Shlomi Dolev (Ed.)

LNCS 4167

Distributed Computing

20th International Symposium, DISC 2006
Stockholm, Sweden, September 2006
Proceedings

DISC 2006 • Stockholm, Sweden • September 18-20 •

Springer

Lecture Notes in Computer Science 4167

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

Shlomi Dolev (Ed.)

Distributed Computing

20th International Symposium, DISC 2006
Stockholm, Sweden, September 18-20, 2006
Proceedings

Springer

Volume Editor

Shlomi Dolev
Ben-Gurion University of the Negev
Department of Computer Science
Beer-Sheva, 84105, Israel
E-mail: dolev@cs.bgu.ac.il

Library of Congress Control Number: 2006932337

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1.1, D.4.4-5

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-44624-9 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-44624-8 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11864219 06/3142 5 4 3 2 1 0

Preface

DISC, the International Symposium on DIStributed Computing, is an annual forum for presentation of research on all facets of distributed computing, including the theory, design, analysis, implementation, and application of distributed systems and networks. The 20th anniversary edition of DISC was held on September 18-20, 2006, in Stockholm, Sweden.

There were 145 extended abstracts submitted to DISC this year, and this volume contains the 35 contributions selected by the Program Committee and one invited paper among these 145 submissions. All submitted papers were read and evaluated by at least three Program Committee members, assisted by external reviewers. The final decision regarding every paper was taken during the Program Committee meeting, which took place in Beer-Sheva, June 30 and July 1, 2006.

The Best Student Award was split and given to two papers: the paper "Exact Distance Labelings Yield Additive-Stretch Compact Routing Schemes," by Arthur Bradly, and Lenore Cowen, and the paper "A Fast Distributed Approximation Algorithm for Minimum Spanning Trees" co-authored by Maleq Khan and Gopal Pandurangan.

The proceedings also include 13 three-page-long brief announcements (BA). These BAs are presentations of ongoing works for which full papers are not ready yet, or of recent results whose full description will soon be or has been recently presented in other conferences. Researchers use the BA track to quickly draw the attention of the community to their experiences, insights and results from ongoing distributed computing research and projects. The BAs included in this proceedings volume were selected among 26 BA submissions.

DISC 2006 was organized in cooperation with the European Association for Theoretical Computer Science (EATCS), the European Research Consortium for Informatics and Mathematics (ERCIM), and Swedish Institute of Computer Science (SICS). The support of Ben-Gurion University, Microsoft Research, Intel, Sun microsystems, Deutsche Telekom Laboratories is also gratefully acknowledged.

July 2006 Shlomi Dolev

Organization

DISC, the International Symposium on DIStributed Computing, is an annual forum for research presentations on all facets of distributed computing. The symposium was called the International Workshop on Distributed Algorithms (WDAG) from 1985 to 1997. DISC 2006 was organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Steering Committee

Hagit Attiya	Technion
Shlomi Dolev	BGU
Pierre Fraigniaud	Université Paris Sud
Rachid Guerraoui	EPFL
Alexander Shvartsman	UCONN, Chair
Paul Vitanyi	CWI, Vice-Chair
Roger Wattenhofer	ETH Zurich

Organization Committee

Conference Chairs	Lenka Carr-Motyckova, LUT, Luleȧ Tekniska Universitet Seif Haridi, SICS, Swedish Institute of Computer Science AB
Program Chair	Shlomi Dolev, Ben-Gurion University of the Negev
20th Anniversary Celebration Chair	Michel Raynal IRISA, Université de Rennes
Web Chair	Heleèn Martìn, SICS, Swedish Institute of Computer Science AB
Finance Chair	Charlotta Jörsäter, SICS, Swedish Institute of Computer Science AB

Program Committee

Lenka Carr-Motyckova	LUT
Shlomi Dolev	BGU, **Program Chair**
Christof Fetzer	Technische Universität Dresden
Tim Harris	Microsoft Research Cambridge
Maurice Herlihy	Brown University

Jaap-Henk Hoepman	RU Nijmegen
Prasad Jayanti	Dartmouth College
Dariusz Kowalski	University of Liverpool
Danny Krizanc	Wesleyan University
Fabian Kuhn	Microsoft Research Silicon Valley
Nancy Lynch	MIT
Anna Lysyanskaya	Brown University
Petros Maniatis	Intel Research Berkeley
Mark Moir	SUN Microsystems Laboratories
Seffi Naor	Microsoft Research and Technion
Marina Papatriantafilou	Chalmers University
Andrzej Pelc	Université du Québec
Michel Raynal	IRISA, Université de Rennes
André Schiper	EPFL
Gadi Taubenfeld	Interdisciplinary Center
Sébastien Tixeuil	Université Paris Sud
Frits Vaandrager	RU Nijmegen

Sponsors

Referees

Ittai Abraham
Yehuda Afek
Marcos Aguilera
James Aspnes
Hagit Attiya
Gildas Avoine
Liskov Barbara
Amotz Bar-Noy
Rida A. Bazzi
Amos Beimel
Fredik Bengtsson
Vartika Bhandari
Andreas Blass
Paolo Boldi
Glencora Borradaile
Anat Bremler-Barr
Olga Brukman
Harry Buhrman
Chi Cao Minh
Bernadette Charron-Bost
Jingsen Chen
Wei Chen
Yan Chenyu
Bogdan Chlebus
Lukasz Chmielewski
Gregory Chockler
Byung-Gon Chun
Mike Dahlin
Xavier Défago
Carole Delporte-Gallet
Feodor Dragan

