1999.
9. Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems*, 4(1):32–53, February 1986.
10. Peter J. Keleher. Decentralized replicated-object protocols. In *18th ACM Symp. on Princ. of Distr. Computing*, April 1999.
11. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
12. Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Trans. on Computer Systems*, 9(2):125–142, 1991.
13. Microsoft. *Windows 2000 Server Resource Kit.* Microsoft Press, 2000.
14. David L. Mills. Improved algorithms for synchronizing computer network clocks. In *SIGCOMM*, pages 317–327, London, UK, September 1994. ACM.
15. P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *ACM SIGCOMM*, August 1988.
16. K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *16th Symp. on Operating Systems Principles*, pages 288–301, October 1997.
17. M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the Internet. Technical Report HA6177000-980305-01-TM, AT&T Labs, March 1998.
18. David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing.* PhD thesis, UC Los Angeles, 1998. UCLA-CSD-970044.
19. Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: a highly scalable, cluster-based mail service. In *17th Symp. on Operating Systems Principles*, December 1999.
20. Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study. http://www.research.digital.com/wrl/projects/newsbench/, September 1998.
21. Robert Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, June 1979.
22. R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Middleware*, 1998.
23. O. Wolfson and S. Jajodia. Distributed algorithms for adaptive replication of data. In *11th ACM Symp. on Princ. of Database Systems*, pages 149–163, 1992.

Scalable Replication in Database Clusters

M. Patiño-Martínez, R. Jiménez-Peris

Technical University of Madrid
Facultad de Informática
Boadilla del Monte
Madrid, 28660, Spain
{mpatino,rjimenez}@fi.upm.es

B. Kemme, G. Alonso

Swiss Federal Institute of Technology
Department of Computer Science
ETH Zentrum (ETHZ)
CH-8092, Zürich, Switzerland
{kemme,alonso}@inf.ethz.ch

Abstract. In this paper, we explore data replication protocols that provide both fault tolerance and good performance without compromising consistency. We do this by combining transactional concurrency control with group communication primitives. In our approach, transactions are executed at only one site so that not all nodes incur in the overhead of producing results. To further reduce latency, we use an optimistic multicast technique that overlaps transaction execution with total order message delivery. The protocols we present in the paper provide correct executions while minimizing overhead and providing higher scalability.

1 Introduction

Conventional algorithms for database replication emphasize consistency and fault tolerance instead of performance [1]. As a result, database designers ignore these algorithms and use *lazy* replication instead, thereby compromising both fault-tolerance and consistency [2]. A way out of this dilemma [7, 6] is to combine database replication techniques with group communication primitives [4]. This approach has produced efficient *eager* replication protocols that guarantee consistency and increase fault tolerance. However, in spite of some suggested optimizations [9, 10], this new type of protocols still have two major drawbacks. One is the amount of redundant work performed at all sites. The other is the high abort rates created when consistency is enforced. In this paper, we address these two issues. First, we present a protocol that minimizes the amount of redundant work. Transactions, even those over replicated data, are executed at only one site. The other sites only install the final changes. With this, and unlike in existing replication protocols, the aggregated computing power actually increases as more nodes are added. This is a significant advantage in environments with expensive transaction processing (e.g., dynamic web pages). A negative aspect of this protocol is that it might abort transactions in order to guarantee serializability. To reduce the rate of aborted transactions while still providing consistency, we propose a second protocol based on a transaction reordering technique.

The paper is organized as follows. Section 2 introduces the system model. Sections 3 and 4 describe the algorithms. Section 5 discusses fault tolerance aspects. Section 6 contains the correctness proofs. Section 7 concludes the paper.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 315–329, 2000.

2 System Model

In a replicated database, a group of nodes $N = \{N_1, N_2, ..., N_n\}$, each containing the entire database, communicate by exchanging messages. Sites only fail by crashing (no byzantine failures) and there is always at least one node available.

2.1 Communication Model

The system uses various group communication primitives [4]. Regarding message ordering, we use a multicast primitive not providing any order, a primitive providing *FIFO order* (messages of one sender are delivered in FIFO order) and one providing a *total order* (all messages are delivered at all sites in the same order). In regard to fault-tolerance, we use both a *reliable delivery service* (whenever a message is delivered at an available site it will be delivered at all available sites) and a *uniform reliable delivery service* (whenever a message is delivered at any faulty or available site it will be delivered at all available sites). We assume a virtual synchronous system, where all group members perceive membership (view) changes at the same virtual time, i.e., two sites deliver exactly the same messages before installing a new view.

We use an aggressive version [9] of the optimistic total order broadcast presented in [10]. Each message corresponds to a transaction. Messages are optimistically delivered as soon as they are received and before the definitive ordering is established. With this, the execution of a transaction can overlap with the calculation of the total order. If the initial order is the same as the definitive order, the transactions can simply be committed. If the final order is different, additional actions have to be taken to guarantee consistency. This optimistic broadcast is defined by three primitives [9]. *To-broadcast*(m) broadcasts the message m to all the sites in the system. *Opt-deliver*(m) delivers message m optimistically to the application (with no order guarantees). *To-deliver*(m) delivers m definitively to the application (in a total order). This means, messages can be opt-delivered in a different order at each site, but are to-delivered in the same total order at all sites. A sequence of opt-delivered messages is a *tentative order*. A sequence of to-delivered messages is the *definitive order* or total order. Furthermore, this optimistic multicast primitive ensures that every to-broadcast message is eventually opt-delivered and to-delivered by every site in the system. It also ensures that no site to-delivers a message before opt-delivering it.

2.2 Transaction Model

Clients interact with the database by issuing transactions, i.e., partially ordered sets of read and write operations. Two transactions conflict if they access the same data item and at least one of them is a write. A history H of committed transactions is serial if it totally orders all transactions. Two histories H_1 and H_2 are conflict equivalent, if they are over the same set of transactions and order conflicting operations in the same way. A history H is serializable, if it is conflict equivalent to some serial history [1]. For replicated databases, the correctness

criterion is 1-copy-serializability [1]. Using this criterion, each copy must appear as a single logical copy and the execution of concurrent transactions must be equivalent to a serial execution over all the physical copies.

In this paper, concurrency control is based on *conflict classes* [9]. Each conflict class represents a partition of the data. Transactions accessing the same conflict class have a high probability of conflicts, as they can access the same data, while transactions in different partitions do not conflict and can be executed concurrently. In [9] each transaction must access a single *basic* conflict class (e.g., C_x). We generalize this model and allow transactions to access *compound conflict classes*. A compound conflict class is a non-empty set of basic conflict classes (e.g., $\{C_x, C_y\}$). We assume that the (compound) conflict class of a transaction is known in advance. Each site has a queue CQ_x associated to each basic conflict class C_x. When a transaction is delivered to a site, it is added to the queues of the basic conflict classes it accesses. This concurrency control mechanism is a simplified version of a lock table [3].

Each conflict class has a *master* site. We use a *read-one/write-all available* approach. Queries (read only transactions) can be executed at any site using a snapshot of the data (i.e., they do not interfere with update transactions). Update transactions are broadcast to all sites, however they are only executed at the master site of their conflict class. We say a transaction is *local* to the master site of its conflict class and is *remote* everywhere else.

3 Increasing Scalability

3.1 The Problem and a Solution

The scalability of data replication protocols heavily depends on the update ratio. To see why, consider a centralized system capable of processing t transactions per second. Now assume a system with n nodes, all of them identical to the centralized one. Assume that the fraction of updates is w. Assume the load of local transactions at a node is x transactions per second. Since nodes must also process the updates that come from other nodes, the following must hold: $x + w\ (n-1)\ x = t$, that is, a node processes x local transactions per second, plus the percentage of updates arriving at other nodes ($w\ x$) times the number of nodes. From here, the number of transactions that can be processed at each node is $x = t\ (1 + w\ (n-1))^{-1}$. The total capacity of the system is n times that expression which yields, with t normalized to 1, $n\ (1 + w\ (n-1))^{-1}$. This expression has a maximum of n when $w = 0$ (there are no updates) and a minimum of 1 when $w = 1$ (all operations are updates).

Thus, as the *update factor* w approaches 1, the total capacity of the system tends to that of a single node, independently of how many nodes are in the system. Note that the drop in system capacity is very sharp. For 50 nodes, $w = 0.2$ (20% updates) results in a system with a tenth of the nominal capacity.

This limitation can be avoided if transactions execute only at one site (the local site) and the other sites only install the corresponding updates. This requires significantly less than actually running the transactions as has been shown

in [8]. In order to guarantee consistency, the total order established by the to-delivery primitive is used as a guideline to serialize transactions. All sites see the same total order for update transactions. Thus, to guarantee correctness, it suffices for a site to ensure that conflicting transactions are ordered according to the definitive order. Transactions can be executed in different orders at different sites if they are not serialized with respect to each other.

When an update transaction T is submitted, it is multicast to all nodes. This message contains the entire transaction and it is first opt-delivered at all sites which can then proceed to add the corresponding entries in the local queues. Only the local site executes T: whenever T is at the head of any of its queues the corresponding operation is executed on a shadow copy of the data. With this, aborting a transaction simply requires to discard the shadow copies. When the transaction commits the shadow copies become the valid versions of the data.

When a transaction is to-delivered at a site, the site checks whether the definitive and tentative orders agree. If they agree, the transaction can be committed after its execution has completed. If they do not agree, there are several cases to consider. The first one is when the lack of agreement is with non-conflicting transactions. In that case, the ordering mismatch can be ignored. If the mismatch is with conflicting transactions, there are two possible scenarios. If no local transactions are involved, the transaction can simply be rescheduled in the queues before the transactions that are only opt-delivered but not yet to-delivered. With this, to-delivered transactions will then follow the definitive order. If local transactions are involved, the procedure is similar but local transactions (that have been executed in the wrong order) must be aborted and rescheduled again (by putting them back in the queues in the proper order).

Once a transaction is to-delivered and completely executed the local site broadcasts the commit message containing all updates (also called write set WS). Upon receiving a commit message (which does not need any ordering guarantee), a remote site installs the updates for a certain basic conflict class as soon as the transaction reaches the head of the corresponding queue. When all updates are installed the transaction commits.

3.2 Example

Assume there are two basic conflict classes C_x, C_y and two sites N and N'. N is the master of conflict classes $\{C_x\}$, and $\{C_x, C_y\}$. N' is the master of $\{C_y\}$. We denote the conflict class of a transaction T_i by C_{T_i}. Assume there are three transactions, $C_{T_1} = \{C_x, C_y\}$, $C_{T_2} = \{C_y\}$ and $C_{T_3} = \{C_x\}$. That is, T_1 and T_3 are local at N and T_2 is local at N'. The tentative order at N is: T_1, T_2, T_3 and at N' is: T_2, T_3, T_1. The definitive order at both sites is: T_1, T_2, T_3. When all the transactions have been opt-delivered, the queues at each site are as follows:

At N:	At N':
$CQ_x = T_1, T_3$	$CQ_x = T_3, T_1$
$CQ_y = T_1, T_2$	$CQ_y = T_2, T_1$

At site N, T_1 can start executing both its operations on C_x and C_y since it is at the head of the corresponding queues. When T_1 is to-delivered the orders

are compared. In this case, the definitive order is the same as the tentative order and hence, T_1 can commit. When T_1 has finished its execution, N will send a commit message with all the corresponding updates. N can then commit T_1 and remove it from the queues. The same will be done for T_3 even if, in principle, T_2 goes first in the final total order. However, since these two transactions do not conflict, this mismatch can be ignored. Parallel to this, when N receives the commit message for T_2 from N', the corresponding changes can be installed since T_2 is at the head of the queue CQ_y. Once the changes are installed, T_2 is committed and removed from CQ_y.

At site N', T_2 can start executing since it is local and at the head of its queue. However, when T_1 is to-delivered, N' realizes that it has executed T_2 out of order and will abort T_2, moving it back in the queue. T_1 is moved to the head of both queues. Since T_3 is remote at N', moving T_1 to the head of the queue CQ_x does not require to abort T_3. T_1 is now the first transaction in all the queues, but it is a remote transaction. Therefore, no transaction is executing at N'. When the commit message of T_1 arrives at N', T_1's updates are applied, T_1 commits and is removed from both queues. Then, T_2 will start executing again. When T_2 is to-delivered and completely executed, a commit message with its updates will be sent, and T_2 will be removed from CQ_y.

3.3 The NODO Algorithm

The first algorithm we propose, NODO (NOn-Disjoint conflict classes and Optimistic multicast), follows that in [9]. The algorithm is described according to the different phases in a transaction's execution: a transaction is opt-delivered, to-delivered, completes execution, and commits. We assume access to the queues is regulated by locks and latches [3]. There are some restrictions on when certain events may happen. For instance, a transaction can only commit when it has been executed and to-delivered. Waiting for the to-delivery is necessary to avoid conflicting serialization orders at the different sites. Each transaction has two state variables to ensure this behavior: The *execution state* of a transaction can be *active* (as soon as it is queued) or *executed* (when its execution has finished). A transaction can only become executed at its master site. The *delivery state* can be *pending* (it has not been to-delivered yet) or *committable* (it has been to-delivered). When a transaction is opt-delivered its state is set to active and pending. In the following we assume that whenever a transaction is local and the first one in any of its queues, the corresponding operations are submitted for execution.

We assume that each of the phases is done in an atomic step. For instance, adding a transaction to the different queues during opt-delivery or rescheduling transactions during to-delivery is not interleaved with any other action. Note that aborting a transaction simply involves discarding the shadow copy, the transaction itself is kept in the queues but in different positions.

```
Upon Opt-delivery of T_i
  Mark T_i as active and pending
  For each conflict class C_x ∈ C_{T_i}
    Append T_i to the queue CQ_x
  EndFor

Upon TO-delivery of T_i:
  Mark T_i as committable
  If T_i is executed then
    Broadcast commit (WS_{T_i})
  Else (still active or not local)
    For each C_x ∈ C_{T_i}
      If First(CQ_x) = T_j
        ∧ Local(T_j)
          ∧ Pending(T_j) then
        Abort T_j
        Mark T_j as active
      EndIf
      Schedule T_i before the first
      pending transaction in CQ_x
    EndFor
  EndIf

Upon complete execution of T_i
  If T_i is marked as committable then
    Broadcast commit(WS_{T_i})
  Else
    Mark T_i as executed
  EndIf

Upon receiving commit(WS_{T_i})
  If ¬ Local (T_i) then
    Delay until T_i becomes committable
    For each C_x ∈ C_{T_i}
      When T_i becomes the first in CQ_x
        Apply the updates of WS_{T_i}
          corresponding to C_x
        Remove T_i from CQ_x
      EndFor
    Else
      Remove T_i from all C_{T_i}
    EndIf
    Commit T_i
```

4 Reducing Transaction Aborts

In the NODO algorithm, a mismatch between the local optimistic order and the total order may result in a transaction being aborted. The resulting abort rate is not necessarily very high since for this to happen, the transactions must conflict, appear in the system at about the same time, and the site where the mismatch occurs must be the local site where the aborted transaction was executing. In all other cases there are no transaction aborts, only reschedulings. Nevertheless, network congestion and high loads can lead to messages not being spontaneously ordered and, thus, to higher abort rates. The number of aborted transactions can be reduced by taking advantage of the fact that NODO is a form of master copy algorithm (remote sites only install updates in the proper order). Thus, a local site can unilaterally decide to change the serialization order of two local transactions (i.e., follow the tentative order instead of the definitive total order), thereby avoiding the abort. To guarantee correctness, the local site must inform the rest of the sites about the new execution order (by appending this information to the commit message). Special care must be taken with transactions that belong to a non basic conflict class (e.g., $C_{T_i} = \{C_x, C_y\}$). A site can only follow the tentative order $T_1 \rightarrow_{OPT} T_2$ instead of the definitive order $T_2 \rightarrow_{TO} T_1$, if T_1's conflict class C_{T_1} is a subset of T_2's conflict class C_{T_2} and botho are local transactions. Otherwise, inconsistencies could occur. We call this new algorithm REORDERING as the serialization order imposed by the definitive order might be changed for the tentative one.

4.1 Example

Assume a database with two basic conflict classes C_x and C_y. Site N is the master of the conflict classes $\{C_x\}$ and $\{C_x, C_y\}$. N' is the master of conflict class $\{C_y\}$. To show how reordering takes place, assume there are three transactions $C_{T_1} = C_{T_3} = \{C_x, C_y\}$, and $C_{T_2} = \{C_x\}$. All three transactions are local to N. The tentative order at both sites is T_2, T_3, T_1. The definitive order is T_1, T_2, T_3. After opt-delivering all transactions they are ordered as follows at both sites:

QC_x : T_2, T_3, T_1
QC_y : T_3, T_1

At site N, T_2 and T_3 can start execution (they are local and are at the head of one of their queues). Assume that T_1 is to-delivered at this stage. In the NODO algorithm, T_1 would be put at the head of both queues which can only be done by aborting T_2 and T_3. This abort is, however, unnecessary since N controls the execution of these transactions and the other sites are simply waiting to be told what to do. Thus, N can simply decide not to follow the total order but serialize according to the tentative order. This is possible because all transactions involved are local and the conflict classes of T_2 and T_3 are a subset of T_1's conflict class. When such a reordering occurs, T_1 becomes the *serializer transaction* of T_2 and T_3. T_2 does now not need to wait to be to-delivered to commit. Being at the head of the queue and with its serializer transaction to-delivered, the commit message for T_2 can be sent once T_2 is completely executed (thereby reducing the latency for T_2). The commit message of T_2 also contains the identifier of the serializer transaction T_1. The same applies to T_3.

Site N' has at the beginning no information about the reordering. Thus, not knowing better, when T_1 is to-delivered at N', N' will reschedule T_1 before T_2 and T_3 as described in the NODO algorithm. However, when N' receives the commit message of T_2, it realizes that a reordering took place (since the commit message contains the information that T_2 has been serialized before T_1). N' will then reorder T_2 ahead of T_1 and mark it committable. N', however, only reschedules T_2 when T_1 has been to-delivered in order to ensure 1-copy serializability. The rescheduling of T_3 will take place when the commit message for T_3 arrives, which will also contain T_1 as the serializer transaction. In order to prevent that T_2 and T_3 are executed in the wrong order at N', commit messages are sent in FIFO order (note, that FIFO is not needed in the NODO algorithm).

As this example suggests, there are restrictions to when reordering can take place. To see this, consider three transactions $C_{T_1} = \{C_x\}$, $C_{T_2} = \{C_y\}$ and $C_{T_3} = \{C_x, C_y\}$. T_1 and T_3 are local to N, T_2 is local to N'. Now assume that the tentative order at N is T_3, T_1, T_2 and at N' it is T_1, T_2, T_3. The definitive total order is T_1, T_2, T_3. After all three transactions have been opt-delivered the queues at both sites look as follows:

Queues at site N:	Queues at site N':
QC_x : T_3, T_1	QC_x : T_1, T_3
QC_y : T_3, T_2	QC_y : T_2, T_3

Since T_3 is local and it is at the head of its queues, N starts executing T_3. For the same reasons, N' starts executing T_2. When T_1 is to-delivered at N, T_3 cannot

be reordered before T_1. Assume this would be done. T_3 would commit and the commit message would be sent to N'. Now assume the following scenario at N'. Before N' receives the commit message for T_3 both T_1 and T_2 are to-delivered. Since T_2 is local, it can commit when it is executed (and the commit is sent to N). Hence, by the time the commit message for T_3 arrives, N' will produce the serialization order $T_2 \rightarrow T_3$. At N, however, when it receives T_2's commit, it has already committed T_3. Thus, N has the serialization order $T_3 \rightarrow T_2$, which contradicts the serialization order at N'.

This situation arises because $C_{T_1} = \{C_x, C_y\}$ is not a subset of $C_{T_3} = \{C_x\}$ and, therefore, T_1 cannot be a serializer transaction for T_3. In order to clarify why subsets (i.e., the conflict class of the reordered transaction is a subset of the conflict class of the serializer transaction) are needed for reordering, assume that T_1 also accesses C_y (with this, $C_{T_3} \subseteq C_{T_1}$). In this case, the queues are:

Queues at site N:	Queues at site N':
$QC_x: \ T_3, T_1$	$QC_x: \ T_1, T_3$
$QC_y: \ T_3, T_1, T_2$	$QC_y: \ T_1, T_2, T_3$

The subset property guarantees that T_1 conflicts with any transaction with which T_3 conflicts. Hence, T_1 and T_2 conflict and N' will delay the execution and commitment of T_2 until the commit message of T_1 is delivered. As the commit message of the reordered transaction T_3 will arrive before the one of T_1, T_3 will be committed before T_1 and thus before T_2 solving the previous problem. This means, that both N and N' will produce the same serialization order $T_3 \rightarrow T_1 \rightarrow T_2$.

4.2 REORDERING Algorithm

In general, the REORDERING algorithm is similar to NODO except in a few points (in the following we omit the actions upon opt-delivery since they are they same as in the NODO algorithm). The commit message must now contain the identifier of the *serializer transaction* (denoted as Ser in the algorithm description) and follow a FIFO order. As in NODO, when a transaction T_i is to-delivered, the transaction is marked as committable. At T_i's local site, any non to-delivered local transaction T_j whose conflict class C_{T_j} is a subset of C_{T_i} and that precedes T_i in the queues (reorder set RS) is marked as committable (since now the commit order is no longer the definitive but the tentative order). Thus, it is possible that when a reordered transaction is to-delivered the transaction is already marked as committable or even has been committed. In this case the to-delivery message is ignored. Local non to-delivered conflicting transactions that cannot be reordered and have started execution are aborted (abort set, AS). When the to-delivered transaction is remote, the algorithm behaves as the NODO algorithm. Note that a remote reordered transaction T_i cannot commit at a site until its serializer transaction is to-delivered at that site. When this happens, T_i is rescheduled before its serializer transaction. The rescheduling together with the FIFO ordering ensure that remote transactions will commit at all sites in the same order in which they did at the local site.

```
Upon to-delivery of transaction T_i
If ¬ Committed(T_i) ∧ Pending(T_i) then
 (T_i has not been reordered)
 If Local(T_i) then
  If T_i is marked executed then
   Ser(T_i) = T_i (T_i is its own serializer)
   Broadcast commit(WS_{T_i}, Ser(T_i))
  Else (T_i has not finished yet)
   AS = {T_j | C_{T_j} ∩ C_{T_i} ≠ ∅ ∧ C_{T_j} ⊈ C_{T_i}
    ∧ ∃C_x ∈ C_{T_j} ∩ C_{T_i}: T_j = First(CQ_x)
    ∧ Pending(T_j) ∧ Local(T_j)}
   For each T_j ∈ AS
    (abort conflicting transactions that
    cannot be reordered)
    Abort T_j and mark it as active
   EndFor
   (try to reorder transactions)
   RS = {T_j | C_{T_j} ⊆ C_{T_i} ∧ T_j →_opt T_i
    ∧ Pending(T_j) ∧ Local(T_j)}
   For each T_j ∈ RS ∪ {T_i}
    in opt-delivery order
     Mark T_j as committable
     Ser(T_j) = T_i (T_i is serializer of T_j)
     Schedule T_j before the first pending
      transaction in all CQ_x | T_j ∈ C_x
   EndFor
  EndIf
 Else (It is a remote transaction)
  Mark T_i committable
  For each conflict class C_x ∈ C_{T_i}
   If T_j = First(CQ_x) ∧ Pending(T_j)
    ∧ Local(T_j) then
    Abort T_j and mark it as active
   EndIf
   Schedule T_i before the first transaction
    marked as pending in queue CQ_x
  EndFor
 EndIf
Else (transaction has been reordered)
 Ignore the message
EndIf
```

```
Upon complete execution of T_i
If T_i is marked as committable then
 Broadcast commit(WS_{T_i}, Ser(T_i))
Else
 Mark T_i as executed
EndIf

Upon receiving commit(WS_{T_i}, Ser(T_i))
If ¬ Local(T_i) then
 Delay until Ser(T_i) is committable
 If T_i ≠ Ser(T_i) then
  Mark T_i as committable
 EndIf
EndIf
For each C_x ∈ C_{T_i}
 If not Local(T_i) then
  If T_i ≠ Ser(T_i) then
   Reschedule T_i just
    before Ser(T_i) in CQ_x
  EndIf
  When T_i becomes the first in CQ_x
   apply the updates of WS_{T_i}
    corresponding to C_x
 EndIf
 Remove T_i from CQ_x
EndFor
Commit T_i
```

5 Dealing with Failures

In our system, each site acts as a primary for the conflict classes it owns and as a backup for all other conflict classes. In the event of site failures, the available sites simply have to select a new master for the conflict classes of the failed node. The

new master will also take over the responsibility for all pending transactions for which the failed node was the owner (i.e., where the commit message has not been received by the available sites). Such a master replacement algorithm guarantees the availability of transactions in the presence of failures. That is, a transaction will commit as far as there is at least one available site.

For both algorithms, transaction messages must be uniformly multicast because only then it is guaranteed that the master will only execute and commit a transaction when all sites will receive it, and thus, be able to take over if the master crashes (reliable multicast does not provide this since the master can commit a transaction which the other sites have not yet received).

In the NODO algorithm, commit messages do not need to be uniform. Local transactions can even be committed before multicasting the commit message. The worst that can happen is that a master commits a transaction and fails before the commit message reaches the other sites. When a new master takes over, it will reexecute the transaction and send a new commit message. As the total order is always followed inconsistencies cannot arise.

In the REORDERING algorithm, commit messages must be uniform and the master may not commit the transaction before the commit message is delivered. If the commit message were not uniform, a master could reorder a transaction, send the commit message and then crash. If the rest of the replicas do not see the commit message, they would use a different serialization order (as the failed node's optimistic order is unknown to the other sites).

6 Correctness

In this section we prove the correctness (i.e., 1-copy-serializability), liveness, and consistency of the protocols. The proofs assume histories encompassing several group views. Important for both protocols is the fact that transactions are enqueued (respectively rescheduled) in one atomic step. Hence, there is no interleaving between transactions and all sites produce automatically serializable histories. As a result, in order to prove 1-copy-serializability, it suffices to show that all histories are conflict equivalent. Since conflict equivalence requires histories to have the same set of transactions, we refer in the corresponding proofs only to the available sites.

6.1 Correctness of NODO

We will show that all sites order conflicting transactions according to the definitive total order.

Definition 1 (Direct conflict). *Two transactions T_1 and T_2 are in direct conflict if they are serialized with respect to each other, $T_1 \longrightarrow T_2$, and there are no transactions serialized between them: $\nexists T_3 \mid T_1 \longrightarrow T_3 \longrightarrow T_2$.*

Lemma 1 (Total order and Serializability in NODO). *Let H_N be the history produced at site N, let T_1 and T_2 be two directly conflicting transactions in H_N. If $T_1 \longrightarrow_{TO} T_2$ then $T_1 \longrightarrow_{H_N} T_2$.*

Proof (lemma 1): Assume the lemma does not hold, i.e., there is a pair of transactions T_1, T_2 such that $T_1 \longrightarrow_{H_N} T_2$ but $T_2 \longrightarrow_{TO} T_1$. The fact that T_2 precedes T_1 in the total order means that T_2 was to-delivered before T_1. Since T_1 and T_2 are in direct conflict, there was at least one queue where both transactions had entries. If $T_1 \longrightarrow_{H_N} T_2$, then the entry for T_1 must have been ahead in the queue. However, upon to-delivery of T_2, if T_1 was the first transaction, NODO would have aborted T_1 and rescheduled it after T_2. If T_1 was not the first in the queue, NODO would have put T_2 ahead of T_1 in the queue. In both cases this would result in $T_2 \longrightarrow_{H_N} T_1$ which contradicts the initial assumption. □

Lemma 2 (Conflict equivalence in NODO). *For any two sites N and N', H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 2) From Lemma 1, all pairs of directly conflicting transactions in both H_N and $H_{N'}$ are ordered according to the total order. Thus, H_N and $H_{N'}$ are conflict equivalent since they are over the same set of transactions and order conflicting transactions in the same way. □

Theorem 1 (1CPSR in NODO). *The* NODO *algorithm produces 1-copy-serializable histories.*

Proof: (theorem 1) Since the histories of all available nodes are conflict equivalent (lemma 2) and serializable, the global history is 1-copy-serializable. □

6.2 Liveness of NODO

Theorem 2 (Liveness in NODO). *Each to-delivered transaction T_i eventually commits in the absence of catastrophic failures.*

Proof: (theorem 2) The theorem is proved by induction.
Induction Basis: Let T_i be the first to-delivered transaction. Upon to-delivery, each site places T_i at the head of all its queues. Thus, T_i's master can execute and commit T_i, and then multicast the commit message. Remote sites will apply the updates and also commit T_i.
Induction Hypothesis: The theorem holds for the to-delivered transactions with positions $n \leq k$, for some $k \geq 1$, in the definitive total order, i.e., all transactions that have at most $k-1$ preceding transactions will eventually commit.
Induction Step: Assume that transaction T_i is at position $n = k+1$ in the definitive total order when it is to-delivered. Each node places T_i in the corresponding queues after any committable transaction (to-delivered before T_i) and before any pending transaction (not yet to-delivered). All committable transactions that are now ordered before T_i have lower positions in the definitive total order. Hence, they will all commit according to the induction hypothesis and be removed from the queues. With this, T_i will eventually be the first in each of its queues and, as in the induction basis, eventually commit.

In all cases, if the master fails before the other sites have received the commit, the new master will reexecute T_i and resend the commit message. □

6.3 Consistency of NODO

Failed sites obviously do not receive the same transactions as available sites. Let $\mathcal{T}$ be the subset of transactions to-delivered to a node before it failed.

Theorem 3 (Consistency of failed sites with NODO). *All transactions, T_i, $T_i \in \mathcal{T}$, that are committed at a failed node N are committed at all available nodes. Moreover, the committed projection of the history in N is conflict equivalent to the committed projection of the history of any of the available nodes when this history is restricted to the transactions in $\mathcal{T}$.*

Proof: (theorem 3) A transaction T_i can only commit at N when it is to-delivered. Since we use uniform reliable delivery, T_i will also be to-delivered and known at all available sites. If T_i was not local at N, then N must have received a commit message from T_i's master. If this master is available for sufficient time all other available sites will also receive the commit message. If the master fails a new master will take over, execute T_i and resend the commit. This procedure will repeat if the new master also fails before the rest of the system receives the commit message. Since we assume there are some available nodes, eventually one of these nodes will become the master and the transaction will commit. If the transaction was local at N, the same argument applies. The equivalence of histories follows directly from Lemma 2. □

6.4 Correctness of REORDERING

In the REORDERING algorithm it is not possible to use the total order as a guideline since nodes can reorder local transactions. Thus, we start by proving that transactions not involved in a reordering cannot get in between the serializer and the transaction being reordered. Let T_s be the serializer transaction of the transactions in the set $\mathcal{T}_{T_s}$.

Lemma 3 (Reordered). *A reordered transaction T_i is always serialized before its serializer transaction T_s, that is, if $T_i \in \mathcal{T}_{T_s}$ then $T_i \longrightarrow T_s$.*

Proof (lemma 3): It follows trivially from the algorithm. □

Lemma 4 (Serializer in REORDERING). *For all transactions T_i, $T_i \in \mathcal{T}_{T_s}$ there is no transaction T_j, $T_j \notin \mathcal{T}_{T_s}$, such that $T_i \longrightarrow T_j \longrightarrow T_s$.*

Proof (lemma 4): Assume that N is the master site where the reordering takes place. Since T_s is the serializer of T_i, $T_i \longrightarrow_{OPT} T_s$, and $T_s \longrightarrow_{TO} T_i$. Additionally, from Lemma 3 $T_i \longrightarrow T_s$. There are two cases to consider: (a) $T_j \longrightarrow_{TO} T_s$ and (b) $T_s \longrightarrow_{TO} T_j$.
(a): since T_j is to-delivered before T_s, in the queues T_j is before T_i, and T_i is before T_s. With T_j ahead of their queues, T_i and T_s cannot be committed until T_j commits. Thus, T_j cannot be serialized in between T_i and T_s.
(b): since T_s is to-delivered before T_j and $T_j \notin \mathcal{T}_{T_s}$, all sites will put T_s ahead of T_j in the queues (T_j cannot have committed because it has not yet been

to-delivered), if it was not the case. Since $C_{T_i} \subseteq C_{T_s}$, this effectively prevents transactions from getting in between T_i and T_s. Any transaction T_j trying to do so will conflict with T_s and since T_s has been to-delivered before T_j, T_j has to wait until T_s commits. By that time, T_i will have committed at its master site and its commit message will have been delivered and processed at all sites before the one of T_s. Therefore, the final serialization order will be $T_i \longrightarrow T_s \longrightarrow T_j$. □

Lemma 5 (Conflict Equivalence in REORDERING). *For any two sites N and N', H_N is conflict equivalent to $H_{N'}$.*

Proof: (lemma 5) We show that two directly conflicting transactions T_1 and T_2 with conflict classes C_{T_1} and C_{T_2} are ordered in the same way at N and N'. We have to distinguish several cases:

• $C_{T_1} \subseteq C_{T_2}$, T_1 and T_2 have the same master N'', and $T_2 \longrightarrow_{TO} T_1$:

(a) If N'' reorders T_1 and T_2 with respect to the total order, then, from Lemma 4, no transaction $T_i \notin \mathcal{T}_{T_2}$ can be serialized in between. The commit for T_1 will be sent before the commit for T_2 in FIFO order. Hence, all sites will then execute T_1 before T_2.

(b) If N'' follows the total order to commit T_1 and T_2, then other sites cannot change this order. The argument is similar to that in Lemma 1 and revolves about the order in which transactions are committed at all sites.

• $C_{T_1} \subseteq C_{T_2}$, T_1 and T_2 have the same master N'', and $T_1 \longrightarrow_{TO} T_2$:

(c) If $C_{T_1} = C_{T_2}$ then cases (a) and (b) apply exchanging T_1 and T_2.

(d) Otherwise $C_{T_1} \subset C_{T_2}$. In this case, N'' has no choice but to commit T_1 and T_2 in to-delivery order (the rules for reordering do not apply). From here, and using the same type of reasoning as in Lemma 1, it follows that all sites must commit T_1 before T_2.

• either $C_{T_1} \subseteq C_{T_2}$ and T_1 and T_2 do not have the same master, or $C_{T_1} \cap C_{T_2} \neq \emptyset$ and neither $C_{T_1} \nsubseteq C_{T_2}$ nor $C_{T_2} \nsubseteq C_{T_1}$.

(e) If T_1 or T_2 are involved in any type of reordering at their nodes, Lemma 4 guarantees that there will be no interleavings between the transactions involved in the reordering and the other transaction. Thus, one transaction will be committed before the other at all sites and, therefore, all sites will produce the same serialization order.

(f) If T_1 and T_2 are not involved in any reordering, then similar to Lemma 1, both of them will be scheduled in the same (total) order at all sites and then committed.

• $C_{T_1} \cap C_{T_2} = \emptyset$.

(g) If there is no serialization order between T_1 and T_2 then they do not need to be considered for equivalence.

(h) If there is a serialization order between T_1 and T_2, it can only be indirect. Assume that in N: $T_1 \ldots \longrightarrow T_i \longrightarrow T_{i+1} \longrightarrow \ldots T_2$. Between each pair of transactions in that sequence, there is a direct conflict. Thus, for each pair, the above cases apply and all sites order the pair in the same way. From here it follows that T_1 and T_2 are also ordered in the same way at all sites. □

Theorem 4 (1CPSR in REORDERING). *The* REORDERING *algorithm produces 1-copy-serializable histories.*

Proof: (theorem 4) From Lemma 5, all histories are conflict equivalent. Moreover, they are all serializable. Thus, the global history is 1-copy-serializable. □

6.5 Liveness of REORDERING

Theorem 5 (Liveness in REORDERING). *Each to-delivered transaction T_i eventually commits in the absence of catastrophic failures.*

Proof: (theorem 5) The proof is by induction.
Induction Basis: Let T_i be the first to-delivered transaction. Upon to-delivery, each remote site will place T_i at the head of all its queues. At the local node, there might be some reordered transactions before T_i hence, T_i will be their serializer. All these transactions can be executed and committed, so that T_i will eventually be executed and committed. Remote sites will apply the updates of the reordered transactions and T_i in FIFO order and will also commit T_i.
Induction Hypothesis: The theorem holds for the to-delivered transactions with positions $n \leq k$, for some $k \geq 1$, in the definitive total order, i.e., all transactions that have at most $k-1$ preceding transactions will eventually commit.
Induction Step: Assume that transaction T_i is at position $n = k+1$ in the definitive total order when it is to-delivered. There are two cases:

a) *T_i is reordered.* This means there is a serializer transaction T_j with a position $n \leq k$ in the total order and T_i is ordered before T_j. Since T_j, according to the induction hypothesis, commits and T_i is executed and committed before T_j at all sites, the theorem holds.

b) *T_i is not a reordered transaction.* T_i will be rescheduled after any committable transaction and before any pending transaction. There exist two types of committable transactions rescheduled before T_i.

i. *Not reordered transactions:* They have a position $n \leq k$ and will therefore commit and be removed from the queues according to the induction hypothesis.

ii. *Reordered transactions:* Each reordered transaction that is serialized by transaction $T_k \neq T_i$ will commit before T_k and T_k will commit according to the previous point (i). All transactions $T_j \in \mathcal{T}_{T_i}$ (i.e., T_i is the serializer) are ordered directly before T_i in the queues (Lemma 3). Let T_k be the first not reordered transaction before this set of reordered transactions. T_k will eventually commit according to the previous point (i), and therefore also all transactions in $\mathcal{T}_{T_i}$ and T_i itself.

Failures lead to masters reassignment but do not introduce different cases to the above ones. □

6.6 Consistency of REORDERING

Again, let $\mathcal{T}$ be the subset of transactions to-delivered to a node before it failed.

Theorem 6 (Consistency of failed sites with REORDERING). *All transactions, T_i, $T_i \in \mathcal{T}$, that are committed at a failed node N are committed at all available nodes. Moreover, the committed projection of the history in N, is conflict equivalent to the committed projection of the history of any of the available nodes when this history is restricted to the transactions in $\mathcal{T}$.*

Proof: (theorem 6) Since both transaction and commit messages are sent with uniform reliable multicast, all transactions and their commit messages in $\mathcal{T}$ have been to-delivered to all available sites and can therefore commit at all sites. The equivalence of histories, follows directly from Lemma 5. □

7 Conclusions

In this paper, we have proposed two replication protocols for cluster based applications. These protocols solve the scalability problem of existing solutions and minimize the number of aborted transactions. We are currently implementing and experimentally evaluating the protocols and, as part of future work, we will deploy a web farm with a replicated database built upon these protocols. For this purpose we will use TransLib [5], a group-based TP-monitor.

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, MA, 1987.
2. 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.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, CA, 1993.
4. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison Wesley, Reading, MA, 1993.
5. R. Jiménez Peris, M. Patiño Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.
6. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, to appear.
7. B. Kemme and G. Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *Proc. of 18th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 156–163, 1998.
8. B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.
9. B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.
10. F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kutten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.

Disk Paxos

Eli Gafni[1] and Leslie Lamport[2]

[1] Computer Science Department, UCLA
[2] Compaq Systems Research Center

Abstract. We present an algorithm, called Disk Paxos, for implementing a reliable distributed system with a network of processors and disks. Like the original Paxos algorithm, Disk Paxos maintains consistency in the presence of arbitrary non-Byzantine faults. Progress can be guaranteed as long as a majority of the disks are available, even if all processors but one have failed.

1 Introduction

Fault tolerance requires redundant components. Maintaining consistency in the event of a system partition makes it impossible for a two-component system to make progress if either component fails. There are innumerable fault-tolerant algorithms for implementing distributed systems, but all that we know of equate *component* with *processor*. But there are other types of components that one might replicate instead. In particular, modern networks can now include disk drives as independent components. Because commodity disks are cheaper than computers, it is attractive to use them as the replicated components for achieving fault tolerance. Commodity disks differ from processors in that they are not programmable, so we can't just substitute disks for processors in existing algorithms.

We present here an algorithm called *Disk Paxos* for implementing an arbitrary fault-tolerant system with a network of processors and disks. It maintains consistency in the event of any number of non-Byzantine failures. That is, the algorithm tolerates faulty processors that pause for arbitrarily long periods, fail completely, and possibly restart; and it tolerates lost and delayed messages. Disk Paxos guarantees progress if the system is stable and there is at least one nonfaulty processor that can read and write a majority of the disks. Stability means that each processor is either nonfaulty or has failed completely, and nonfaulty processors can access nonfaulty disks.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 330–344, 2000.

Disk Paxos is a variant of the classic Paxos algorithm [3, 10, 12], a simple, efficient algorithm that has been used in practical distributed systems [13, 16]. Classic Paxos can be viewed as an implementation of Disk Paxos in which there is one disk per processor, and a disk can be accessed directly only by its processor.

In the next section, we recall how to reduce the problem of implementing an arbitrary distributed system to the consensus problem. Section 3 informally describes Disk Synod, the consensus algorithm used by Disk Paxos. It includes a sketch of an incomplete correctness proof and explains the relation between Disk Synod and the Synod protocol of classic Paxos. Section 4 briefly discusses some implementation details and contains the conventional concluding remarks. An appendix gives formal specifications of the consensus problem and the Disk Synod algorithm. Further discussion of the specifications and a sketch of a rigorous correctness proof appear in [5].

2 The State-Machine Approach

The state-machine approach [6, 14] is a general method for implementing an arbitrary distributed system. The system is designed as a deterministic state machine that executes a sequence of commands, and a consensus algorithm ensures that, for each n, all processors agree on the n^{th} command. This reduces the problem of building an arbitrary system to solving the consensus problem. In the consensus problem, each processor p starts with an input value $input[p]$, and all processors output the same value, which equals $input[p]$ for some p. A solution should be:

Consistent All values output are the same.

Nonblocking If the system is stable and a nonfaulty processor can communicate with a majority of disks, then the processor will eventually output a value.

It has long been known that a consistent, nonblocking consensus algorithm requires a three-phase commit protocol [15], with *voting*, *prepare to commit*, and *commit* phases. Nonblocking algorithms that use fewer phases don't guarantee consistency. For example, the group communication algorithms of Isis [2] permit two processors belonging to the current group to disagree on whether a message was broadcast in a previous group to which they both belonged. This algorithm cannot, by itself, guarantee consistency because disagreement about whether a message had been broadcast can result in disagreement about the output value.

The classic Paxos algorithm [3, 10, 12] achieves its efficiency by using a three-phase commit protocol, called the *Synod* algorithm, in which the value to be committed is not chosen until the second phase. When a new leader is elected, it executes the first phase just once for the entire sequence of consensus algorithms performed for all later system commands. Only the last two phases are performed separately for each individual command.

In the *Disk Synod* algorithm, the consensus algorithm used by Disk Paxos, each processor has an assigned block on each disk. The algorithm has two phases.

In each phase, a processor writes to its own block and reads each other processor's block on a majority of the disks.[1] Only the last phase needs to be executed anew for each command. So, in the normal steady-state case, a leader chooses a state-machine command by executing a single write to each of its blocks and a single read of every other processor's blocks.

The classic result of Fischer, Lynch, and Patterson [4] implies that a purely asynchronous nonblocking consensus algorithm is impossible. So, real-time clocks must be introduced. The typical industry approach is to use an *ad hoc* algorithm based on timeouts to elect a leader, and then have the leader choose the output. It is easy to devise a leader-election algorithm that works when the system is stable, which means that it works most of the time. It is very hard to make one that always works correctly even when the system is unstable. Both classic Paxos and Disk Paxos also assume a real-time algorithm for electing a leader. However, the leader is used only to ensure progress. Consistency is maintained even if there are multiple leaders. Thus, if the leader-election algorithm fails because the network is unstable, the system can fail to make progress; it cannot become inconsistent. The system will again make progress when it becomes stable and a single leader is elected.

3 An Informal Description of Disk Synod

We now informally describe the Disk Synod algorithm and explain why it works. (A formal specification appears in the appendix.) We also discuss its relation to classic Paxos's Synod Protocol. Remember that, in normal operation, only a single leader will be executing the algorithm. The other processors do nothing; they simply wait for the leader to inform them of the outcome. However, the algorithm must preserve consistency even when it is executed by multiple processors, or when the leader fails before announcing the outcome, and a new leader is chosen.

3.1 The Algorithm

We assume that each processor p starts with an input value $input[p]$.[2] As in Paxos's Synod algorithm, a processor executes a sequence of numbered ballots, with increasing ballot numbers. A ballot number is a positive integer, and different processors use different ballot numbers. For example, if the processors are numbered from 1 through N, then processor i could use ballot numbers i, $i+N$, $i+2N$, etc. A ballot has two phases:

Phase 1 Choose a value v.
Phase 2 Try to commit v.

[1] There is also an extra phase that a processor executes when recovering from a failure.
[2] If processor p fails, it can restart with a new value of $input[p]$.

In either phase, a processor aborts its ballot if it learns that another processor has begun a higher-numbered ballot. In that case, the processor may then choose a higher ballot number and start a new ballot. If the processor completes phase 2 without aborting—that is, without learning of a higher-numbered ballot—then value v is *committed* and the processor can output it. Since a processor does not choose the value to be committed until phase 2, phase 1 can be performed once for any number of separate instances of the algorithm.

To ensure consistency, we must guarantee that two different values cannot be successfully committed—either by different processors or by the same processor in two different ballots. To ensure that the algorithm is nonblocking, we must guarantee that, if there is only a single processor p executing it, then p will eventually commit a value.

In practice, when a processor successfully commits a value, it will write on its disk block that the value was committed and also broadcast that fact to the other processors. If a processor learns that a value has been committed, it will abort its ballot and simply output the value. It is obvious that this optimization preserves correctness; we will not consider it further.

To execute the algorithm, a processor p maintains a record $dblock[p]$ containing the following three components:

mbal The current ballot number.
bal The largest ballot number for which p reached phase 2.
inp The value p tried to commit in ballot number *bal*.

Initially, *bal* equal 0, *inp* equals a special value *NotAnInput* that is not a possible input value, and *mbal* is any ballot number. We let $disk[d][p]$ be the block on disk d in which processor p writes $dblock[p]$. We assume that reading and writing a block are atomic operations.

Processor p executes phase 1 or 2 of a ballot as follows. For each disk d, it tries first to write $dblock[p]$ to $disk[d][p]$ and then to read $disk[d][q]$ for all other processors q. It aborts the ballot if, for any d and q, it finds $disk[d][q].mbal > dblock[p].mbal$. The phase completes when p has written and read a majority of the disks, without reading any block whose *mbal* component is greater than $dblock[p].mbal$. When it completes phase 1, p chooses a new value of $dblock[p].inp$, sets $dblock[p].bal$ to $dblock[p].mbal$ (its current ballot number), and begins phase 2. When it completes phase 2, p has committed $dblock[p].inp$.

To complete our description of the two phases, we now describe how processor p chooses the value of $dblock[p].inp$ that it tries to commit in phase 2. Let *blocksSeen* be the set consisting of $dblock[p]$ and all the records $disk[d][q]$ read by p in phase 1. Let *nonInitBlks* be the subset of *blocksSeen* consisting of those records whose *inp* field is not *NotAnInput*. If *nonInitBlks* is empty, then p sets $dblock[p].inp$ to its own input value $input[p]$. Otherwise, it sets $dblock[p].inp$ to $bk.inp$ for some record bk in *nonInitBlks* having the largest value of $bk.bal$.

Finally, we describe what processor p does when it recovers from a failure. In this case, p reads its own block $disk[d][p]$ from a majority of disks d. It then sets $dblock[p]$ to any block bk it read having the maximum value of $bk.mbal$, and it starts a new ballot by increasing $dblock[p].mbal$ and beginning phase 1.

3.2 Why the Algorithm Works

Suppose processor p can read and write a majority of the disks, and all processors other than p stop executing the algorithm. In this case, p will eventually choose a ballot number greater than the *mbal* field of all blocks on the disks it can read, and its ballot will succeed. Hence, this algorithm is nonblocking, in the sense explained above.

We now explain, intuitively, why the Disk Synod algorithm maintains consistency. First, we consider the following shared-memory version of the algorithm that uses single-writer, multiple-reader regular registers.[3] Instead of writing to disk, processor p writes $dblock[p]$ to a shared register; and it reads the values of $dblock[q]$ for other processors q from the registers. A processor chooses its *bal* and *inp* values for phase 2 the same way as before, except that it reads just one *dblock* value for each other processor, rather than one from each disk. We assume for now that processors do not fail.

To prove consistency, we must show that, for any processors p and q, if p finishes phase 2 and commits the value v_p and q finishes phase 2 and commits the value v_q, then $v_p = v_q$. Let b_p and b_q be the respective ballot numbers on which these values are committed. Without loss of generality, we can assume $b_p \leq b_q$. Moreover, using induction on b_q, we can assume that, if any processor r starts phase 2 for a ballot b_r with $b_p \leq b_r < b_q$, then it does so with $dblock[r].inp = v_p$.

When reading in phase 2, p cannot have seen the value of $dblock[q].mbal$ written by q in phase 1—otherwise, p would have aborted. Hence p's read of $dblock[q]$ in phase 2 did not follow q's phase 1 write. Because reading follows writing in each phase, this implies that q's phase 1 read of $dblock[p]$ must have followed p's phase 2 write. Hence, q read the current (final) value of $dblock[p]$ in phase 1—a record with *bal* field b_p and *inp* field v_p. Let bk be any other block that q read in its phase 1. Since q did not abort, $b_q > bk.mbal$. Since $bk.mbal \geq bk.bal$ for any block bk, this implies $b_q > bk.bal$. By the induction assumption, we obtain that, if $bk.bal \geq b_p$, then $bk.inp = v_p$. Since this is true for all blocks bk read by q in phase 1, and since q read the final value of $dblock[p]$, the algorithm implies that q must set $dblock[q].inp$ to v_p for phase 2, proving that $v_p = v_q$.

To obtain the Disk Synod algorithm from the shared-memory version, we use a technique due to Attiya, Bar-Noy, and Dolev [1] to implement a single-writer, multiple reader register with a network of disks. To write a value, a processor writes the value together with a version number to a majority of the disks. To read, a processor reads a majority of the disks and takes the value with the largest version number. Since two majorities of disks contain at least one disk in common, a read must obtain either the last version for which the write was completed, or else a later version. Hence, this implements a regular register. With this technique, we transform the shared-memory version into a version for a network of processors and disks.

[3] A regular register is one in which a read that does not overlap a write returns the register's current value, and a read that overlaps one or more writes returns either the register's previous value or one of the values being written [7].

The actual Disk Synod algorithm simplifies the algorithm obtained by this transformation in two ways. First, the version number is not needed. The *mbal* and *bal* values play the role of a version number. Second, a processor p need not choose a single version of $dblock[q]$ from among the ones it reads from disk. Because *mbal* and *bal* values do not decrease, earlier versions have no effect.

So far, we have ignored processor failures. There is a trivial way to extend the shared-memory algorithm to allow processor failures. A processor recovers by simply reading its *dblock* value from its register and starting a new ballot. A failed process then acts like one in which a processor may start a new ballot at any time. We can show that this generalized version is also correct. However, in the actual disk algorithm, a processor can fail while it is writing. This can leave its disk blocks in a state in which no value has been written to a majority of the disks. Such a state has no counterpart in the shared-memory version. There seems to be no easy way to derive the recovery procedure from a shared-memory algorithm. The proof of the complete Disk Synod algorithm, with failures, is much more complicated than the one for the simple shared-memory version. Trying to write the kind of behavioral proof given above for the simple algorithm leads to the kind of complicated, error-prone reasoning that we have learned to avoid. A sketch of a rigorous assertional proof is given in [5].

3.3 Deriving Classic Paxos from Disk Paxos

In the usual view of a distributed fault-tolerant system, a processor performs actions and maintains its state in local memory, using stable storage to recover from failures. An alternative view is that a processor maintains the state of its stable storage, using local memory only to cache the contents of stable storage. Identifying disks with stable storage, a traditional distributed system is then a network of disks and processors in which each disk belongs to a separate processor; other processors can read a disk only by sending messages to its owner.

Let us now consider how to implement Disk Synod on a network of processors that each has its own disk. To perform phase 1 or 2, a processor p would access a disk d by sending a message containing $dblock[p]$ to disk d's owner q. Processor q could write $dblock[p]$ to $disk[d][p]$, read $disk[d][r]$ for all $r \neq p$, and send the values it read back to p. However, examining the Disk Synod algorithm reveals that there's no need to send back all that data. All p needs are (i) to know if its *mbal* field is larger than any other block's *mbal* field and, if it is, (ii) the *bal* and *inp* fields for the block having the maximum *bal* field. Hence, q need only store on disk three values: the *bal* and *inp* fields for the block with maximum *bal* field, and the maximum *mbal* field of all disk blocks. Of course, q would have those values cached in its memory, so it would actually write to disk only if any of those values are changed.

A processor must also read its own disk blocks to recover from a failure. Suppose we implement Disk Synod by letting p write to its own disk before sending messages to any other processor. This ensures that its own disk has the maximum value of $disk[d][p].mbal$ among all the disks d. Hence, to restart after

a failure, p need only read its block from its own disk. In addition to the *mbal*, *bal*, and *inp* value mentioned above, p would also keep the value of $dblock[p]$ on its disk.

We can now compare this algorithm with classic Paxos's Synod protocol [10]. The *mbal*, *bal*, and *inp* components of $dblock[p]$ are just $lastTried[p]$, $nextBal[p]$, and $prevVote[p]$ of the Synod Protocol. Phase 1 of the Disk Synod algorithm corresponds to sending the *NextBallot* message and receiving the *LastVote* responses in the Synod Protocol. Phase 2 corresponds to sending the *BeginBallot* and receiving the *Voted* replies.[4] The Synod Protocol's *Success* message corresponds to the optimization mentioned above of recording on disk that a value has been committed.

This version of the Disk Synod algorithm differs from the Synod Protocol in two ways. First, the Synod Protocol's *NextBallot* message contains only the *mbal* value; it does not contain *bal* and *inp* values. To obtain the Synod Protocol, we would have to modify the Disk Synod algorithm so that, in phase 1, it writes only the *mbal* field of its disk block and leaves the *bal* and *inp* fields unchanged. The algorithm remains correct, with essentially the same proof, under this modification. However, the modification makes the algorithm harder to implement with real disks.

The second difference between this version of the Disk Synod algorithm and the Synod Protocol is in the restart procedure. A disk contains only the aforementioned *mbal*, *bal*, and *inp* values. It does not contain a separate copy of its owner's *dblock* value. The Synod Protocol can be obtained from the following variant of the Disk Synod algorithm. Let bk be the block $disk[d][p]$ with maximum *bal* field read by processor p in the restart procedure. Processor p can begin phase 1 with *bal* and *inp* values obtained from any disk block bk', written by any processor, such that $bk'.bal \geq bk.bal$. It can be shown that the Disk Synod algorithm remains correct under this modification too.

4 Conclusion

4.1 Implementation Considerations

Implicit in our description of the Disk Synod algorithm are certain assumptions about how reading and writing are implemented when disks are accessed over a network. If operations sent to the disks may be lost, a processor p must receive an acknowledgment from disk d that its write to $disk[d][p]$ succeeded. This may require p to explicitly read its disk block after writing it. If operations may arrive at the disk in a different order than they were sent, p will have to wait for the acknowledgment that its write to disk d succeeded before reading other processors' blocks from d. Moreover, some mechanism is needed to ensure that a write from an earlier ballot does not arrive after a write from a later one,

[4] In the Synod Protocol, a processor q does not bother sending a response if p sends it a disk block with a value of *mbal* smaller than one already on disk. Sending back the maximum *mbal* value is an optimization mentioned in [10].

overwriting the later value with the earlier one. How this is achieved will be system dependent. (It is impossible to implement any fault-tolerant system if writes to disk can linger arbitrarily long in the network and cause later values to be overwritten.)

Recall that, in Disk Paxos, a sequence of instances of the Disk Synod algorithm is used to commit a sequence of commands. In a straightforward implementation of Disk Paxos, processor p would write to its disk blocks the value of $dblock[p]$ for the current instance of Disk Synod, plus the sequence of all commands that have already been committed. The sequence of all commands that have ever been committed is probably too large to fit on a single disk block. However, the complete sequence can be stored on multiple disk blocks. All that must be kept in the same disk block as $dblock[p]$ is a pointer to the head of the queue. For most applications, it is not necessary to remember the entire sequence of commands [10, Section 3.3.2]. In many cases, all the data that must be kept will fit in a single disk block.

In the application for which Disk Paxos was devised (a future Compaq product), the set of processors is not known in advance. Each disk contains a directory listing the processors and the locations of their disk blocks. Before reading a disk, a processor reads the disk's directory. To write a disk's directory, a processor must acquire a lock for that disk by executing a real-time mutual exclusion algorithm based on Fischer's protocol [8]. A processor joins the system by adding itself to the directory on a majority of disks.

4.2 Concluding Remarks

We have presented Disk Paxos, an efficient implementation of the state machine approach in a system in which processors communicate by accessing ordinary (nonprogrammable) disks. In the normal case, the leader commits a command by writing its own block and reading every other processor's block on a majority of the shared disks. This is clearly the minimal number of disk accesses needed.

Disk Paxos was motivated by the recent development of the Storage Area Network (SAN)—an architecture consisting of a network of computers and disks in which all disks can be accessed by each computer. Commodity disks are cheaper than computers, so using redundant disks for fault tolerance is more economical than using redundant computers. Moreover, since disks do not run application-level programs, they are less likely to crash than computers.

Because commodity disks are not programmable, we could not simply substitute disks for processors in the classic Paxos algorithm. Instead we took the ideas of classic Paxos and transplanted them to the SAN environment. What we obtained is almost, but not quite, a generalization of classic Paxos. Indeed, when Disk Paxos is instantiated to a single disk, we obtain what may be called Shared-Memory Paxos. Algorithms for shared memory are usually more succinct and clear than their message passing counterparts. Thus, Disk Paxos can be considered yet another revisiting of classic Paxos that exposes its underlying ideas by removing the message-passing clutter. Perhaps other distributed algorithms can also be made more clear by recasting them in a shared-memory setting.

References

1. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
2. Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
3. Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. In Marios Mavronicolas and Philippas Tsigas, editors, *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125, Saarbruken, Germany, 1997. Springer-Verlag.
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.
5. Eli Gafni and Leslie Lamport. Disk paxos. Technical Report 163, Compaq Systems Research Center, July 2000. Currently available on the World Wide Web at `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-163.html`.
6. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
7. Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.
8. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
9. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
10. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
11. Leslie Lamport. Specifying concurrent systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247, Amsterdam, 1999. IOS Press.
12. Butler W. Lampson. How to build a highly available system using consensus. In Ozalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, 1996. Springer-Verlag.
13. Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 84–92, New York, October 1996. ACM Press.
14. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
15. Marion Dale Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California, Berkeley, May 1982.
16. Chandramohan Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, New York, October 1997. ACM Press.
17. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, September 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.

Appendix

We now give precise specifications of the consensus problem solved by the Disk Synod algorithm and of the algorithm itself. The specifications are written in TLA$^+$, a formal language that combines the temporal logic of actions (TLA) [9], set theory, and first-order logic with notation for making definitions and encapsulating them in modules. These specifications have been debugged with the aid of the TLC model checker [17]. (However, errors may have been introduced by the manual process of translating from TLA$^+$ to LaTeX.) TLA$^+$ is described in [11]; annotated versions of the specifications, with fuller explanations of the TLA$^+$ constructs, appear in [5].

We feel that the algorithm's nonblocking property is sufficiently obvious not to need a rigorous specification and proof, so we consider only consistency. We therefore do not specify any liveness properties, so we make very little use of temporal logic.

The Specification of Consensus

We assume that there are N processors, numbered 1 through N. Each processor p has two registers: an input register $input[p]$ that initially equals some element of the set *Inputs* of possible input values, and an output register $output[p]$ that initially equals a special value *NotAnInput* that is not an element of *Inputs*. Processor p chooses an output value by setting $output[p]$. It can also fail, which it does by setting $input[p]$ to any value in *Inputs* and resetting $output[p]$ to *NotAnInput*. The precise condition to be satisfied is that, if some processor p ever sets $output[p]$ to some value v, then

- v must be a value that is, or at one time was, the value of $input[q]$ for some processor q
- if any processor r (including p itself) later sets $output[r]$ to some value w other than *NotAnInput*, then $w = v$.

We first define a specification *ISpec* that has two additional variables: *allInput*, the set of all inputs chosen so far, and *chosen*, which is set to the first output value chosen. The actual specification *SynodSpec* is obtained from *ISpec* by hiding *allInput* and *chosen*. Hiding in TLA is expressed by the temporal existential quantifier $\exists$. To formally define *SynodSpec* in TLA$^+$, we define *ISpec* in a submodule that is then instantiated. However, the reader not familiar with TLA$^+$ can ignore these details and pretend that *SynodSpec* is simply defined to equal $\exists$ *allInput*, *chosen* : *ISpec*.

The reader unfamiliar with TLA can consider the specification *ISpec* to consist of two parts: the initial predicate *IInit* and the next-state action *INext*, which is a predicate relating the new (primed) state with the old (unprimed) state.

Most of the TLA$^+$ notation used in the definitions should be self-evident, except for the following function constructs: $[x \in S \mapsto g(x)]$ is the function f

with domain S such that $f[x] = g(x)$ for all x in S; $[S \to T]$ is the set of all functions with domain S and range a subset of T; and $[f \text{ EXCEPT } ![x] = e]$ is the function $\widehat{f}$ that is the same as f except that $\widehat{f}[x] = e$. TLA$^+$ allows conjunctions and disjunctions to be written as bulleted lists, with indentation used to eliminate parentheses.

The specification is contained in the following module named *SynodSpec*. The module begins with an EXTENDS statement that imports the *Naturals* module, which defines the set *Nat* of natural numbers and the usual arithmetic operations. The *Naturals* module also defines $i \,..\, j$ to be the set of natural numbers from i through j.

MODULE *SynodSpec*

EXTENDS *Naturals*

CONSTANT N, *Inputs*
ASSUME $(N \in Nat) \land (N > 0)$

$Proc \triangleq 1 \,..\, N$
$NotAnInput \triangleq \text{CHOOSE } c : c \notin Inputs$

VARIABLES *input*, *output*

MODULE *Inner*

VARIABLES *allInput*, *chosen*

$$\begin{aligned} IInit \triangleq\ & \land\ input = [Proc \to Inputs] \\ & \land\ output = [p \in Proc \mapsto NotAnInput] \\ & \land\ chosen = NotAnInput \\ & \land\ allInput = \{input[p] : p \in Proc\} \end{aligned}$$

$$\begin{aligned} Choose(p) \triangleq\ & \land\ output[p] = NotAnInput \\ & \land\ \text{IF } chosen = NotAnInput \\ & \quad \text{THEN } \exists\, ip \in allInput : \land\ chosen' = ip \\ & \qquad\qquad\qquad\qquad\ \land\ output' = [output \text{ EXCEPT } ![p] = ip] \\ & \quad \text{ELSE } \land\ output' = [output \text{ EXCEPT } ![p] = chosen] \\ & \qquad\ \land \text{ UNCHANGED } chosen \\ & \land \text{ UNCHANGED } \langle input, allInput \rangle \end{aligned}$$

$$\begin{aligned} Fail(p) \triangleq\ & \land\ output' = [output \text{ EXCEPT } ![p] = NotAnInput] \\ & \land\ \exists\, ip \in Inputs : \land\ input' = [input \text{ EXCEPT } ![p] = ip] \\ & \qquad\qquad\qquad\ \land\ allInput' = allInput \cup \{ip\} \\ & \land \text{ UNCHANGED } chosen \end{aligned}$$

$INext \triangleq \exists\, p \in Proc : Choose(p) \lor Fail(p)$

$ISpec \triangleq IInit \land \Box[INext]_{\langle input,\, output,\, chosen,\, allInput \rangle}$

$IS(chosen, allInput) \triangleq \text{INSTANCE } Inner$
$SynodSpec \triangleq \exists\, chosen,\, allInput : IS(chosen, allInput)!ISpec$

The Disk Synod Algorithm

The Disk Synod algorithm's specification appears in module *DiskSynod*, which uses an EXTENDS statement to import all the declarations and definitions from the *SynodSpec* module. The specification introduces three new constant parameters: an operator *Ballot* such that $Ballot(p)$ is the set of ballot numbers that processor p can use; a set *Disk* of disks; and a predicate *IsMajority*, which generalizes the notion of a majority. The specification asserts the assumptions that different processors have disjoint sets of ballot numbers, and that, for any subsets S and T of *Disk*, if $IsMajority(S)$ and $IsMajority(T)$ are true, then S and T are not disjoint.

The specification uses the following variables: *input* and *output* are imported from the *SynodSpec* module; *dblock* and *disk* were explained in the informal description of the algorithm; $phase[p]$ is the current phase of processor p, which is set to 0 when p fails and to 3 when p chooses its output; $disksWritten[p]$ is the set of disks that processor p has written during its current phase; and $blocksRead[p][d]$ is the set of values p has read from disk d during its current phase.

Some additional TLA$^+$ notation is introduced in the specification. TLA$^+$ has the following record constructs: $[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$ is the record r with fields $f_1, \ldots, f_n$ such that $r.f_i = v_i$, for each i; and $[f_1 : S_1, \ldots, f_n : S_n]$ is the set of all such records with v_i an element of the set S_i, for each i. The EXCEPT construct has the following extensions: in $[f \text{ EXCEPT } ![x] = e]$, an @ in expression e denotes $f[x]$; the EXCEPT part can have multiple "replacements" separated by commas; and the construct generalizes to functions of functions in the obvious way—for example, $[f \text{ EXCEPT } ![x][y] = e]$. In TLA$^+$, SUBSET S is the set of all subsets of S, and UNION S is the union of all the elements of S.

The algorithm's specification is formula *DiskSynodSpec*, but the reader unfamiliar with TLA can consider the specification to be the initial predicate *Init* and the next-state action *Next*. The module ends by asserting the correctness of the algorithm, expressed in TLA by the statement that the algorithm's specification implies its correctness condition. On first reading, we recommend jumping from the definition of *Init* to the definition of *Next*, and then reading backwards to see what is defined in terms of what.

MODULE *DiskSynod*

EXTENDS *SynodSpec*

CONSTANTS $Ballot(\_)$, *Disk*, $IsMajority(\_)$

ASSUME $\land \forall\, p \in Proc : \land\ Ballot(p) \subseteq \{n \in Nat : n > 0\}$
$\qquad\qquad\qquad\qquad\qquad\qquad \land \forall\, q \in Proc \setminus \{p\} : Ballot(p) \cap Ballot(q) = \{\}$
$\qquad\qquad \land \forall\, S, T \in \text{SUBSET } Disk :$
$\qquad\qquad\qquad IsMajority(S) \land IsMajority(T) \Rightarrow (S \cap T \neq \{\})$

$$DiskBlock \triangleq [\, mbal : (\text{UNION } \{Ballot(p) : p \in Proc\}) \cup \{0\},\ bal : (\text{UNION } \{Ballot(p) : p \in Proc\}) \cup \{0\},\ inp : Inputs \cup \{NotAnInput\}\,]$$

$InitDB \triangleq [mbal \mapsto 0,\ bal \mapsto 0,\ inp \mapsto NotAnInput]$

VARIABLES $disk,\ dblock,\ phase,\ disksWritten,\ blocksRead$

$vars \triangleq \langle input,\ output,\ disk,\ phase,\ dblock,\ disksWritten,\ blocksRead \rangle$

$$\begin{array}{lll} Init \triangleq & \land\ input & \in [Proc \to Inputs] \\ & \land\ output & = [p \in Proc \mapsto NotAnInput] \\ & \land\ disk & = [d \in Disk \mapsto [p \in Proc \mapsto InitDB]] \\ & \land\ phase & = [p \in Proc \mapsto 0] \\ & \land\ dblock & = [p \in Proc \mapsto InitDB] \\ & \land\ output & = [p \in Proc \mapsto NotAnInput] \\ & \land\ disksWritten & = [p \in Proc \mapsto \{\}] \\ & \land\ blocksRead & = [p \in Proc \mapsto [d \in Disk \mapsto \{\}]] \end{array}$$

$hasRead(p, d, q) \triangleq \exists\, br \in blocksRead[p][d] : br.proc = q$

$$\begin{array}{lll} allBlocksRead(p) \triangleq & \text{LET} & allRdBlks \triangleq \text{UNION}\ \{blocksRead[p][d] : d \in Disk\} \\ & \text{IN} & \{br.block : br \in allRdBlks\} \end{array}$$

$InitializePhase(p) \triangleq$

$$\begin{array}{lll} \land\ disksWritten' & = & [disksWritten\ \text{EXCEPT}\ ![p] = \{\}] \\ \land\ blocksRead' & = & [blocksRead\ \text{EXCEPT}\ ![p] = [d \in Disk \mapsto \{\}]] \end{array}$$

$StartBallot(p) \triangleq$

$\land\ phase[p] \in \{1, 2\}$
$\land\ phase' = [phase\ \text{EXCEPT}\ ![p] = 1]$
$\land\ \exists\, b \in Ballot(p) : \land\ b > dblock[p].mbal$
$\qquad\qquad\qquad\qquad\land\ dblock' = [dblock\ \text{EXCEPT}\ ![p].mbal = b]$
$\land\ InitializePhase(p)$
$\land\ \text{UNCHANGED}\ \langle input, output, disk \rangle$

$Phase1or2Write(p, d) \triangleq$

$\land\ phase[p] \in \{1, 2\}$
$\land\ disk' = [disk\ \text{EXCEPT}\ ![d][p] = dblock[p]]$
$\land\ disksWritten' = [disksWritten\ \text{EXCEPT}\ ![p] = @ \cup \{d\}]$
$\land\ \text{UNCHANGED}\ \langle input, output, phase, dblock, blocksRead \rangle$

$Phase1or2Read(p, d, q) \triangleq$

$\land\ d \in disksWritten[p]$
$\land\ \text{IF}\ disk[d][q].mbal < dblock[p].mbal$
$\quad \text{THEN}\ \land\ blocksRead' =$
$\qquad\qquad [blocksRead\ \text{EXCEPT}$
$\qquad\qquad\quad ![p][d] = @ \cup \{[block \mapsto disk[d][q],\ proc \mapsto q]\}]$
$\qquad\quad \land\ \text{UNCHANGED}$
$\qquad\qquad\quad \langle input, output, disk, phase, dblock, disksWritten \rangle$
$\quad \text{ELSE}\ \ StartBallot(p)$

```
EndPhase1or2(p) ≜
  ∧ IsMajority({d ∈ disksWritten[p] : ∀ q ∈ Proc \ {p} : hasRead(p, d, q)})
  ∧ ∨ ∧ phase[p] = 1
      ∧ dblock′ =
          [ dblock EXCEPT
            ![p].bal = dblock[p].mbal,
            ![p].inp = LET blocksSeen ≜ allBlocksRead(p) ∪ {dblock[p]}
                           nonInitBlks ≜
                               {bs ∈ blocksSeen : bs.inp ≠ NotAnInput}
                           maxBlk ≜ CHOOSE b ∈ nonInitBlks :
                                        ∀ c ∈ nonInitBlks : b.bal ≥ c.bal
                       IN  IF nonInitBlks = {} THEN input[p]
                                               ELSE maxBlk.inp ]
      ∧ UNCHANGED output
    ∨ ∧ phase[p] = 2
      ∧ output′ = [output EXCEPT ![p] = dblock[p].inp]
      ∧ UNCHANGED dblock
  ∧ phase′ = [phase EXCEPT ![p] = @ + 1]
  ∧ InitializePhase(p)
  ∧ UNCHANGED ⟨input, disk⟩

Fail(p) ≜ ∧ ∃ ip ∈ Inputs : input′ = [input EXCEPT ![p] = ip]
          ∧ phase′ = [phase EXCEPT ![p] = 0]
          ∧ dblock′ = [dblock EXCEPT ![p] = InitDB]
          ∧ output′ = [output EXCEPT ![p] = NotAnInput]
          ∧ InitializePhase(p)
          ∧ UNCHANGED disk

Phase0Read(p, d) ≜
  ∧ phase[p] = 0
  ∧ blocksRead′ = [blocksRead EXCEPT
                     ![p][d] = @ ∪ {[block ↦ disk[d][p], proc ↦ p]}]
  ∧ UNCHANGED ⟨input, output, disk, phase, dblock, disksWritten⟩

EndPhase0(p) ≜
  ∧ phase[p] = 0
  ∧ IsMajority({d ∈ Disk : hasRead(p, d, p)})
  ∧ ∃ b ∈ Ballot(p) :
      ∧ ∀ r ∈ allBlocksRead(p) : b > r.mbal
      ∧ dblock′ = [dblock EXCEPT
                     ![p] = [ (CHOOSE r ∈ allBlocksRead(p) :
                                   ∀ s ∈ allBlocksRead(p) : r.bal ≥ s.bal)
                              EXCEPT !.mbal = b] ]
  ∧ InitializePhase(p)
  ∧ phase′ = [phase EXCEPT ![p] = 1]
  ∧ UNCHANGED ⟨input, output, disk⟩
```

$$
\begin{array}{l}
Next \triangleq \exists\, p \in Proc : \\
\qquad \lor\ StartBallot(p) \\
\qquad \lor\ \exists\, d \in Disk : \lor\ Phase0Read(p, d) \\
\qquad\qquad\qquad\qquad \lor\ Phase1or2Write(p, d) \\
\qquad\qquad\qquad\qquad \lor\ \exists\, q \in Proc \setminus \{p\} : Phase1or2Read(p, d, q) \\
\qquad \lor\ EndPhase1or2(p) \\
\qquad \lor\ Fail(p) \\
\qquad \lor\ EndPhase0(p)
\end{array}
$$

$$DiskSynodSpec \triangleq Init \land \Box[Next]_{vars}$$

THEOREM $DiskSynodSpec \Rightarrow SynodSpec$

Objects Shared by Byzantine Processes

(Extended Abstract)

Dahlia Malkhi* Michael Merritt** Michael Reiter*** Gadi Taubenfeld†

Abstract. Work to date on algorithms for message-passing systems has explored a wide variety of types of faults, but corresponding work on shared memory systems has usually assumed that only crash faults are possible. In this work, we explore situations in which processes accessing shared objects can fail arbitrarily (Byzantine faults).

1 Introduction

1.1 Motivation

It is commonly believed that message-passing systems are more difficult to program than systems that enable processes to communicate via shared memory. Many experimental and commercial processors provide direct support for shared memory abstractions, and increasing attention is being paid to implementing shared memory systems either in hardware or in software [Bel92, CG89, LH89, TKB92]. Moreover, several middleware systems have been built to implement shared memory abstractions in a message-passing environment. Of primary interest here are those that employ replication to provide fault-tolerant shared memory abstractions, particularly those designed to mask the arbitrary (Byzantine) failure of processes implementing these abstractions (e.g., see [PG89, SE+92, Rei96, KMM98, CL99, MR00]). These middleware systems generally guarantee that shared objects themselves do not "fail", and hence, that their integrity, safety properties, and access interfaces and restrictions, are preserved. Nevertheless, since legitimate clients accessing these objects might fail arbitrarily, they could corrupt the states of these objects in any way allowed by the object interfaces.

The question we address in this paper is: What power do shared memory objects have in such environments, in achieving any form of coordination among distributed processes that access these objects? This question is daunting, as Byzantine faulty processes can configure objects in any way allowed by the object

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

** AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971. mischu@research.att.com

*** Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. reiter@research.bell-labs.com

† The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs. gadi@cs.openu.ac.il

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 345–359, 2000.

interfaces. Thus, seemingly even very strong shared objects such as consensus objects (which are universal for crash failures) might not be very useful in such a Byzantine environment, as faulty processes erroneously set their decision values. Surprisingly, although work to date on algorithms for message-passing systems has explored a wide variety of types of faults, corresponding work on shared memory systems has usually assumed that only crash faults are possible. Hence, our work is the first study of the power of objects shared by Byzantine processes.

1.2 Summary of results

We generalize the crash-fault model of shared memory to accommodate Byzantine faults. We show how a variety of techniques can be used to cooperate reliably in the presence of Byzantine faults, including bounds on the numbers of faulty processes, redundancy, *access control lists* that constrain faulty processes from accessing specific objects, and *persistent* objects (such as sticky bits [Plo89]) which cannot be overwritten. (We call objects that are not persistent, such as read/write registers, *ephemeral*.) We define a notion of shared object that is appropriate for this fault model, in which waiting between concurrent operations is permitted. We explore the power of some specific shared objects in this model, proving both universality and impossibility results, and finally identify some non-trivial problems that can be solved in the presence of Byzantine faults even when using only ephemeral objects.

The notions of consensus objects and sticky bits (a persistent, readable consensus object) in the Byzantine model, are formally defined in section 2. The results are:

1. **Universality result:** Our main result shows that sticky bits can be used to construct any other object (i.e., they are universal), assuming that the number of (Byzantine) faults is bounded by $(\sqrt{n}-1)/2$, where n is the total number of processes.

   To prove this result, a universal construction is presented that works as follows: First, sticky bits are used to construct a *strong* consensus object, i.e., a consensus object whose decision is a value proposed by some *correct* process. Equipped with strong consensus objects, we proceed to emulate any object. Our emulation borrows closely from Herlihy's universal construction for crash faults [Her91], but differs in significant ways due to the need to cope with Byzantine failures.
2. **Bounds on faults:** We observe that strong consensus objects, used to prove the universality result, cannot be constructed when the possible number of faults is $t \geq n/3$. We observe that there exists a simple bounded-space universal object assuming $t < n/3$, and a trivial unbounded-space universal object assuming any number of $t \leq n$ faults. We prove that when a majority of the processes may be faulty, even *weak* consensus (i.e., a consensus object whose decision is a value proposed by some *correct* or *faulty* process) cannot be solved using any of the familiar non-sticky objects.

3. **Constructions using ephemeral objects:** While the universality result involves sticky bits, the impossibility result shows that consensus cannot be implemented using known objects that are not persistent. This raises the question of what can be done with such ephemeral objects. We show how various objects, such as k-set consensus and k-pairwise consensus, can be implemented in a Byzantine environment using only atomic registers. Then we show that familiar objects such as test&set, swap, compare&swap, and read-modify-write, can be used to implement election objects for any number of processes and under any number of Byzantine faults.

1.3 Related work

The power of various shared objects has been studied extensively in shared memory environments where processes may fail benignly, and where every operation is wait-free: the operation is guaranteed to return within a finite number of steps. Objects that can be used (together with atomic registers) to give a wait-free implementation of any other objects are called *universal objects*. Previous work on wait-free (and non-blocking) shared objects provided methods (called *universal constructions*) to transform sequential implementations of arbitrary shared objects into wait-free concurrent implementations, assuming the existence of a universal object [Her91, Plo89, JT92]. In particular, Plotkin showed that sticky bits are universal [Plo89], and independently, Herlihy proved that consensus objects are universal [Her91]. Herlihy also showed that shared objects can be classified according to their consensus number: that is, the maximum number of processes that can reach consensus using the object [Her91]. Attie investigates the power of shared objects accessed by Byzantine processes for achieving wait-free Byzantine agreement. He proves that strong agreement is impossible to achieve using *resettable* objects, i.e., objects that can be reset back to their initial setting, and constructs weak agreement using sticky bits [Att00].

Assume that at some point in a computation a shared register is set to some unexpected value. There are two complementary ways to explain how this may happen. One is to assume that the register's value was set by a Byzantine process (as may happen in the model of this paper). The other way is to assume that the processes are correct but the register itself is faulty. The subject of memory faults (as opposed to process faults) has been investigated recently in several papers [AGMT95, JCT98]. These papers assume any number of process crash failures, but bound the number of faulty objects, whereas we bound the number of (Byzantine) faulty processes, but each might sabotage all the objects to which it has access.

As described in the introduction, our focus on a shared memory Byzantine environment is driven by previous work on message-passing systems that emulate shared memory abstractions tolerant of Byzantine failures (e.g., [PG89, SE+92, Rei96, KMM98, CL99, MR00]). Though these systems guarantee the correctness of the emulated shared objects themselves, the question is what power do these objects provide to the correct processes that use them, in the face of corrupt processes accessing them.

2 Model and definitions

Our model of computation consists of an asynchronous collection of n processes, denoted $p_1, \ldots, p_n$, that communicate via shared objects. In any run any process may be either correct or faulty. Correct processes are constrained to obey their specifications, while faulty processes can deviate arbitrarily from their specifications (Byzantine failures) limited only by the assumptions stated below. We denote by t the maximum number of faulty processes.

2.1 Shared objects with access control lists

Each shared object presents a set of operations. e.g., $x.\mathsf{op}$ denotes operation op on object x. For each such operation on x, there is an associated access control list (ACL) that names the processes allowed to *invoke* that operation. Each operation execution begins with an invocation by a process in the operation's ACL, and remains pending until a response is received by the invoking process. The ACLs for two different operations on the same object can differ, as can the ACLs for the same operation on two different objects. The ACLs for an object do not change. For any operation $x.\mathsf{op}$, we say that x is k-op if the ACL for $x.\mathsf{op}$ lists k processes. We assume that a process not on the ACL for $x.\mathsf{op}$ cannot invoke $x.\mathsf{op}$, regardless of whether the process is correct or Byzantine (faulty). That is, a (correct or faulty) process cannot access an object in any way except via the operations for which it appears on the associated ACLs.

We note that the systems that motivated our study typically employ replication to fault-tolerantly emulate shared memory abstractions. Therefore, ACLs can be implemented, e.g., by storing a copy of the ACL with each replica and filtering out disallowed operations before applying them to the replica. In this way, only operations allowed by the ACLs will be applied at correct replicas.

2.2 Fault tolerance and termination conditions

In wait-free fault models, no bound is assumed on the number of potentially faulty processes. (Hence, no process may safely wait upon an action by another.) Any operation by a process p on a shared object must terminate, regardless of the concurrent actions of other processes. This model supports a natural and powerful notion of abstraction, which allows complex implementations to be viewed as atomic [HW90]. We extend this model in two ways: first, we make the more pessimistic assumption that process faults are Byzantine, and second, we make the more optimistic assumption that the number of faults is bounded by t, where t is less than the total number of processes, n. With the numbers of failures bounded away from n, it becomes possible (and indeed necessary) for processes to coordinate with each other, using redundancy to overcome the Byzantine failures of their peers. This means that processes may need to wait for each other within individual operation implementations.

An example that may provide some intuition is a sticky bit object emulated by an ensemble of data servers, such that the value written to it must reflect a

value written by some *correct* process. A distributed emulation may implement this object by having servers set the object's value only when $t + 1$ different processes write to it the same value. Of course, this object will be useful only when any value written to the object is indeed written by at least $t+1$ processes, and so an application must guarantee that $t+1$ correct processes write identical values. Below, we will see examples of such constructions.

Such an implementation is not wait-free, and raises the question of appropriate termination conditions for object invocations in a Byzantine environment. To address such concerns, we introduce two object properties, t-threshold and t-resilience. The first captures termination conditions appropriate for an object on which each client should invoke a single operation, and which function correctly once enough correct processes access them. The second is appropriate when processes perform multiple operations on an object, each of which may require support from a collection of correct processes.

t-threshold: For any operation $x.\mathsf{op}$, we say that $x.\mathsf{op}$ is t-threshold if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run ρ in which $n - t$ correct processes invoke $x.\mathsf{op}$.

t-resilience: For any operation $x.\mathsf{op}$, we say that $x.\mathsf{op}$ is t-resilient if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run ρ in which each of at least $n - t$ correct processes infinitely often has a pending invocation of $x.\mathsf{op}$.

An object is t-threshold (t-resilient) if all the operations it supports are t-threshold (t-resilient). Notice that t-threshold implies t-resilience, but not vice versa.

2.3 Object definitions

Below we specify some of the objects used in this paper.

Atomic registers: An atomic register x is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{write}(v)$ where $v \neq \perp$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{write}()$ returns the value written in the last preceding $x.\mathsf{write}()$ operation. Throughout this paper we employ wait-free atomic registers, i.e., $x.\mathsf{read}$ or $x.\mathsf{write}()$ operations by correct processes eventually return (regardless of the behavior of other processes).

Sticky bits: A sticky bit x is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{write}(v)$ where $v \in \{0, 1\}$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{write}()$ returns the value written in the first $x.\mathsf{write}()$ operation. We will be concerned with wait-free sticky bits.

Weak consensus objects: A weak (binary) consensus object x is an object with one operation: $x.\mathsf{propose}(v)$, where $v \in \{0, 1\}$, satisfying: (1) The $x.\mathsf{propose}()$ operation returns the same value, called the consensus value, to every process that invokes it. (2) If the consensus value is v, then some process invoked $x.\mathsf{propose}(v)$.

Strong consensus objects: A strong (binary) consensus object x strengthens the second condition above to read: (2) If the consensus value is v, then some *correct* process invoked $x.\mathsf{propose}(v)$.

Observe that one sticky bit does not trivially implement a strong consensus object, where each process first writes this bit and then reads it and decides on the value returned. The first process to write the bit might be a faulty one, violating the requirement that the consensus value must be proposed by some *correct* process. (In Lemmas 2 and 3 we describe more complex implementations of strong consensus from sticky bits.) Indeed, strong consensus objects do not have sequential runs: the additional condition, using redundancy to mask failures, requires at least $t + 1$ processes to invoke x.propose() before any correct process returns from this operation. (In addition, Theorem 4 in Section 3.2 shows that t-resilient strong consensus objects are ill-defined when $t \geq n/3$.)

Throughout the paper, unless otherwise stated, by a consensus object we mean a *strong* consensus object. Also, atomic registers and sticky bits are always assumed to be *wait-free*.

3 A universal construction

This section contains the main result of this paper, the construction of a universal t-resilient object from wait-free sticky bits. That is, we show that sticky bits are universal when the number of faults is small enough.

We assume any fault-tolerant object, o, is specified by two relations:

$$apply \subset \text{INVOKE} \times \text{STATE} \times \text{STATE},$$

$$\text{and } reply \subset \text{INVOKE} \times \text{STATE} \times \text{RESPONSE},$$

where INVOKE is the object's domain of invocations, STATE is its domain of states (with a designated set of start states), and RESPONSE is its domain of responses. The *apply* relation denotes a nondeterministic state change based on the specific pending invocation and the current state (invocations do not block: we require a target state for every invocation and current state), and the *reply* relation nondeterministically determines the calculated response, based on the pending invocation and the updated state.[5] It is necessary to define two relations because in fault-tolerant objects (such as strong consensus), the response may depend on later invocations. The *apply* relation allows the state to be updated once the invocation occurs, without yet determining the response. The *reply* relation may only allow a response to be determined when other pending invocations update the state.

For example, a t-threshold strong consensus object can be specified as follows: STATE is the set of integer pairs, (x, y), $0 \leq x, y \leq t$, or the singletons 0 and 1, with $(0, 0)$ as the single start state. For all integers x, y and u, v in $\{0, 1\}$ (constrained as shown), the *apply* relation is, $\{(\text{PROPOSE}(0), (x < t, y), (x +$

[5] This formulation generalizes Herlihy's specification of wait-free objects by a single relation $apply \subset \text{INVOKE} \times \text{STATE} \times \text{STATE} \times \text{RESPONSE}$, restricted (by the wait-free condition) to have at least one target state and response defined for any pair INVOKE × STATE [Her91]. This formulation is insufficient to define fault-tolerant objects such as strong consensus.

$1, y)\}) \cup \{(\text{PROPOSE}(1), (x, y < t), (x, y + 1))\} \cup \{(\text{PROPOSE}(0), (t, y), 0)\} \cup \{(\text{PROPOSE}(1), (x, t), 1\}) \cup \{(\text{PROPOSE}(u), v \in \{0, 1\}, v)\}$, and the *reply* relation is $\{(\text{PROPOSE}(u), v \in \{0, 1\}, \text{RETURN}(v))\}$. Hence, each invocation of a propose operation enables *apply* to increment the appropriate counter in the state. Concurrent invocations introduce race conditions (as to which application of *apply* occurs first. Once $t+1$ applications of the same value occur, the state is committed to that binary value, and the responses of pending invocations are enabled.

For the purposes of the universal construction below, we resolve any nondeterminism, and assume that the first relation is a function from INVOKE $\times$ STATE to STATE, and that the second relation is a partial function from INVOKE $\times$ STATE to RESPONSE. Given these restrictions, we may assume, without loss of generality, that the object's domain of states is the set of strings of invocations, and that the function from INVOKE $\times$ STATE to STATE, simply appends the pending invocation to the current state.

Theorem 1. *Any t-resilient object can be implemented using:*

1. $(t + 1)$-write(), n-read *sticky bits and* 1-write(), n-read *sticky bits, provided that* $n \geq (t + 1)(2t + 1)$*; or*
2. $(2t + 1)$-write(), $(2t + 1)$-read *sticky bits and* 1-write(), n-read *sticky bits, provided that* $n \geq (2t + 1)^2$.

Figure 1 describes a universal implementation. In the lemmas, we provide two constructions of (strong) binary consensus objects using sticky bits, which differ in the access restrictions.

Lemma 2. *If $n \geq (t + 1)(2t + 1)$, then an n-propose() t-threshold consensus object can be implemented using $(t + 1)$-write(), n-read sticky bits.*

Proof: Let o be the consensus object that is being implemented. Let $m = \lfloor \frac{n}{t+1} \rfloor$. Partition the n processes into blocks $B_1, \ldots, B_m$, each of size at least $t + 1$, and let $x_1, \ldots, x_m$ be sticky bits with the property that the ACL for x_i.write() is B_i (or a $(t + 1)$-subset thereof) and the ACL for x_i.read is $\{p_1, \ldots, p_n\}$. For a correct process $p \in B_i$ to emulate o.propose(v), it executes x_i.write(v) (or *skip* if p is not in the ACL for x_i) and, once that completes, repeatedly executes x_j.read for all $1 \leq j \leq m$ until none return $\perp$. p chooses the return value from o.propose(v) to be the value that is returned from the read operations on a majority of the x_j's.[6] All correct processes obtain the same return value from their o.propose() emulations because the x_i's are sticky. If no correct process emulates o.propose(v), then since $m \geq 2t + 1$, v will not be returned from the reads on a majority of the x_j's and thus will not be the consensus value. Because each correct process reads x_j, $1 \leq j \leq m$, until none return $\perp$, termination is guaranteed provided that each sticky bit is set. Since each x_j has $t+1$ processes proposing to it, it follows that o.propose() is guaranteed to return when at least $n - t$ perform propose() operations. □

[6] In case m is even and the number of 1's equals the number of 0's, the majority value is defined to be 1.

Lemma 3. *If $n \geq (2t+1)^2$, then an n-propose() t-threshold consensus object can be implemented using $(2t+1)$-write(), $(2t+1)$-read sticky bits and* 1-write(), *n-read sticky bits.*

Proof: Let o be the consensus object that is being implemented. Let $r_1, \ldots, r_n$ be 1-write(), n-read sticky bits such that the ACL for r_i.write() is $\{p_i\}$. Let $m = \lfloor \frac{n}{2t+1} \rfloor$. Partition the n processes into blocks $B_1, \ldots, B_m$, each of size at least $2t+1$, and let $x_1, \ldots, x_m$ be sticky bits with the property that the ACLs for x_i.write() and x_i.read are both B_i (or a $(2t+1)$-subset thereof). For a correct process $p_j \in B_i$ to emulate o.propose(v), it executes x_i.write(v) (or *skip* if p is not in the ACL for x_i) and, once that completes, it executes $r_j \leftarrow x_i$.read. p_j then repeatedly reads the (single-writer) bits of all processes until for each B_k, it observes the same value V_k in the bits of $t+1$ processes in B_k; note that V_k must be the value returned by x_k.read (to a process allowed to execute x_k.read). The value that occurs as $t+1$ such V_k's is selected as the return value from o.propose(v). Because x_i is sticky and B_i contains at most t faulty processes, V_i is unique; thus, all correct processes obtain the same return value from their o.propose() emulations. If no correct process emulates o.propose(v), then since $m \geq 2t+1$, v cannot occur in the majority of the V_j's. □

3.1 Proof of Theorem 1

For simplicity, we initially describe a universal construction of objects for which the domain of invocations is finite. Subsequently, we explain how to modify the construction to implement objects with (countably) infinite invocation domains.

The construction conceptually mimics Herlihy's construction showing that consensus is universal for wait-free objects in the fail-stop model [Her91]. Due to the possibility of arbitrarily faulty processes in our system model, however, construction below differs in significant ways.

The construction labors to ensure that operations by correct processes eventually complete, and that each operation by Byzantine processes either has no impact, or appears as (the same) valid operation to the correct processes. There are two principal data structures:

1. For each process p_i there is an unbounded array Announce[i][1...], each element of which is a "cell", where a cell is an array of $\lceil \log(|\text{INVOKE}|) \rceil$ sticky bits. The Announce[i][j] cell describes the j-th invocation (operation name and arguments) by p_i on o. Accordingly, the ACL for the write() operation of each sticky bit in each cell of Announce[i] names p_i.
2. The object itself is represented as an unbounded array Sequence[1...] of process-id's, where each Sequence[k] is a $\lceil \log(n) \rceil$ string of t-threshold, strong binary consensus objects. We refer to the value represented by the string of bits in Sequence[k] simply as Sequence[k]. Intuitively, if Sequence[k] $= i$ and Sequence[1], ..., Sequence[$k-1$] contains the value i in exactly $j-1$ positions, then the k-th invocation on o is described by Announce[i][j]. In this case, we say that Announce[i][j] has been *threaded.*

```
type:   ID: array of ⌈log(n)⌉ strong consensus objects
        CELL: array of ⌈log(|INVOKE|)⌉ sticky bits
global variables:
        Announce[1..n][1...], array of CELL
           for all i, 1 ≤ i ≤ n, and j, elements of Announce[i][j] are writable by p_i
        Sequence[1...], infinite array of IDs, each accessible by all processes
variables private to process p_i:
        MyNextAnnounce, index of next vacant cell in Announce[i], initially 1
        NextAnnounce[1..n], for each 1 ≤ j ≤ n, index in Announce[j][]
                                  of next operation of p_j to be read by p_i, initially 1
        CurrentState ∈ STATE, p_i's view of the state of o, initially the initial state of o.
        NextSeq, next position to be threaded in Sequence[] as seen by p_i, initially 1
        NameSuffix, ⌈log(n)⌉ bit string
o.op:
(1)     write, bit by bit, the invocation,
                                  o.op.invoke of o.op into Announce[i][MyNextAnnounce]
(2)     MyNextAnnounce++
                                ; Apply operations until o.op is applied and p_i can return.
                                ; Each while loop iteration applies exactly one operation.
(3)     while ((NextAnnounce[i] < MyNextAnnounce) or
              ((NextAnnounce[i] ≥ MyNextAnnounce)
              and (reply(o.op.invoke, CurrentState) is not defined))) do
(4)       ℓ ← NextSeq (mod n)                          ; Select preferred process to help.
(5)       NameSuffix ← emptystring
(6)       for k = 0 to ⌈log(n)⌉ do ; Loop applies the operation one bit per iteration.
                                                  Search for a valid process index to propose
(7)         while ((Announce[ℓ + 1][NextAnnounce[ℓ + 1]] is invalid)
                    or (NameSuffix is not a suffix of the bit encoding of ℓ + 1)) do
(8)           ℓ ← ℓ + 1 (mod n) od
                                          ; Propose the k'th bit (right to left) of ℓ + 1
(9)         prepend(NameSuffix, Sequence[NextSeq][k].propose((ℓ + 1)&(2^k))
(10)      od     ; A new cell has been threaded by NameSuffix in Sequence[NextSeq]
(11)      CurrentState ←
              apply(Announce[NameSuffix][NextAnnounce[NameSuffix]], CurrentState)
(12)      NextAnnounce[NameSuffix] + +
(13)      NextSeq++
(14)    od
(15)    return(reply(o.op.invoke, CurrentState))
```

Figure 1: Universal implementation of o.op at p_i.

The universal construction of object o is described in Figure 1 as the code process p_i executes to implement an operation o.op, with invocation o.op.*invoke*. In outline, the emulation works as follows: process p_i first announces its next invocation, and then threads unthreaded, announced invocations onto the end of Sequence. It continues until it sees that its own operation has been threaded, and that enough additional invocations (if any) have been threaded, that it can compute a response and return. To assure that each announced invocation is eventually threaded, the correct processes first try to thread any announced,

unthreaded cell of process $p_{\ell+1}$ into entry Sequence[k], where $\ell = k \pmod{n}$. (Once process $p_{\ell+1}$ announces an operation, at most n other operations can be threaded before $p_{\ell+1}$'s.)

In more detail, process p_i keeps track of the first index of Announce[i] that is vacant in a variable denoted MyNextAnnounce, and first (line 1) writes the invocation, bit by bit, into Announce[i][MyNextAnnounce], and (line 2) increments MyNextAnnounce. To keep track of which cells it has seen threaded (including its own), p_i keeps n counters in an array NextAnnounce[1..n], where each NextAnnounce[j] is one plus the number of times i has read cells of j in Sequence, and hence the index of Announce[j] where i looks to find the next operation announced by j. Hence, having incremented MyNextAnnounce, NextAnnounce[i] = MyNextAnnounce $-$ 1 until the current operation of p_i has been threaded.

This inequality is thus one disjunct (line 3) in the loop (lines 4-10) in which p_i **threads cells. Once p_i's cell is threaded, (and NextAnnounce[i] = MyNextAnnounce),** the next conjunct (again line 3) keeps p_i threading cells until a response to the threaded operation can be computed. (At which time it exits the loop and returns the associated value (line 15).) Notice that in some cases, this may require any finite number of additional operations to be threaded after o.op, but by the t-resilient condition, as long as operations of correct processes are eventually threaded, eventually o.op can return. For example, if o.op is the propose() operation of a strong consensus object, then it can return once at least $t+1$ propose() invocations with identical values occur. Process p_i keeps an index NextSeq which points to the next entry in Sequence[1, ...] whose cells it has not yet accessed.

To thread cells, process p_i proposes (line 9) the binary encoding of a process id, $\ell + 1$, bit by bit, to Sequence[NextSeq]. In choosing $p_{\ell+1}$, process p_i first checks (first disjunct, line 7) that Announce[$\ell+1$][NextAnnounce[$\ell+1$]] contains a valid encoding of an operation invocation. (And, as discussed above, p_i gives preference (line 4) to a different process for each cell in Sequence.)

Starting (line 5) with the *emptystring*, p_i accumulates (line 9) the bit-by-bit encoding of the id being recorded in Sequence[NextSeq] into a local variable, NameSuffix. If a bit being proposed by p_i is not the result returned (second disjunct, line 7), then p_i searches (line 8) for another process to help, whose id matches the bits accumulated in NameSuffix. (The properties of strong consensus assure that such a process exists.)

Once process p_i accumulates all the bits of the threaded cell into NameSuffix (the termination condition (line 6) of the **for** loop (lines 7-10)), it can update (line 11) its view of the object's state with this invocation, and increment its records of (line 12) process NameSuffix's successfully threaded cells and (line 13) the next unread cell in Sequence. Having successfully threaded a cell, p_i returns to the top of the **while** loop (line 3).

The sequencing and correct semantics of each operation follow trivially from the sequential ordering of invocations in Sequence and the application of the *apply* and *reply* functions. The proper termination of all correct operations follow as argued above from the t-threshold property of the embedded consensus objects and from the t-resilience of the object.

The construction and this argument address objects with finite domains of invocation. We next briefly outline the modifications necessary to accommodate objects with (countably) infinite domains of invocation. The quandary here is that the representations of invocations using sticky bits are unbounded. Suppose we naively change the type $CELL$ to (unbounded) sequence of sticky bits.

When process p_i attempts to read (line 7) an invocation in Announce$[\ell + 1]$[NextAnnounce$[\ell + 1]$], a faulty process might cause p_i to read forever, by itself writing forever, in such a way that each finite prefix is a valid but incomplete encoding of an invocation. (For any encoding, such a sequence exists by König's lemma.) This problem can be avoided by interleaving reads of the bits of each entry in Announce$[\ell+1]$[NextAnnounce$[1..n]$], starting as before with the next bit of NextAnnounce$[\ell + 1]$, until one of the accumulated strings validly encodes an invocation. Details of the bookkeeping required, and the argument that correct invocations are eventually threaded, are left to the reader. (Though note that the number of invocations that may be threaded before a correct process's announcement is now dependent on the relative lengths of different encodings.) □

3.2 Resilience and impossibility

The proof of Theorem 1 presents a universal construction of t-resilient objects, where $t \leq (\sqrt{n}-1)/2$. Naturally, one would like to know whether there are more fault-tolerant universal constructions, and in the limit, whether wait-free universal constructions exist. Focusing on improving the the bound $t \leq (\sqrt{n}-1)/2$ in Theorem 1, that is, finding a universal construction or impossibility proofs $t > (\sqrt{n}-1)/2$, we note that the construction in Figure 1 builds modularly on t-resilient strong consensus. The $t \leq (\sqrt{n}-1)/2$ bound of Theorem 1 follows from the constructions of strong consensus from sticky bits, in Lemmas 2 and 3. Constructions of strong consensus from sticky bits for larger values of t would imply a more resilient universality result. The theorem below demonstrates that such a search is bounded by $t < n/3$.

Theorem 4. *For $t \geq n/3$, there is no t-resilient n-*`propose()` *(strong) consensus object.*

Proof. Assume to the contrary that there exists such an object. Let P_0 and P_1 be two sets of processes such that for each P_i (where $i \in \{0,1\}$) the size of P_i is $\lceil n/3 \rceil$ and all processes in P_i propose the value i (i.e., have input i). Run these two groups as if all the $2\lceil n/3 \rceil$ processes are correct until they all commit to a consensus value. Without loss of generality, let this value be 0. Next, we let all the remaining processes propose 1 and run until all commit to 0. We can now assume that all the processes in P_0 are faulty and reach a contradiction. □

We point out that it is easy to define objects that are universal for any number of faults. An example is the *append-queue* object, which supports two operations. The first appends a value onto the queue, and the second reads the entire contents of the queue. By directly appending invocations onto the queue, the entire history of the object can be read.

4 Ephemeral objects

In this section, we explore the power of ephemeral objects. We prove an impossibility result for a class of *erasable* objects, and give several fault-tolerant constructions.

Erasable objects: An erasable object is an object in which each pair of operations op_1 and op_2, when invoked by different processes, either (1) commute (such as a read and any other operation) or (2) for every pair of states s_1 and s_2, have invocations $invoke_1$ and $invoke_2$ such that $apply(invoke_1, s_1) = apply(invoke_2, s_2)$. **Such familiar objects as registers, test&set, swap, read-modify-write are erasable.** (This definition generalizes the notion of commutative and overwriting operations [Her91].)

Theorem 5. *For any $t > n/2$, there is no implementation of a t-resilient n-*`propose()` *weak consensus object using any set of erasable objects.*

Proof. Assume to the contrary the such an implementation, called A, is possible. We divide the n processes into three disjoint groups: P_0 and P_1 each of size at least $\lfloor (n-1)/2 \rfloor$, and a singleton which includes process p. Consider the following finite runs of algorithm A:

1. ρ_0 is a run in which only processes in P_0 participate with input 0 and halt once they have decided. They must all decide on 0. Let O_0 be the (finite) set of objects that were accessed in this run. and let s_0^i be the state of object o_i at the end of this run.
2. ρ_1 is a run in which only processes in P_1 participate with input 1 and halt once they have decided. They must all decide on 1. Let O_1 be the (finite) set of objects that were accessed in this run, and let s_1^i be the state of object o_i at the end of this run.
3. ρ_0' is a run in which processes from P_0 are correct and start with input 0, and processes from P_1 are faulty and start with input 1. It is constructed as follows. First the process from P_0 run exactly as in ρ_0 until they all decide on 0. Then, the processes from P_1 set all the shared objects in $(O_1 - O_0)$ to the values that these objects have at (the end of) ρ_1, and set the values of the objects in $(O_1 \cap O_0)$ to hide the order of previous accesses That is, for objects in which all operations accessible by P_0 and P_1 commute, P_1 runs the same operations as in run ρ_0. For each remaining object o_i, P_0 invokes an operation $invoke_0$ such that P_1 has access to an operation $invoke_1$ where $apply(invoke_1, s_0^i) = apply(invoke_1, s_2^i)$.
4. ρ_1' is a run in which processes from P_1 are correct and start with input 1, and processes from P_0 are faulty and start with input 0. It is constructed symmetrically to ρ_2: First the process from P_1 run exactly as in ρ_1 until they all decide on 1. Then, as above the processes from P_0 set all the shared objects in $(O_0 - O_1)$ to the values that these objects have at (the end of) ρ_0. For objects in which all operations accessible by P_0 and P_1 commute, P_0 runs the same operations as in run ρ_0. For each remaining object o_i, P_0 invokes the operation $invoke_1$ defined in ρ_0'.

By construction, every object is in the same state after ρ'_0 and ρ'_1. But if we activate process p alone at the end of ρ'_0, it cannot yet decide, because it would decide the same value if we activate process p alone at the end of ρ'_1. So p must wait for help from the correct processes (which the t-resilience condition allows it to do) to disambiguate these identical states.

Having allowed p to take some (ineffectual) steps, we can repeat the construction again, scheduling P_0 and P_1 to take additional steps in each run, but bringing the two runs again to identical states. By repeating this indefinitely, we create two infinite runs, in each of which the correct processes, including p, take an infinite number of steps, but in which p never decides, a contradiction. □

4.1 Atomic registers

Next we provide some examples of implementations using (ephemeral) atomic registers. The first such object is t-resilient k-set consensus [Cha93].

k-set consensus objects: A k-set consensus object x is an object with one operation: x.`propose`(v) where v is some number. The x.`propose`() operation returns a value such that (1) each value returned is proposed by some process, and (2) the set of values returned is of size at most k.

Theorem 6. *For any $t < n/3$, if $t < k$ then there is an implementation of a t-resilient n-*`propose`*() k-set consensus object using atomic registers.*

Proof. Processes p_1 through p_{t+1} announce their input value by writing it into a register $announce[i]$, whose value is initially $\perp$. Each process repeatedly reads the $announce[1..t+1]$ registers, and echoes the first non-$\perp$ value it sees in any $announce[j]$ entry by copying it into a 1-writer register $echo[i,j]$. Interleaved with this process, p_i also reads all the $echo[1..n, 1..t+1]$ registers, and returns the value it first finds echoed the super-majority of $2n/3+1$ times in some column $echo[1..n, k]$. In subsequent operations, it returns the same value, but first examines $announce[1..t+1]$ array and writes any new values to $echo[i, 1..t+1]$.

Using this construction, no process can have two values for which a super-majority of echos are ever read. Moreover, any correct process among p_1 through p_{t+1} will eventually have its value echoed by a super-majority. Hence, every operation by a correct process will eventually return one of at most $t+1$ different values. □

The implementation above of k-set-consensus constructs a t-resilient object. The next result shows that registers can be used to implement the stronger t-threshold condition. (The proof is omitted from this extended abstract.)

k-pairwise set-consensus objects: A k-pairwise set-consensus object x is an object with one operation: x.`propose`(v) where v is some number. The x.`propose`() operation returns a set of at most k values such that (1) each value in the set returned is proposed by some process, and (2) the intersection of any two sets returned is non-empty.

Theorem 7. *For any $t < n/3$, there is an implementation of a t-threshold n-*`propose`*() $(2t+1)$-pairwise set-consensus object using atomic registers.*

4.2 Fault-tolerant constructions using objects other than registers

Even in the presence of only one crash failure, it is not possible to implement election objects [TM96, MW87] or consensus objects [LA87, FLP85] using only atomic registers. Next we show that many other familiar objects, such as 2-process weak consensus, test&set, swap, compare&swap, and read-modify-write, can be used to implement election objects for any number of processes and under any number of Byzantine faults.

Election objects: **An election object x is an object with one operation: x.`elect`()** The x.`elect`() operation returns a value, either 0 or 1, such that at most one correct process returns 1, and if only correct processes participate then exactly one process gets 1 (that process is called the *leader*). Notice that it is not required for all the processes to "know" the identity of the leader. We have the following result. (Proof omitted from this extended abstract.)

Theorem 8. *There is an implementation of (1) n-threshold* n-`elect`() *election from two-process versions of weak consensus, test&set, swap, compare&swap, or read-modify-write, and (2) 2-threshold* 2-`propose()` *weak consensus from* 2-`elect`() *election.*

5 Discussion

The main positive result in this paper shows that there is a t-resilient universal construction out of wait-free sticky bits, in a Byzantine shared memory environment, when the number of failures t is limited. This leaves open the specific questions of whether it is possible to weaken the wait-freedom assumption (assuming sticky bits which are t-threshold or t-resilient) and/or to implement a t-threshold object (instead of a t-resilient one).

We have also presented several impossibility and positive results for implementing fault-tolerant objects. There are further natural questions concerning the power of objects in this environment, such as: Is the resilience bound in our universality construction tight for sticky bits? What is the resilience bound for universality using other types of objects? What type of objects can be implemented by others? The few observations regarding these questions in Section 3.2 and 4 only begin to explore these questions.

References

[Att00] P.C. Attie Wait-free Byzantine Agreement. Technical Report NU-CCS-00-02, College of Computer Science, Northeastern University, May 2000.

[AGMT95] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. *Journal of the ACM*, 42(6):1231-1274, November 1995.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[CL99] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation - OSDI'99*, February, 1999, New Orleans, LA.

[Cha93] S. Chaudhuri. More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, July 1993.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 11(1):124–149, January 1991.

[JCT98] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451-500, May 1998.

[JT92] P. Jayanti, and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. *Proc. of the 6th Int. Workshop on Distributed Algorithms: LNCS, 647*, pages 69–84. Springer Verlag, Nov. 1992.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463–492, July 1990.

[KMM98] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Hawaii Int. Conf. on System Sciences*, pages 317–326, January 1998.

[LA87] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research, JAI Press*, 4:163–183, 1987.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Programming Languages and Systems*, 7(4):321–359, 1989.

[MR00] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

[MW87] S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Info. Processing Letters*, 26:141–151, 1987.

[PG89] F. M. Pittelli and H. Garcia-Molina. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, August 1989.

[Rei96] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM* 39(4):71–74, April 1996.

[SE+92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully. Principal features of the VOLTAN family of reliable node architectures for distributed systems. *IEEE Trans. on Computers*, 41(5):542–549, May 1992.

[TKB92] A. S. Tannenbaum, M. F. Kaashoek, and H. E. Balvrije. Parallel programming using shared objects. *IEEE Computer*, pages 10–19, August 1992.

[TM96] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996.

Short Headers Suffice for Communication in a DAG with Link Failures

Faith E. Fich and Andreas Jakoby

Department of Computer Science,
University of Toronto,
Toronto, Ontario, Canada M5S 3G4
(fich,jakoby)@cs.toronto.edu

Abstract. This paper considers a number of communication problems, including end-to-end communication and multicasts, in networks whose underlying graphs are directed and acyclic and whose links are subject to permanent failures. In the case that each processor has separate input queues for each in-edge, we present protocols for these problems that use single bit headers. In the case that each processor has a single input queue for all of its in-edges, we prove that $\Theta(\log d)$-bit headers are necessary and sufficient, where d is the indegree of the graph.

1 Introduction

The end-to-end communication problem is to send a sequence of messages from a sender to a receiver through an unreliable network. It is a fundamental and well studied problem [3–6, 8, 9, 11, 13–15], whose solution allows distributed algorithms to treat unreliable communication networks as if they were reliable channels.

In this paper, we consider the situation in which intermediate vertices are memoryless: they store no information about the state of the communication between the sender and the receiver. When an intermediate vertex receives a packet, it bases its actions on the contents of the packet header. The rest of the packet is assumed to have no effect on a protocol. This is to enforce a clear separation between the protocol and the application programs using this protocol. Memoryless protocols are particularly relevant for public networks such as the Internet. They also provide a good starting point for more general theoretical investigations.

A very simple memoryless protocol for end-to-end communication in a directed acyclic graph (DAG) is for the sender to flood the network with each message it wishes to send. Whenever an intermediate process receives a packet along one of its incoming links, it sends a copy of that packet along all of its outgoing links. To allow the receiver to ignore duplicate copies of a message, each packet header contains a sequence number (the index of the message in the body of the packet among the sequence of messages the sender wishes to transmit). This protocol handles links that can duplicate and reorder messages.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 360–373, 2000.

It can also tolerate permanent link failures, provided that at least one path from the sender to the receiver remains operational. With acknowledgements (sent in the reverse direction through the DAG), message losses (which are equivalent to temporary link failures) can also be handled. The idea is that the sender repeatedly sends a message until it receives an acknowledgement for that message from the receiver. Similarly, when the receiver receives a new message, it repeatedly sends acknowledgements for that message until it receives the next message in the sequence [7, 17, 19].

A bad feature of the flood protocol is that the sizes of the packet headers increase without bound as the number of messages that the sender is transmitting increases. Moreover, if the network is not a DAG, this protocol can generate an infinite amount of traffic, even if the sender only wants to transmit a single message. In general networks, this problem can be avoided by adding a $\lceil \log_2 n \rceil$-bit hop counter to packet headers, where n is the number of processors in the network [16].

Even in the simplest network consisting of two processors and a single link, unbounded sequence numbers are necessary if both packet reordering and duplication can occur [20]. In contrast, the alternating bit protocol [10, 18], which uses one bit headers (specifically, the least significant bit of the sequence number), solves the end-to-end communication problem in this network, provided links cannot reorder packets. However, since intermediate vertices are memoryless, the alternating bit protocol cannot be directly used on links other than direct connections from S to R.

For general networks in which only permanent link failures can occur, Dolev and Welch [11] have a protocol that uses bounded headers: if there are p simple paths from the sender to the receiver, then $O(\log p)$ bit headers suffice. By combining their protocol with the alternating bit protocol performed separately along each simple path between the sender and the receiver, they can also handle packet duplication and loss.

Adler and Fich [1] have proved lower bounds on the size of packet headers, as a function of the network topology, when message losses can occur. For many networks, including complete graphs, series-parallel graphs, and fixed degree meshes, these lower bounds match the packet header lengths used by Dolev and Welch's protocol to within a constant factor. An open question was whether Adler and Fich's lower bounds could be extended to networks that exhibit only permanent link failures.

Here, we give a negative answer to this question. In fact, we prove that, for directed acyclic graphs, single bit headers suffice if only permanent link failures occur. We also extend our protocol to allow streams of messages to be sent from various senders to various receivers.

These results cannot be extended to all graphs. In particular, for any graph that contains the complete graph on k vertices as a minor (such as the k^2-input butterfly or the $k \times k \times 2$ mesh), any memoryless protocol that ensures delivery of a single message from the sender to the receiver using headers with fewer than $\lceil \log_2 k \rceil - 3$ bits, generates an infinite amount of message traffic [1]. Similarly,

for any graph that contains a $3 \times k$ mesh as a minor, $\Omega(\log \log k)$-bit headers are required to send a single message from the sender to the receiver [2].

We also consider a variant of the model where processors are not told the in-edge along which each packet arrives. This might be the case when each vertex has a single input queue rather than a separate input queue for each of its in-edges. In this model, we prove that $\Theta(\log d)$-bit headers are necessary and sufficient for end-to-end communication, where d is the indegree of the graph.

In Section 2, we present a more detailed description of the model. This is followed, in Section 3, by our end-to-end protocol for DAGs that uses one bit headers. Variants of our protocol for other problems and related models appear in Sections 4 and 5. Our lower bound appears in Section 6.

2 The Model

We model the network by a directed acyclic graph, with source S and sink R. Each vertex corresponds to a processor, with S corresponding to the sender and R corresponding to the receiver. An edge (u, v) represents to a direct communication link from the processor corresponding to vertex u and the processor corresponding to vertex v. Packets can only travel in the forward direction along edges. Throughout the paper, we use n to denote the number of processors.

At any point in time, a link is either operational or has failed. Once a link has failed, it remains so and delivers no packets that are sent along it. Operational links do not lose, duplicate, or reorder packets. However, they are asynchronous, delivering every packet within a finite but unbounded amount of time. Hence it is impossible to distinguish between a link which has failed and an operational link which is just very slow (and may contain a sequence of messages which it still has to deliver). We assume that there is always some directed path of operational links from the sender S to the receiver R; otherwise it is impossible to transmit any information from S to R. Processes are assumed to be reliable, although a process which has failed can be simulated by considering all of its out-edges to have failed.

From time to time, the sender S is given a message by an external application which it must send to the receiver R. Eventually, R must report the exact sequence of messages given to S.

When S is given a message, it creates packets containing the message and sends them to its neighbours. The packet headers and which packets to send along each out-edge can be based on its current state (but not the message contents). On receipt of a packet, a processor can forward the message in the packet by sending packets with the same or different headers along its out-edges. It can also send packets that only contain header information. The number of packets the processor sends along each out-edge, the headers of those packets, and whether or not a given packet contains a copy of the message can depend on the header of the packet and the edge along which the packet arrived. Because we are considering only memoryless algorithms, these decisions must be independent of the past history of the communication through the network.

Links may fail at any time and it is possible that only one directed S–R path of operational edges will exist. Therefore, a processor that receives a packet with a message must send at least one packet containing the message along each of its out-edges.

3 A Protocol with One Bit Headers

In this section, we present a protocol to send an arbitrarily long sequence of messages through a directed acyclic graph $G = (V, E)$ from a source vertex S to a sink vertex R. The protocol uses packets with single bit headers.

Let d be the maximum indegree of any vertex and let $f : E \to \{0, \ldots, d-1\}$ be any function that numbers the edges into each vertex. More specifically, if $e = (u, v)$ then $f(e) < indegree(v)$ and, if $e' = (u', v)$, where $u \neq u'$, then $f(e) \neq f(e')$.

Consider the following protocol:

- When the sender S is given a message m to send to the receiver R, it sends a packet with message m and an arbitrary header along all its out-edges.
- When an intermediate vertex v receives a packet p along an in-edge e, then along each of its out-edges, v sends the sequence of $\ell = \lceil \log_2 indegree(v) \rceil$ packets with headers $b_1, \ldots, b_\ell$ and no messages, immediately followed by the packet p, where $b_1 \cdots b_\ell$ is the ℓ-bit binary representation of $f(e)$. In particular, if v has indegree 1, then v just forwards all the packets it receives to each of its out-edges.
- The receiver R records the sequence of packets it receives along each in-edge. From this information, it determines which messages to report.

For example, consider the following network.

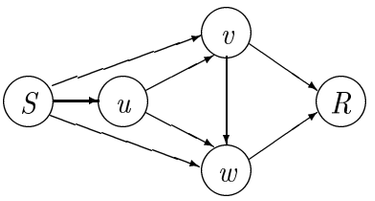


Fig. 1. A Directed Acyclic Network

If the sender S is given only one message m and none of the links fail, then there is an execution in which:

- S and u send the packet $[0, m]$ along all their out-edges,

- v sends the sequence of packets $[0,-]$,$[0,m]$,$[1,-]$,$[0,m]$ along its two out-edges, and
- w sends the sequence of packets $[0,-]$,$[1,-]$,$[0,m]$, $[1,-]$,$[0,-]$,$[0,-]$, $[1,-]$,$[0,-]$,$[0,m]$, $[1,-]$,$[0,-]$,$[1,-]$, $[0,-]$,$[0,-]$,$[0,m]$, $[1,-]$,$[0,-]$,$[0,m]$ along its out-edge.

It remains to describe how R determines the sequence of messages to report from the information it records. We shall show that the messages R reports are in the same order that S sends them, R does not duplicate messages, and, provided there is an S–R path of operational edges, R reports all the messages that S sends.

When an intermediate processor (i.e. a vertex other than S or R) receives a packet p, we say that the packets it sends in response are *directly caused by* p. The *caused by* relation is the reflexive transitive closure of the directly caused by relation. In other words, a packet p' is caused by a packet p if $p' = p$ or p' is directly caused by a packet which, in turn, is caused by p. A packet p is said to have *followed* directed path $\pi = v_1, \ldots, v_k$, if there is a sequence of packets $p_1, \ldots, p_{k-1}$, where $p_{k-1} = p$, p_i travelled along edge (v_i, v_{i+1}), for $i = 1, \ldots, k-1$, and packet p_i is directly caused by packet p_{i-1} for $1 < i \leq k-1$.

For any directed path π to R in the graph, let $s(\pi)$ denote the vertex at the beginning of path π and let

$$\ell(\pi) = \lceil \log_2 \max\{1, indegree(s(\pi))\} \rceil .$$

We begin by presenting a scheme in which R stores a lot of information. This facilitates the proof of correctness. Then we show how R can do essentially the same thing while storing significantly less information.

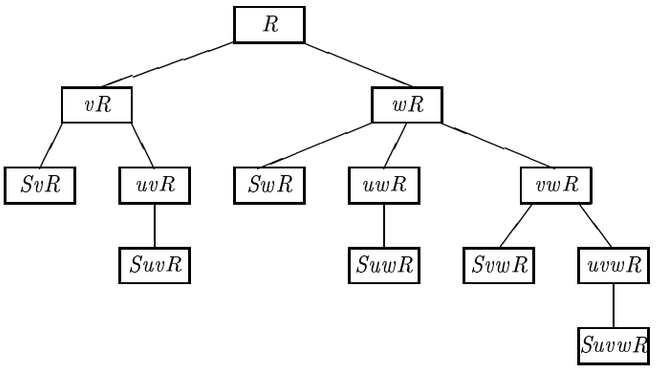


Fig. 2. The Tree Used by R for the Network in Figure 1

The receiver R uses a rooted d-ary tree to record and process the packets it has received. The nodes of the tree are the directed paths to R that start at

vertices reachable from S (i.e. paths that are suffixes of directed S–R paths). The empty path to R is the root of the tree. Node π is the i'th child of node ρ if π is ρ preceded by the i'th edge into ρ, i.e., more formally, if $\pi = (s(\pi), s(\rho))\rho$ and $f(s(\pi), s(\rho)) = i - 1$. Since S is a source, all directed S–R paths are leaves in the tree. The tree for the network in Figure 1 is given in Figure 2. At each node of the tree, except the root, a list of packets is kept.

- When R receives a packet p along its i'th in-edge, it appends the packet to the end of the list at the i'th child of the root.
- Whenever R appends a packet p to the list at an internal node π causing the length of the list to become equal to a multiple of $\ell(\pi) + 1$, and $b_1, \ldots, b_{\ell(\pi)}$ are the headers of the previous $\ell(\pi)$ packets in the list, R also appends a copy of the packet p to the list at the i'th child of node π, where $b_1 \cdots b_{\ell(\pi)}$ is the binary representation of $i - 1$. In particular, if $s(\pi)$ has only one child, then R immediately copies each packet in $s(\pi)$'s list to its child's list.
- Whenever R appends a packet to the list at an S–R path, causing the length of this list to be greater than the number of messages R has reported, R reports the message part of this packet.

For the execution used in the example above, where S is given only one message m and none of the links fail,

- the list of packets stored at node wR is $[0, -]$,$[1, -]$,$[0, m]$, $[1, -]$,$[0, -]$,$[0, -]$, $[1, -]$,$[0, -]$,$[0, m]$, $[1, -]$,$[0, -]$,$[1, -]$, $[0, -]$,$[0, -]$,$[0, m]$, $[1, -]$,$[0, -]$,$[0, m]$,
- the list of packets stored at nodes vR and vwR is $[0, -]$,$[0, m]$,$[1, -]$,$[0, m]$, and
- the list of packets stored at nodes SvR, uvR, $SuvR$, SwR, uwR, $SuwR$, $SvwR$, $uvwR$, and $SuvwR$ is $[0, m]$.

For any nonempty directed path $\pi = v_1, \ldots, v_k$ from $v_1 = s(\pi)$ to $v_k = R$ in the graph, let

$$L(\pi) = \prod_{1 \le i < k} \lceil 1 + \log_2 \max\{1, indegree(v_i)\} \rceil.$$

If no links on path π fail, each packet received by $s(\pi)$ causes $L(\pi)$ packets that follow path π to arrive at R and each packet sent by $s(\pi)$ causes

$$L(\pi)/(\ell(\pi) + 1) = \prod_{1 < i < k} \lceil 1 + \log_2 \max\{1, indegree(v_i)\} \rceil$$

packets that follow path π to arrive at R. This is because, whenever processor $v_i \neq R$ receives a packet, it sends $1 + \lceil \log_2(\max\{1, indegree(v_i)\}) \rceil$ packets along edge (v_i, v_{i+1}). For the example in Figure 1, $L(vR) = L(SvR) = L(uvR) = L(SuvR) = 2$, $L(wR) = L(SwR) = L(uwR) = L(SuwR) = 3$, and $L(vwR) = L(SvwR) = L(uvwR) = L(SuvwR) = 6$.

The next result provides the key invariant satisfied by our protocol.

Lemma 1. *If R has received n packets that followed path π, then the list of packets stored at node π is the length $\lfloor n(\ell(\pi)+1)/L(\pi) \rfloor$ prefix of the sequence of packets sent by processor $s(\pi)$.*

Proof. The proof is by induction on the length of π. If π consists of a single edge into R, then $L(\pi) = \ell(\pi) + 1$. Since links do not reorder, duplicate, or lose messages, the sequence of packets that R receives along this edge is a prefix of the sequence of packets sent by processor $s(\pi)$. Processor R records all $n = \lfloor n(\ell(\pi)+1)/L(\pi) \rfloor$ packets it receives along this edge in the list at node π. Hence the claim is true for π.

Suppose the claim is true for node π. Let $\pi_1, \ldots, \pi_k$ be the children of π and let n_i denote the number of packets received by R that followed path π_i. Then R has received $n = n_1 + \cdots + n_k$ packets that followed path π. Links do not reorder, duplicate, or lose packets and, if no links of π fail, each packet received by $s(\pi)$ causes $L(\pi)$ packets to follow path π. These facts imply that $s(\pi)$ has received at least $\lceil n/L(\pi) \rceil$ packets, the first $\lfloor n/L(\pi) \rfloor$ of which have each caused $L(\pi)$ consecutive packets that followed path π. If n is not divisible by $L(\pi)$, then the $\lceil n/L(\pi) \rceil$'th packet received by $s(\pi)$ has caused $n - L(\pi)\lfloor n/L(\pi) \rfloor < L(\pi)$ packets that followed path π. All remaining packets received by $s(\pi)$ have caused no packets that followed path π.

Of the first $\lfloor n/L(\pi) \rfloor$ packets received by $s(\pi)$, let n'_i denote the number received by $s(\pi)$ along its i'th in-edge, so $n'_1 + \ldots + n'_k = \lfloor n/L(\pi) \rfloor$. Each of these n'_i packets was sent by $s(\pi_i)$ and caused $L(\pi)$ packets at R that followed path π_i. Therefore $n_i \geq n'_i L(\pi)$. It follows that $\lfloor n_1/L(\pi) \rfloor + \cdots \lfloor n_k/L(\pi) \rfloor \geq n'_1 + \ldots + n'_k = \lfloor n/L(\pi) \rfloor = \lfloor (n_1 + \cdots + n_k)/L(\pi) \rfloor \geq \lfloor n_1/L(\pi) \rfloor + \cdots + \lfloor n_k/L(\pi) \rfloor$, so $n'_i = \lfloor n_i/L(\pi) \rfloor$, for $i = 1, \ldots, k$.

By the induction hypothesis, the list of packets stored at π is the length $n' = \lfloor n(\ell(\pi)+1)/L(\pi) \rfloor$ prefix of the sequence of packets sent by processor $s(\pi)$. Processor $s(\pi)$ sends a block of $\ell(\pi) + 1$ consecutive packets along each of its out-edges in response to each packet it receives. Thus, the n' packets stored at node π are directly caused by the first $\lceil n'/(\ell(\pi)+1) \rceil$ packets received by $s(\pi)$. Furthermore, the first $\lfloor n'/(\ell(\pi)+1) \rfloor = \lfloor n/L(\pi) \rfloor$ packets received by $s(\pi)$ each directly caused a block of $\ell(\pi) + 1$ consecutive packets stored at node π. The first $\ell(\pi)$ packets in each block indicate the edge on which the packet causing this block of packets arrived at $s(\pi)$ and, hence, which processor $s(\pi_i)$ sent it. The last packet in the block is a copy of this packet, which R copies to the list stored at the corresponding path π_i. Therefore, for $i = 1, \ldots, k$, the list of packets stored at node π_i is the length $n'_i = \lfloor n_i/L(\pi) \rfloor = \lfloor n_i(\ell(\pi_i)+1)/L(\pi_i) \rfloor$ prefix of the sequence of packets sent by processor $s(\pi_i)$. □

Our main result follows directly from this lemma.

Theorem 1. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform end-to-end communication from a sender to a receiver.*

Proof. From Lemma 1, the list at each leaf node is a prefix of the sequence of packets sent by S. The t'th packet reported by R is the t'th packet from one of these lists. Thus, the sequence of packets reported by R is a prefix of the sequence of packets sent by S.

Now suppose that all the edges on some S–R path π remain operational. Then each packet given to S causes $L(\pi)$ packets to follow path π to R. If S has been given t packets, eventually R receives $tL(\pi)$ packets that followed path π. By Lemma 1, the list at node π will eventually have length t and, thus, the t'th packet will be reported by R. □

At each internal node π, R uses the single bit headers of the $\ell(\pi)$ previous packets plus a modulo $\ell(\pi)+1$ counter. Thus, only $\ell(\pi) + \lceil \log_2(\ell(\pi)+1) \rceil$ bits are needed to represent the required information at this node. Similarly, R only uses the length of the list stored at a leaf, hence a counter suffices in place of the list. In addition, R must keep track of the number of messages it has reported.

Notice that, on receipt of a packet, each processor sends the same sequence of packets on all of its out-edges. Thus the protocol will work unchanged if a broadcast is used in place of these sends.

4 Variants of the Basic Protocol for Related Problems

With simple changes, the basic protocol described in Section 3 can be adapted to solve more general communication problems on a network with a directed acyclic underlying graph $G = (V, E)$. We consider both multiple senders and multiple receivers and require all, one or some of the receivers to report a particular message. In all these cases, the resulting protocols use single bit headers.

The simplest variant allows the sender to perform a sequence of broadcasts to a set of receivers.

Theorem 2. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform a sequence of broadcasts from a sender to a set of receivers.*

Proof. Let $\mathcal{R}$ denote the set of receivers. Each vertex $u \in \mathcal{R}$ behaves like the single receiver R, as described in Section 3, using a tree data structure rooted at the empty path to u. Those vertices in $\mathcal{R}$ that are not sinks also send packets along their out-edges in response to receiving a packet. □

It is possible to extend this protocol to handle the situation where different messages can be sent to different receivers.

Theorem 3. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform end-to-end communication from a sender to a set of receivers.*

Proof. Imagine that, for each receiver $u \in \mathcal{R}$, there is one source node $u' \notin V$, which has a single out-edge leading to S. To send a message to u, the sender S pretends that it has received a packet from u' containing the message. This causes S to send $\lceil \log_2 |\mathcal{R}| \rceil$ packets, each consisting of a single header bit, which together form the $\lceil \log_2 |\mathcal{R}| \rceil$-bit binary representation of the index of (u', S) among the in-edges of S, followed by a packet containing the message. packet could contain the message, In effect, the message is preceded by packets that specify the receiver to whom it is directed. Each receiver $u \in \mathcal{R}$ only keeps track of the leaves corresponding to u'–u paths and reports a new message whenever the maximum of the lengths of the lists at these leaves increases. □

Another problem is to handle a set of senders $\mathcal{S}$, each of which may have a sequence of messages to transmit to the receiver R. Here R separately reports the messages received from each processor in $\mathcal{S}$. A similar modification to the basic protocol will handle this situation, provided that each processor in $\mathcal{S}$ has at least one nonempty directed path to R.

Theorem 4. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform end-to-end communication from a set of senders to a receiver.*

Proof. Imagine a source node $S \notin V$ that is connected to all the vertices in $\mathcal{S}$ and is the source of all messages. This increases the indegree of every vertex in $\mathcal{S}$ by 1. Each vertex $v \in \mathcal{S}$ that wants to send a message to R behaves as if S had just sent it a packet containing the message. Then, each packet R receives that appears to have followed a path beginning with edge (S, v), actually originated at vertex v.

Now R uses $|\mathcal{S}|$ counters, one to keep track of the number of messages originating at each of the vertices in $\mathcal{S}$. Whenever R copies a packet to the list at a leaf (i.e. at a node corresponding to a path π from the imaginary source node S to R), R compares the new length of this list to the number of messages it has reported that originated from v, the second vertex on path π. If it is larger, then R reports the message part of this packet to be the next message that originated from v.

Note that, if all of the senders in $\mathcal{S}$ are source nodes in the network, the imaginary source node S is not needed. In this case, the leaves in the tree are all the directed paths to R from nodes in $\mathcal{S}$. □

The ideas in Theorems 3 and 4 can be combined to handle multiple senders and receivers.

Theorem 5. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform end-to-end communication from a set of senders to a set of receivers.*

Proof. Let $\mathcal{R}$ denote the set of receivers and let $\mathcal{S}$ denote the set of senders. As in the proof of Theorem 3, for each receiver $u \in \mathcal{R}$, there is an imaginary source node $u' \notin V$ and the leaves in u's tree are all u'–u paths. Each sender $v \in \mathcal{S}$ has an in-edge (u', v) for each receiver $u \in \mathcal{R}$ with which it may communicate. Whenever v has a message to send to $u \in \mathcal{R}$, it pretends that it has received a packet from u' containing the message.

As in the proof of Theorem 4, each receiver $u \in \mathcal{R}$ has one counter for each sender $v \in \mathcal{S}$ that may communicate with it, to keep track of the number of different personal messages it has received originating from v, i.e. following paths that seem to begin with edge (u', v). Whenever one of these counters is incremented, u reports the message from the packet under consideration. □

A simple modification extends this protocol to handle multicasts, where senders may wish to broadcast messages to subsets of the receivers.

Theorem 6. *Consider a network whose underlying graph is a DAG and whose links are subject to permanent failure. There is a memoryless protocol for this network that uses one bit headers to perform a sequence of multicasts from a set of senders to a set of receivers.*

Proof. The idea is to have a separate imaginary source node for each desired subset of receivers, with edges directed from it to each sender that may wish to broadcast to this subset of receivers. Each leaf in u's tree is a path to u originating from an imaginary node that represents a set which contains u. □

As was the case for the basic protocol described in Section 3, all of the protocols described in this section will work unchanged if broadcasts are used instead of sends.

5 Variants of the Protocols for Other Models

In some models, processors do not have built-in access to information about the source of each of their incoming packets. In other words, processors are not told on which of their in-edges each packet arrives. We call such models *source oblivious*. Source oblivious models may be appropriate when each vertex has a single input queue, rather than a separate input queue for each of its in-edges. Any of the previous protocols can be modified to work in source oblivious models by using longer headers. For example, the following result is the analogue of Theorem 1.

Theorem 7. *Consider a network whose underlying graph is a DAG with indegree d and links that are subject to permanent failure. There is a memoryless, source oblivious protocol for this network that uses headers of length $1 + \lceil \log_2 d \rceil$ to perform end-to-end communication from a sender to a receiver.*

Proof. The longer headers allow the protocol described in Section 3 to be simulated in this weaker model. Specifically, when a processor sends a packet along

edge e to processor v, it includes the $\lceil \log_2 indegree(v) \rceil$-bit binary representation of $f(e)$ as part of the header. When v receives the packet, it strips off this information and uses it to identify the in-edge along which the packet arrived. □

Notice that processors send different packets along their different out-edges. Therefore broadcasts do not suffice for this protocol. If each processor has a single input queue and can only broadcast packets to its out-neighbours, there is another variant of the protocol that can be used, instead.

Theorem 8. *Consider a network whose underlying graph $G = (V, E)$ is a DAG with n vertices, indegree d, outdegree D, and links that are subject to permanent failure. There is a memoryless, source oblivious protocol for this network that uses headers of length $1 + \lceil \log_2 \min\{n, D(d-1)+1\} \rceil$ to perform end-to-end communication from a sender to a receiver, where processors only broadcast packets.*

Proof. Consider the hypergraph $G' = (V, E')$, where

$$E' = \{\text{in-neighbours}(v) \mid v \in V\}.$$

The *degree* in G' of vertex $v \in V$ is the number of nodes that share at least one hyperedge with v and $degree(G') = \max_{v \in V} degree(v)$. Since the indegree of G is d and its outdegree is D, it follows that $degree(G') \leq \min\{n, D(d-1)+1\}$.

A *strong colouring* is a function assigning each vertex a colour so that no hyperedge contains two vertices of the same colour. There is a strong colouring $g : V \to \{0, 1, \ldots, c-1\}$ of G' using at most $c = 1 + degree(G')$ colours [12].

As in Theorem 7, the longer headers allow the protocol described in Section 3 to be simulated. When a processor u wants to send a packet to its out-neighbours, it includes the $\lceil \log_2 c \rceil$-bit colour $g(u)$ as part of the header. Let v be an out-neighbour of u. Since all of v's other in-neighbours are contained in the hyperedge in-neighbours(v) together with u, they all have a colour different from $g(u)$. Therefore, when processor v receives the packet, it can strip off the colour from the header and use this information to identify its in-neighbour u that sent the packet. □

6 A Lower Bound

In this section, we prove that headers of length $\lceil \log_2 d \rceil$ are necessary for end-to-end communication in a DAG of indegree d using source oblivious protocols, i.e. when processors are not told the in-edge along which each packet arrives. In other words, the behaviour of a processor on receipt of a packet cannot directly depend on the in-edge on which the packet arrives, but only on the packet header (and, for processor R, its state).

Our lower bound is for the class of *data oblivious* protocols, where processors (including S and R) do not perform actions based on the contents of the

messages being sent [1, 6]. In particular, processors cannot compare messages or perform computation on them. This is appropriate when one views end-to-end communication protocols as providing a reliable communication layer that will be used by many different distributed algorithms.

Also, it is necessary to assume that there is a bound on the number of packets sent by S when it is given a message to send to R. Otherwise, there are counting protocols [3, 18] that use no headers. In these protocols, for each successive message in the sequence it wants to send to the receiver R, the sender S broadcasts an exponentially increasing number of packets containing that message. Moreover, R performs equality tests on the packets.

The following lower bound shows that each processor must be able to distinguish between the packets arriving on different in-edges, so that R can avoid confusing packets caused by different messages.

Theorem 9. *Consider any network whose underlying graph G is a DAG with a single source S, a single sink R, and indegree d. Then any memoryless, source oblivious, data oblivious protocol for sending an arbitrarily long sequence of messages from S to R in this network must use at least d different headers (and hence have header length at least $\lceil \log_2 d \rceil$).*

Proof. To obtain a contradiction, suppose there is a protocol for end-to-end communication from S to R in G that is memoryless, source oblivious, data oblivious, and uses fewer than d different headers. Let H be the set of different packet headers used by the protocol.

Let v be a vertex of G with indegree d and let I denote the set of edges into v. Let π be a directed path from v to R and let T be a tree of directed edges connecting S to all the in-neighbours of v. Suppose that all edges in π, T, and I are operational and all others edges have failed. The edges in π and T will have delay 0, but the speed of the edges into v will be controlled by an adversary.

Suppose that S is given an infinite sequence of different messages to send to R. Consider the resulting infinite sequence of packets $P(e)$ that travel along each edge $e \in I$. Let $H(e)$ denote the set of packet headers that occur infinitely often in $P(e)$.

Consider the bipartite graph with vertex set $I \cup H$ and an edge between $e \in I$ and $h \in H$ if and only if $h \in H(e)$. Let M be any maximal matching of this graph. Since $|I| > |H|$, there exists $e' \in I$ which is unmatched. Since M is maximal, each $h \in H(e')$ is matched; otherwise edge (e', h) could be added to the matching. Let $m(h) \in I$ denote the match of h for each $h \in H(e')$.

Let B be an upper bound on the number of packets in $P(e)$ caused by a single message given to S, for all $e \in I$. This value must exist because there is a bound on the number of packets sent by S when it is given a message, and the intermediate processors are memoryless, so each packet received by an intermediate processor can directly cause only a finite number of outgoing packets. Let k be sufficiently large so that

- the prefix of $P(e')$ caused by the first k messages contains all headers that do not occur in $P(e')$ infinitely often (i.e. all $h \in H - H(e')$), and

– for each $h \in H(e')$, the prefix of $P(m(h))$ caused by the first k messages contains at least B occurrences of header h.

The adversary constructs two executions with the property that R reports the wrong sequence for at least one of them. For both executions, the adversary begins by giving S the first k messages. It allows all packets on edge e' to travel with delay 0, but makes all other edges $e \in I$ so slow that not one packet has yet arrived at v along any of them.

Then the adversary gives message $k+1$ to S. Suppose this message causes a sequence of packets on edge e' with headers $h_1, \ldots, h_b$, where $b \leq B$. Before delivering the next packet to v along edge e', the adversary delivers packets to v from $P(m(h_1))$ (i.e. along edge $m(h_1)$) up to but not including the first occurrence of h_1. In the first execution, the adversary delivers the packet with header h_1 along edge e' followed by the packet with header h_1 along edge $m(h_1)$. In the second execution, the adversary delivers these two packets to v in the opposite order. This continues for the remaining $b-1$ packets on edge e'.

Next, the adversary causes all of the edges in $I - \{e'\}$ to fail and the edge e' is given 0 delay. The adversary gives messages to S until R reports $k+1$ messages. Since the protocol is data oblivious, these two executions are indistinguishable to v and, hence, to R.

To be correct, the last message R reports must be taken from one of the packets caused by message $k+1$. By construction, all such packets are caused by packets that travelled along edge e'. In fact, by the choice of k, all packets received by v along edges other than e', in either of the two executions, are caused by one of the first k messages. Thus, for each packet caused by message $k+1$ that travelled along edge e' in one of the two executions, the corresponding packet in the other execution is not caused by message $k+1$. This implies that, in at least one of the two executions, the last message R reports is taken from a packet not caused by message $k+1$ and, hence, is incorrect. □

Using similar techniques, we can prove a lower bound on the number of packet transmissions caused by one message in any memoryless, data oblivious, end-to-end protocol with single bit headers that is close to the number caused by one message in the basic protocol, given in Section 3.

Acknowledgements

We thank Andre Kundgen and Radhika Ramamurthi for giving us very useful information about hypergraph colouring. This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario.

References

1. Micah Adler and Faith E. Fich, *The Complexity of End-to-End Communication in Memoryless Networks*, 18th Annual ACM Symposium on Principles of Distributed Computing, May 1999, pages 239–248.

2. Micah Adler, Faith E. Fich, Leslie Ann Goldberg, and Mike Paterson *Tight Size Bounds for Packet Headers in Narrow Meshes*, to appear in the Proceedings of the 27th International Colloquium on Automata, Languages and Programming, 2000.
3. Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, and L. Zuck, *Reliable Communication Over Unreliable Channels*, Journal of the ACM, vol. 41, no. 6, 1994, pages 1267–1297.
4. Y. Afek, B. Awerbuch, E. Gafni, Y. Mansour, A. Rosèn, and N. Shavit, *Slide–The Key to Polynomial End-to-End Communication*, Journal of Algorithms, vol. 22, no. 1, 1997, pages 158–186.
5. Y. Afek and E. Gafni, *End-to End Communication in Unreliable Networks*, Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, 1988, pages 131–148.
6. Y. Afek and E. Gafni, *Bootstrap Network Resynchronization*, Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing, 1991, pages 295–307.
7. B. Awerbuch and S. Even, *Reliable Broadcast Protocols in Unreliable Networks*, Networks, vol. 16, 1986, pages 381–396.
8. B. Awerbuch, Y. Mansour, and N. Shavit, *Polynomial End to End Communication*, Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, 1989, pages 358–363.
9. B. Awerbuch, B. Patt-Shamir, and G. Varghese, *Self-stabilizing End-to-End Communication*, Journal of High Speed Networks, vol. 5, no. 4, 1996, pages 365–381.
10. K. Bartlett, R. Scantlebury, and P. Wilkinson, *A Note on Reliable, Full-Duplex Transmission over Half-Duplex Links*, Communications of the ACM, vol. 12, 1969, pages 260–261.
11. S. Dolev and J. Welch, *Crash Resilient Communication in Dynamic Networks*, IEEE Transactions of Computers, vol. 46, 1997, pages 14–26.
12. R. Dutton and R. Brigham, *Strong, weak and other colorings of uniform hypergraphs*, Proceedings of the Sixteenth Southeastern Conference on Combinatorics, Graph Theory and Computing, Congressus Numerantium, vol. 47, 1985, pages 299–314.
13. F. Fich, *End-to-end Communication*, Proceedings of OPODIS '98, Amiens, France, 1998, pages 37–43.
14. E. Kushilevitz, R. Ostrovsky, and A. Rosén, *Log-Space Polynomial End-to-End Communication*, Proceedings of the 28th ACM Symposium on the Theory of Computing, 1995, pages 559–568.
15. R. Ladner, A. LaMarca, and E. Tempero, *Counting Protocols for Reliable End-to-End Transmission*, Journal of Computer and Systems Sciences, vol. 56, no. 1, 1998, pages 96–111.
16. J. Postel, *Internet Protocol*, Network Working Group Request for Comments 791, September 1981.
17. M. Stenning, *A Data Transfer Protocol*, Computer Net., vol. 1, 1976, pages 99–110.
18. E. Tempero and R. Ladner, *Recoverable Sequence Transmission Protocols*, Journal of the ACM, vol. 42, no. 5, 1995, pages 1059–1090.
19. U. Vishkin, *A Distributed Orientation Algorithm*, IEEE Trans. on Information Theory, vol. 29, 1983, pages 624–629.
20. D. Wang and L. Zuck, *Tight Bounds for the Sequence Transmission Problem*, 8th Annual ACM Symposium on Principles of Distributed Computing, 1989, pages 73–83.

Consistency Conditions for a CORBA Caching Service*

Gregory Chockler[1], Roy Friedman[2], and Roman Vitenberg[2]

[1] Institute of Computer Science, The Hebrew University, Givat Ram, Jerusalem 91904, Israel,
grishac@cs.huji.ac.il,
WWW home page: http://www.cs.huji.ac.il/~grishac
[2] Computer Science Department, Technion – The Israel Institute of Technology, Haifa 32000, Israel,
{roy,romanv}@cs.technion.ac.il,
WWW home page: http://www.cs.technion.ac.il/{~roy,~romanv}

Abstract. Distributed object caching is essential for building and deploying Internet wide services based on middlewares such as CORBA. By caching objects, it is possible to mask much of the latency associated with accessing remote objects, to provide more predictable quality of service to clients, and to improve the scalability of the service. This paper presents a combined theoretical and practical view on specifying and implementing consistency conditions for such a service. First, a formal definition of a set of basic consistency conditions is given in an abstract, implementation independent manner. It is then shown that common consistency conditions such as sequential consistency, causal consistency, and PRAM can be formally specified as a combination of these more basic conditions. Finally, the paper describes the implementation of the proposed basic consistency conditions in CASCADE, a distributed CORBA object caching service.

1 Introduction

Object caching is a promising approach for improving the scalability, performance, and predictability of Internet oriented services that are based on object-oriented middlewares such as CORBA [17]. Accessing a local or nearby cache incurs a much lower latency than accessing a far away object, and the access time and availability of a cached copy is much more predictable than when accessing a remote object. Also, object caching greatly enhances the scalability of services because most client requests can be satisfied from a local cache and the service provider is relieved from the burden of servicing a large number of concurrent clients.

An inevitable side-effect of caching is the need to maintain copies of the same cached object consistent, at least to some degree. In this paper, we explore, both

* This work was supported in part by the Israeli Ministry of Science grant number 1230-1-98.

M. Herlihy (Ed.): DISC 2000, LNCS 1914, pp. 374–388, 2000.

theoretically and practically, a flexible approach to consistency of such services. That is, we start by formally defining a basic set of consistency conditions, in an abstract, implementation independent, way, and show how other consistency conditions can be implemented as a combination of these basic conditions. We then describe how we have implemented these conditions within CASCADE [7], our CORBA object caching service [1]. The paper also describes some interesting optimizations we employed in this implementation.

Our specifications and resulting implementation have the following distinct features:

Rigor: We provide a formal definition of basic consistency conditions, given from the application point of view, as requirements on possible ordering of clients' local histories. As discussed in [4], this implementation independent approach yields more rigorous definitions, and it is easier to prove program correctness with such definitions than with operational definitions.

Modularity: Our conditions can be combined in various ways to yield guarantees with different levels of strength and complexity. This approach allows the known tradeoff between the strength of the consistency semantics and the overhead it imposes (cf. [4]) to be taken into consideration when configuring the set of consistency guarantees for a particular application. For users of our service, this means that they have more freedom in choosing the exact consistency semantics they need. From the implementation standpoint, this yields a more modular implementation. Since the implementation can be divided into basic conditions, each of which is easier to implement than, say sequential consistency [12], the entire implementation is simpler, and therefore more robust. Similarly, the implementation correctness proof is easier, since we can prove the correctness of the implementation of each basic condition separately; the formal proof about the combination of these conditions then immediately implies that our service correctly implements the corresponding more elaborate consistency conditions, e.g., sequential consistency.

Comprehensiveness and usefulness for applications: The presented specifications cover a wide range of consistency requirements for distributed applications. This is shown by proving that many existing consistency conditions such as sequential consistency [12], PRAM [13], and causal consistency [2] can be specified as certain combinations of our basic conditions. We also discuss usefulness of other combinations and analyze the interdependencies within the set of guarantees. In the full version of this paper we present examples of several applications, each of which requires some of our guarantees or a combination of them. Moreover, we show there that all of our conditions are indeed useful, i.e., that there are applications that require each of them.

[1] In [7], we described CASCADE, the motivation behind it, its general implementation and a performance analysis. The current paper is the first place where we formally specify the basic consistency conditions, and elaborate on their exact implementation within CASCADE.

Although, our implementation of consistency conditions is based on the widely known notion of version number and vectors, it is nevertheless unique in exploiting the peculiarities of hierarchical cache architecture such as the one used in CASCADE. We envision that similar techniques can be applied to other systems that employ hierarchical caches.

Finally, our implementation preserves consistency guarantees even when clients access cached copies at different servers during the execution. Furthermore, we have designed novel optimizations that reduce the amount of information transferred between roaming clients and static servers. We believe that this latter contribution can be applied to other systems where it is required to maintain consistency for mobile clients. As Internet mobile clients become more common, we expect that our techniques will be useful for a wider range of applications.

1.1 Related Work

Many consistency conditions have been defined and investigated, mostly in the context of distributed shared memory, e.g., [2, 4, 9, 12, 13, 16, 18] and databases, e.g., [14, 21]. Vast amount of research was dedicated to implementing shared memory systems with various consistency guarantees, including *sequential consistency* (sometimes referred to as *strong consistency*) [12], *weak consistency* [10], *release consistency* [6], *causal consistency* [2], *lazy release consistency* [11], *entry consistency* [5], and *hybrid consistency* [8]. In contrast to our service, such systems are geared towards high-performance computing, and generally assume non-faulty environments and fast local communication.

Much less attention, however, was devoted to exploring consistency guarantees suitable for object-oriented middlewares, especially for middlewares in which a client is not bound to a particular server and can switch the servers all the time.

Our work is motivated by Bayou project [21], which introduced a set of basic consistency conditions for sessions of mobile clients and discussed version vectors as a possible way of their implementation. This work also brought numerous examples illustrating that these conditions are indeed useful for applications. However, these definitions are introduced in [21] as constraints on an implementation and are defined in a framework of a particular database model.

The Globe system [22] follows an approach similar to CASCADE by providing a flexible framework for associating various replication coherence models with distributed objects. Among the coherence models supported by Globe are the PRAM coherence, the causal coherence, the eventual coherence, etc.

2 Definitions and Conventions

We generally adopt the model and definitions as provided in [3] and [1], but slightly adjust them to our needs. We assume a world consisting of *clients* and

servers. Clients invoke *methods* on objects as specified in a *program*. These methods are then transformed into messages sent to one or more servers. The servers can exchange messages among themselves and eventually send a reply to the client. We assume that message delivery is (eventually) reliable and FIFO, and that processes do not fail. However, the network might delay messages for an arbitrarily long time and neither clients nor servers have access to real-time clocks. The rationale behind this failure model is discussed in the full version of this paper.

We assume that each method operates on one object, but each object might have several read/write variables. Our definitions below are given from the client point of view and thus, for the rest of this section, we will no longer discuss servers; servers will be important in discussing the implementation (Section 4). Also, our definitions and discussions assume one object [2]. Note that since each object has multiple variables, each object can be thought of as a single distributed shared memory.

We assume that methods can be classified as either queries or updates, depending on whether they simply return the value of variables they access, or change them. To make the definitions comparable to the ones used in distributed shared memory research, we will refer to updates as WRITEs and to queries as READs. Each method can either read or write several variables atomically. In particular, a single READ operation might return values written by several WRITE operations.

A *local execution* of a client process p_i, denoted σ_i, is a sequence of READ and WRITE operations, denoted $o_1, o_2, \ldots$, that are performed by p_i. We assume that client's operations are always ordered in its local history in the order specified in the program. For the sake of simplicity, we will omit the variables accessed by an operation whenever possible. In what follows, we sometimes refer to local execution as session, and use these terms interchangeably. A *global execution*, or just *execution* σ, is a collection of local executions for a given system run, one for each client of the system.

Given a sequence S of operations, we denote $o_1 \xrightarrow{S} o_2$ when o_1 precedes o_2 in the sequence. An execution σ induces a partial order, $\xrightarrow{\sigma}$, on the operations that appear in σ: $o_1 \xrightarrow{\sigma} o_2$ if $o_1 \xrightarrow{\sigma_i} o_2$ for some p_i.

For a given execution σ and a process p_i, denote by $\sigma|i$ the restriction of σ to events of p_i; denote by $\sigma|i + w$ the partial execution consisting of all the operations of p_i and all the WRITE operations of other processes. Similarly, for a given sequence S of operations, denote by $S|i$ the restriction of S to operations invoked by p_i and denote by $S|w$ the restriction of S to WRITE operations.

We use the standard notions of *serializations*, *legal serializations* and *consistency conditions* as defined, e.g., in [4].

[2] This is sufficient for CASCADE in which consistency conditions are indeed provided per each object since each object has a separate hierarchy. However, in the future it would be interesting to extend the definitions to multiple objects.

3 Consistency Conditions

3.1 Basic Consistency Conditions

Eventual Propagation: For every process p_i there exists a legal serialization S_i of $\sigma|i+w$.

This requirement essentially expresses liveness of update propagation: For a given execution and a given update in this execution, if some process invokes an infinite number of queries, it will eventually see the result of this update. An implementation in which updates are not propagated does not guarantee any level of consistency. Henceforward, we assume that this condition always holds.

Let us define a *serialization set of* σ as a set of legal serializations of $\sigma|i+w$, one for each p_i. Due to Eventual Propagation, at least one serialization set exists for a given execution.

We now present five session guarantees. Each guarantee is defined as a predicate that takes a serialization or a serialization set and verifies whether this set satisfies the condition w.r.t. a session.

Read Your Writes: For a given execution σ and a process p_i, a valid serialization S_i of $\sigma|i+w$ preserves *Read Your Writes for the session* σ_i if for every two operations o_1 and o_2 in σ_i such that $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

FIFO of Reads: For a given execution σ and a process p_i, a valid serialization S_i of $\sigma|i+w$ preserves *FIFO of Reads for the session* σ_i if for every two operations o_1 and o_2 in σ_i such that $o_1 = \text{READ}$, $o_2 = \text{READ}$, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

FIFO of Writes: For a given execution σ and a process p_i, a serialization set $S = \{S_j\}$ preserves *FIFO of Writes for the session* σ_i if for every two operations o_1 and o_2 in σ_i such that $o_1 = \text{WRITE}$, $o_2 = \text{WRITE}$, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $\forall p_j\ o_1 \xrightarrow{S_j} o_2$.

Reads Before Writes: For a given execution σ and a process p_i, a valid serialization S_i of $\sigma|i+w$ preserves *Reads Before Writes for the session* σ_i if for every two operations o_1 and o_2 in σ_i such that $o_1 = \text{READ}$, $o_2 = \text{WRITE}$, and $o_1 \xrightarrow{\sigma_i} o_2$, holds $o_1 \xrightarrow{S_i} o_2$.

Session Causality:[3] For this definition we assume that no value is written more than once to the same variable. For a given execution σ and a process p_i, a serialization set $S = \{S_j\}$ preserves *Session Causality for the session* σ_i if for every three operations o_1, o_2 and o_3 such that o_2 and o_3 are in σ_i, $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, $o_3 = \text{WRITE}$, o_2 read a result written by o_1 and $o_2 \xrightarrow{\sigma_i} o_3$, holds $\forall p_j\ o_1 \xrightarrow{S_j} o_3$.

[3] Called *Writes Follow Reads* in [21]

As noticed in [21], while Read Your Writes, FIFO of Reads and Reads Before Writes only affect the sessions for which they are provided, Session Causality and FIFO of Writes contain guarantees w.r.t. the executions of other processes. Accordingly, we define the former conditions for a single serialization and the latter conditions for a serialization set. However, the following definitions require the same form for all the conditions. Therefore, we assume below that Read Your Writes, FIFO of Reads and Reads Before Writes are defined for a serialization set in which only a single serialization is used in the definition (the definitions in this latter form can be found in the full version of this paper).

For any condition X of these five session properties and a given execution σ, we say that a serialization set $S = \{S_j\}$ *globally preserves* X if it preserves X for all the sessions $\sigma_i \in \sigma$.

We now introduce a definition of the *Total Order* condition:

Total Order: For a given execution σ, a serialization set $S = \{S_j\}$ globally preserves *Total Order* if for every two serializations S_i and S_j in S, $S_i|w = S_j|w$.

For a given execution σ, a serialization set $S = \{S_j\}$ globally preserves some set of the conditions defined above if S globally preserves each condition in this set. Finally, we say that an execution σ is consistent with respect to a condition set (or a single condition) X if there exists a serialization set S of σ such that S globally preserves X. We say that an implementation A *obeys a condition set* (or a single consistency condition) X if every execution generated by A is consistent with respect to X.

3.2 Examples of Known Consistency Conditions

The following is a list of several important and well known consistency conditions:

Sequential Consistency (SC) [12]: An execution σ is sequentially consistent if there exists a legal serialization S of σ such that for each process p_i, $\sigma|i = S|i$.

PRAM Consistency [13]: An execution σ is PRAM consistent if for every process p_i there exists a legal serialization S_i of $\sigma|i + w$ such that if o_1 and o_2 are two operations in $\sigma|i + w$ and $o_1 \xrightarrow{\sigma} o_2$, then $o_1 \xrightarrow{S_i} o_2$.
Note that instead of requiring a legal serialization S_i for every process p_i this definition can be rephrased to require an existence of a serialization set. We will use this latter form in order to define conjunction of PRAM consistency with other consistency conditions, e.g., in the theorems below. This latter form also appears in the full version of this paper.

Causal Consistency [2]: For the definition of causal consistency we assume that no value is written more than once to the same variable. Given an execution σ, an operation o_1 *directly precedes* o_2 (denoted $o_1 \xrightarrow{\sigma} o_2$) if either $o_1 \xrightarrow{\sigma} o_2$ or $o_1 = \text{WRITE}$, $o_2 = \text{READ}$, and o_2 read a result written by o_1. Let $\xrightarrow{*}$ denote the transitive closure of $\xrightarrow{\sigma}$.

An execution σ is causally consistent if for every process p_i there exists a legal serialization S_i of $\sigma|i + w$ such that S_i respects $\xrightarrow{*}$, i.e., if o_1 and o_2 are two operations in $\sigma|i + w$ and $o_1 \xrightarrow{*} o_2$, then $o_1 \xrightarrow{S_i} o_2$.

3.3 Discussion

Most consistency implementations preserve Reads Before Writes mainly because in most implementation a READ operation is blocking and execution is resumed only after a result is returned. We bring this condition here, however, for completeness and because it plays an important role in dependencies between consistency conditions. In the future, we intend to investigate the implications for the systems in which this condition does not hold.

Any single condition that relates two events of the same type is trivial by itself. For example, if we only require FIFO of Reads, then naturally we can always find legal serializations in which all reads are ordered in FIFO order. This is because we have not placed any requirements on writes, and thus we have the freedom to order the writes in the serialization so all the reads are legal. This applies similarly also to FIFO of Writes, Session Causality and Total Order. Thus, these guarantees become meaningful only in combinations that contain several guarantees of different types. The only guarantee that is not trivial by itself is Read Your Writes.

We now present several theorems that show how some combinations of the basic consistency conditions relate to each other and to other known consistency conditions. The proofs of these theorems can be found in the full version of this paper.

Theorem 1. *Any execution that is consistent w.r.t. Total Order and Reads Before Writes is also consistent w.r.t. Session Causality* [4].

Conclusion: Since Reads Before Writes holds in almost all implementations, the practical meaning of this theorem is that Total Order implies Session Causality.

Theorem 2. *Any execution that is consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes is also PRAM consistent. Vice versa, any PRAM consistent execution is also consistent w.r.t. FIFO of Writes, FIFO of Reads, Read Your Writes and Reads Before Writes.*

Theorem 3. *Any execution that is PRAM consistent and is consistent w.r.t. Session Causality is also causally consistent. Vice versa, any causally consistent execution is also consistent w.r.t. Session Causality and PRAM.*

Theorem 4. *Any execution that is PRAM consistent and is consistent w.r.t. Total Order is also sequentially consistent* [5]. *Vice versa, any sequentially consistent execution is also consistent w.r.t. Total Order and PRAM.*

[4] Note that being consistent w.r.t. a set of properties is a stronger property than just being consistent w.r.t. each property in the set.

[5] This claim can also be derived from the results of [18] whose focus, however, is different from ours.

4 Implementation of Consistency Conditions in CASCADE

The general implementation of CASCADE has been presented in [7], but without specific details about the support for the basic consistency conditions that were presented in Section 3. We start this section by covering general elements of CASCADE architecture that are needed to provide the right context for describing consistency implementation. Then, we explain in detail how each individual consistency condition is implemented in CASCADE.

However, due to lack of space, we do not present here any pseudo code or proofs that our implementation obeys a given combination of basic consistency conditions; these appear in the full version of the paper. Furthermore, we do not discuss Reads Before Writes in this section: As explained in Section 3.3, Reads Before Writes trivially holds in any natural implementation.

4.1 Hierarchical Caching in CASCADE

A detailed description of CASCADE architecture can be found in [7]. Here we only briefly summarize the design choices that are important for consistency implementation: The service is provided by a number of servers each of which is responsible for a specific *logical* domain. In practice, these domains can correspond to geographical areas. We call these servers *Domain Caching Servers (DCSs)*.

Cached copies of each object are organized into a hierarchy. A separate hierarchy is constructed for each object. The construction mechanism ensures that for each client, client's local DCS (i.e., the DCS responsible for the client's domain) obtains a copy of the object. In addition, this mechanism attempts to guarantee that the object copy is obtained from the nearest DCS having a copy of this object. Once the local DCS has an object copy, client requests for object method invocation normally go to this DCS, so that the client does not have to communicate to a far server. Only if the local DCS becomes overloaded or unavailable, the client can decide to switch to another DCS. While we count for such a possibility in CASCADE, we consider it an unlikely event. Therefore, our implementation is optimized for the case when the client communicates with a small number of DCSs during its execution (see Section 4.4).

4.2 Implementation of Eventual Propagation and Total Order

CASCADE always guarantees Eventual Update Propagation while the use of other conditions can be controlled by the application. To guarantee Eventual Update Propagation, queries are always locally executed at the DCS a client communicates to and updates are propagated through the hierarchy. However, the way updates propagate and the order in which they are being applied depend on whether Total Order is required.

If Total Order is not required by the application, Eventual Propagation is implemented as follows: A DCS that receives an update request from a client

applies it locally and sends it to all its neighbors in the hierarchy in parallel. A DCS that receives an update request from a neighbor DCS X applies the update and performs flooding, i.e., sends the request to all its neighbors but X. Note that this propagation protocol preserves per-DCS FIFO of updates because all the links are FIFO (as specified in Section 4.1) and because there is only one path in the hierarchy between any pair of nodes. Furthermore, per-DCS Session Causality also holds: If a DCS receives and applies an update, and then some client queries the object state and issues another update at this DCS, then the second update will be broadcast to the neighbors of this DCS after the first one. We will show later in this section how these facts can be exploited in order to provide an efficient implementation of session guarantees.

The Totally Ordered Eventual Propagation (i.e., the Total Order + Eventual Propagation conditions) is implemented as follows: Updates first ascend through the hierarchy towards the root. The root of the hierarchy orders the updates in a sequence, applies them and propagates ordered updates through the hierarchy downwards towards the leaves.

Note that this implementation of Total Order is not affected by presence or absence of application demand for other consistency conditions. Moreover, this implementation is entirely based on the DCS algorithm and inter-DCS protocol, and does not require any client involvement.

Since our goal is to address Internet applications, where extremely long delays are common, we have made the design choice that update requests can return before the update has traversed the entire object hierarchy. The result, however, is that the implementation of session conditions requires client cooperation in most cases.

Also, the implementation of the session conditions adapts itself to the set of consistency requirements chosen by the application. In particular, their implementation is significantly affected by presence or absence of Total Order. Therefore, we discuss their implementation with and without Total Order separately.

4.3 Implementing Session Guarantees in Presence of Total Order

The implementation of the session guarantees is greatly simplified by the presence of the Total Order implementation. First, Session Causality is achieved for free, as Theorem 1 implies. Second, the root of the hierarchy can assign each update a global *update identifier* that serves as a version number of the object. Hence, an object version can be identified by a single number. As a result, the implementation of session guarantees becomes simpler, less information needs to be stored at both clients and DCSs, and most important, less consistency related data needs to be transferred between a client and a DCS per method invocation.

Specifically, with each query result, a DCS returns to the client the number of the object version this query sees. It would be more complicated to handle updates in a similar way because updates have to be propagated first to the root DCS which assigns them an update identifier. In principle, a DCS that received an update request from a client can block the client until the update identifier

is received from the root DCS and then pass this version number to the client. This way, the only consistency information to be transferred between a client and a DCS would be a single global version number. However, since CASCADE is intended to operate in a WAN environment and the propagation latency between a client DCS and a root DCS may be quite significant, this solution may block the client for prohibitively long.

Therefore, CASCADE adopts an alternative identifier scheme for updates: Each DCS maintains a counter of updates originated at this DCS and each update is assigned a local update identifier consisting of the DCS identifier and a counter value. In contrast to the global version numbers, two local update identifiers assigned by different DCSs are incomparable. When a client invokes an update request on a DCS, the DCS immediately produces a new local update identifier and returns it to the client.

For implementation of some session guarantees we need to maintain a version vector for an object with one entry per DCS in the hierarchy; each entry in this vector corresponds to the last local update identifier received from the corresponding DCS. Version vectors are maintained in the following way: When an update ascends through the hierarchy towards the root, the local update identifier is piggybacked on the update message. When this update is propagated from the root towards the leaves, its global version number and local update identifier are both piggybacked. Upon receiving and applying this update, a DCS updates its current object version number and version vector.

We now describe the individual implementation of the three session guarantees that require a non-trivial implementation:

FIFO of Reads: As previously explained, with each query result, a DCS returns to the client the object's version number that this query sees. The client passes this number to a (possibly different) DCS upon its next query. This DCS does not apply the query and blocks the client until it receives and applies the update referred to by the version number (in other words, the DCS *synchronizes* the query with the version number).

FIFO of Writes: For implementing FIFO of Writes, the root DCS should maintain a version vector which contains the last local update identifier received from each DCS. Keeping only the last update identifier is sufficient because Total Order preserves per-DCS FIFO of updates: Two updates issued at the same DCS reach the root where they are ordered in order of their issuance.

As previously explained, when a client invokes an update request on a DCS, the DCS transfers a local update identifier back to the client. The client only remembers the last local identifier it received from some DCS and forgets all previous local identifiers. This is sufficient because FIFO of Writes is a transitive relation and it is enough to remember only the last predecessor.

The client passes the last known local identifier to a (possibly different) DCS upon the invocation of its next update request. The DCS piggybacks this identifier on the update message that traverses the hierarchy towards the root. The root DCS compares this identifier against the version vector and

blocks the message until the update referred to by the identifier is received and applied.

Read Your Writes: For implementing Read Your Writes, each DCS maintains a version vector. When a client invokes a query request on a DCS, it passes the local update identifier(s) of the last update(s) it initiated. The DCS synchronizes the query with these identifiers based on the information stored in its version vector.

If Read Your Writes is provided along with FIFO of Writes, one last local update identifier is sufficient to be synchronized with because FIFO is a transitive relation. Otherwise, for each DCS the client sent an update request to, it should remember the last local update identifier received from this DCS. In this case, the query must be synchronized with the entire set of identifiers. However, since we assume in the model that a client only communicates with a small subset of all existing DCSs in the object hierarchy, the set of identifiers is also small and its transfer between a client and a DCS is not an expensive operation.

If Read Your Writes is provided along with FIFO of Reads, the amount of information to transfer and store at a client can be optimized in a different way: The client should only remember the local update identifiers it received since the last query. If the client first issues several updates and then two queries, the first query will be synchronized with the updates and the second query will be synchronized with the first one. Therefore, no explicit synchronization of the second query with the updates is necessary in this case.

In summary, if Total Order is provided, the implementation of the session guarantees introduces an insignificant extra overhead: The amount of consistency information that needs to be stored at clients and transferred between clients and DCSs is small and does not depend on the number of clients and DCSs in the system.

4.4 Implementing Session Guarantees Without Total Order

When the Total Order implementation is not employed, an object does not have a single version number. In this case its state can only be characterized by the version vector that has to be maintained by each DCS. While this does not affect the implementation of Read Your Writes and the implementation of FIFO of Writes remains almost as simple as in the case of Total Order, the implementation of FIFO of Reads becomes more complicated and expensive. In addition, an implementation of Session Causality should now be provided. We elaborate on the changes in the implementations below:

FIFO of Writes: As with Total Order, a DCS returns a local update identifier to the client that initiates the update, a client remembers only the last local identifier and forgets the previous one, and this local identifier is transferred to a DCS upon the next update request. The only change is that now the

DCS blocks the client and does not assign the update request a local identifier until it receives the referred update. If the DCS immediately produced a local update identifier, released the client and left the update request in a pending state, then all later (unrelated) update requests with higher local update identifiers would have to wait until this update would be applied. This is a shortcoming of the version vector method which assumes that updates originated at the same DCS are applied in the order of their local identifiers.
An appealing alternative to blocking the client is to use a version vector of sliding windows instead of just a vector of update identifiers. In this solution updates can sometimes be applied in an order different from that of their local identifiers. However, a DCS has to remember the identifiers of the updates applied out of order. Therefore, while eliminating unnecessary delays, this solution requires more space and more complicated version management. Moreover, this solution makes the implementation of FIFO of Reads complicated and inefficient.

FIFO of Reads: Without Total Order, the simplest implementation of this condition is that a DCS transfers the entire version vector to a client along with the results of a query. The client remembers the version vector it received the last time and forgets the previous vector. This vector is passed to a (possibly different) DCS upon the next client query, and the DCS synchronizes the query with each local identifier in the vector.
This implementation is inefficient because the entire version vector whose length is the number of DCSs in the object hierarchy is sent twice per each query. Below we introduce optimizations that allows us to reduce the average amount of transferred information.

Session Causality: Again, the simplest implementation is that a DCS transfers the entire version vector to a client along with the query results. However, unless FIFO of Reads is also provided, it is not sufficient that a client remembers only the version vector it received in the previous interaction with the DCS. Actually, the client must merge all the vectors it received during the execution by computing their maximum. This merged vector is passed to a DCS upon the next client update. Furthermore, since every DCS has to synchronize this update with this vector, the DCS piggybacks the entire vector on the update message sent to other DCSs. In the future, we intend to investigate the possibility of using the *causal separators* technique [19] in order to reduce the amount of piggybacked information. This technique appears especially appealing due to the hierarchical architecture employed by CASCADE in which each intermediate node can act as a causal separator.

As we see, when Total Order is not employed, the straightforward implementations of FIFO of Reads and Session Causality are quite expensive in terms of the amount of information to be transferred over the network. Fortunately, the implementation of FIFO of Reads can be significantly improved by using the optimization that is explained below.

Efficient FIFO of Reads Implementation First, rather than sending the entire version vector to a client as part of the response to queries, a DCS can send the difference between its current version vector and the vector received from the client for the purpose of synchronization [6]. This difference is usually shorter than the entire version vector. For example, the difference of $\langle\langle A,1\rangle,\langle B,3\rangle\rangle$ and $\langle\langle A,1\rangle,\langle B,1\rangle\rangle$ is $\langle\langle B,3\rangle\rangle$. Upon receiving such a vector difference the client can add it to the vector it sent and restore the entire version vector of the DCS in its local memory. However, the client should still send its entire version vector for synchronization.

Another optimization is based on the following observation: If a client does not switch DCSs (in other words, it invokes all updates and queries on the same DCS), then FIFO of Reads always holds in a trivial way and does not need to be implemented at all. Furthermore, FIFO of Writes and Session Causality also trivially hold due to per-DCS FIFO of Writes and per-DCS Session Causality, respectively. This situation is summarized in Table 1 that clearly shows the cost of client mobility.

Table 1. The implementation cost of session guarantees

Session Guarantee	with TO		w/o TO	
	Mobile Clients	Not	Mobile Clients	Not
FIFO of Reads	×	√	××	√
Session Causality	√	√	××	√
FIFO of Writes	×	√	×	√
Read Your Writes	×	×	×	×

√ – trivially holds
× – adds extra cost
×× – requires costly communication

Unfortunately, if the consistency implementation is unaware that the client continues to work with the same DCS, it transfers the same high amount of information as if the client switched DCSs. This observation calls for optimizing the implementation for the most usual and frequent case when a client communicates with a single DCS. The client can just verify that it invokes a current request on the same DCS as the previous one. If this is true, the client does not need to send any information for synchronization.

However, a DCS still has to return its version vector along with the query results in order to account for the possibility that a client invokes the next query on another DCS. Furthermore, if a client sends no synchronization information, we can no longer use the differential optimization described above because a DCS has no reference point to compute the difference of vectors.

Thus, there is a need for synchronization information shorter than just an entire version vector. To this end, we introduce a notion of *local DCS history* which is a numbered sequence of update identifiers of all the updates applied at the DCS during the execution. A *local history pointer* is just an index to local

[6] This optimization is similar to Singhal-Kshemkalyani technique [20] for implementing vector clocks.

DCS history. A DCS can return this pointer to a client, and a client can transfer it back to the DCS for synchronization at some later point. As a result, only local history pointers and vector differences are transferred over the network instead of entire version vectors.

As part of this optimization, a DCS should be able to compute the difference between its current version vector and a pointer to some past point of its local history. An important question is how this can be done efficiently without keeping the whole local history. The full version of this paper provides a detailed explanation of the algorithm used in CASCADE that satisfies these requirements. It also describes a generalization of this optimization for the case when a client communicates with several DCSs. This generalization proves to be efficient when the number of DCSs is small (which is the usual case as noted in Section 4.1).

5 Future Work

It would be interesting to arrive at a complete set of basic consistency conditions. That is, be able to show that any consistency condition can be provided as a combination of a subset of these conditions, and that each of this conditions is necessary for implementing at least one consistency condition. In our opinion, this should be made at the application point of view, like our definitions and the works of [2,4], since such definitions are more rigorous, easier to understand, and can be used more easily by programmers to prove correctness of their applications.

As for the implementation, it is possible to implement each of the basic consistency conditions separately, and then trigger the required ones based on the application's need. We have decided not to follow this path, and to optimize the implementation of various conditions based on the other conditions being provided, since an independent implementation of each condition was too wasteful and slow. Perhaps the right way to tackle this issue is by providing an independent implementation for each condition, and then use a high-level compiler to optimize combinations of conditions, similar to the work on automatically optimizing and proving group communication protocol stacks in Ensemble [15].

References

1. M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *Proc. of the 5th ACM Symposium On Parallel Algorithms and Architectures*, pages 251–260, June/July 1993.
2. M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1), 93.
3. H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. *SIAM Journal of Computing*, 27(1), February 1998.
4. H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

5. B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Intl. Computer Conf. (COMPCON)*, pages 528–537, February 1993.
6. J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Rel ease Consistency*. PhD thesis, Computer Science Dpt., Rice University, 1993.
7. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proceedings of Middleware '00*, pages 1–23, April 2000. The Best Conference Paper award.
8. R. Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, Department of Computer Science, The Technion, 1994.
9. M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
10. P. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. Technical Report TR GIT-ICS-89/39, Georgia Institute of Technology, October 1989.
11. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, December 1994.
12. L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.
13. R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Computer Science Dpt., Princeton University, September 1988.
14. B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD Intl. Symp. on Management of Data*, pages 318–329, June 1996.
15. X. Liu, C. Keitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *the 17th Symp. on Operating Systems Principles*, December 1999.
16. M. Mizuno, M. Raynal, and J. Zhou. Sequential Consistency in Distributed Systems. In *Proceedings of the Intl Workshop "Theory and Practice in Distributed Systems"*, pages 224–241, September 1994.
17. OMG. *The Common Object Request Broker: Architecture and Specification*. 1995.
18. M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. In *the 15th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 180–194, December 1995.
19. L. Rodrigues and P. Verissimo. Causal Separators for Large-Scale Multicast Communication. In *Proceedings of the 15th IEEE Intl. Conference on Distributed Computing Systems*, pages 83–91, June 1995.
20. M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47–52, August 1992.
21. D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welsh. Session Guarantees for Weakly Consistent Replicated Data. In *IEEE Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, 1994.
22. M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurency*, 7(1):70–78, January-March 1999.

Author Index