Michael Elkin
Robert Ennals
Leah Epstein
Jittat Fakcharoenphol
Rui Fan
Hugues Fauconnier
Sasha Fedorova
Eyal Felstaine
Tim Finin
Hen Fitoussi
Pierre Fraigniaud
Nissim Francez
Matt Franklin
Eli Gafni
Juan Garay
Flavio Garcia
Vijay K. Garg
Cyril Gavoille
Lezek Gasieniec
Roland Gemesi
Chryssis Georgiou
Sukumar Ghosh
Andres Gidenstam
Seth Gilbert
Mayer Goldberg
Maria Gradinariu
Michael Greenwald
Rachid Guerraoui
Phuong Ha Hoai
Yinnon Haviv
Danny Hendler
Thomas Herault
Ted Herman
Chien-Chung Huang
Michel Hurfin
Adam Iwanicki
Tomas Johansson
Ronen Kat
Idit Keidar
Alex Kesselman
Ralf Klasing
Geir Koien
Boris Koldehofe
Kishori Konwar
Marina Kopeetsky
Maciej Kurowski
Klaus Kursawe
Shay Kutten
Limor Lahiani
Kevin Lai
Zvi Lotker
Victor Luchangco
Ritesh Madan
Adam Malinowski
Stéphane Messika
Yves Metivier
Maria Meyerovich
Maged Michael
Saya Mitra
Emilia Monakhova
Achour Mostefaoui
Mikhail Nesterenko
Calvin Newport
Tina Nolte
Boaz Patt-Shamir
Fernando Pedone
David Peleg
Franck Petit
Kaustubh Phanse
Laurence Pilard
Benny Pinkas
Sara Porat
Guiseppe Prencipe
Rami Puzis
Tomasz Radzik
Sylvia Ratnasamy
Rodrigo Rodrigues
Mariusz Rokicki
Christian Scheideler
Elad Michael Schiller
Roberto Segala
Ori Shalev
Nir Shavit
Abhi Shelat
Alex Shvartsman
Radu Siminiceanu
Thanh Son
Thanh Son Nguyen
Paul Spirakis
Scott Stoller
Michal Strojnowski
Ram Swaminathan
Boleslaw K. Szymanski
Nesime Tatbul
Philippas Tsigas
Nir Tzachar
Shinya Umeno
Eli Upfal
Sebastiano Vigna
Jennifer Walter
Michael Warres
Mike Warres
Jennifer Welch
Yang Xiang
Reuven Yagel
Praveen Yalagandula
Piotr Zielinski
Michele Zito
Uri Zwick

Table of Contents

Invited Talks

Exploring Gafni's Reduction Land: From Ω^k to Wait-Free Adaptive $(2p - \lceil \frac{p}{k} \rceil)$-Renaming Via k-Set Agreement

Achour Mostefaoui, Michel Raynal, and Corentin Travers

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{achour, raynal, ctravers}@irisa.fr

Abstract. The adaptive renaming problem consists in designing an algorithm that allows p processes (in a set of n processes) to obtain new names despite asynchrony and process crashes, in such a way that the size of the new renaming space M be as small as possible. It has been shown that $M = 2p-1$ is a lower bound for that problem in asynchronous atomic read/write register systems.

This paper is an attempt to circumvent that lower bound. To that end, considering first that the system is provided with a k-set object, the paper presents a surprisingly simple adaptive M-renaming wait-free algorithm where $M = 2p - \lceil \frac{p}{k} \rceil$. To attain this goal, the paper visits what we call Gafni's reduction land, namely, a set of reductions from one object to another object as advocated and investigated by Gafni. Then, the paper shows how a k-set object can be implemented from a leader oracle (failure detector) of a class denoted Ω^k. To our knowledge, this is the first time that the failure detector approach is investigated to circumvent the $M = 2p-1$ lower bound associated with the adaptive renaming problem. In that sense, the paper establishes a connection between renaming and failure detectors.

1 Introduction

The renaming problem The *renaming* problem is a coordination problem initially introduced in the context of asynchronous message-passing systems prone to process crashes [3]. Informally, it consists in the following. Each of the n processes that define the system has an initial name taken from a very large domain $[1..N]$ (usually, $n << N$). Initially, a process knows only its name, the value n, and the fact that no two processes have the same initial name. The processes have to cooperate to choose new names from a name space $[1..M]$ such that $M << N$ and no two processes obtain the same new name. The problem is then called *M-renaming*.

Let t denote the upper bound on the number of processes that can crash. It has been shown that $t < n/2$ is a necessary and sufficient requirement for solving the renaming problem in an asynchronous message-passing system [3]. That paper presents also a message-passing algorithm whose size of the renaming space is $M = n + t$.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 1–15, 2006.

The problem has then received a lot of attention in the context of asynchronous shared memory systems made up of atomic read/write registers. Numerous wait-free renaming algorithms have been designed (e.g., [2,4,5,6]). *Wait-free* means here that a process that does not crash has to obtain a new name in a finite number of its own computation steps, regardless of the behavior of the other processes (they can be arbitrarily slow or even crash) [12]. Consequently, wait-free implies $t = n - 1$. An important result in such a context, concerns the lower bound on the new name space. It has been shown in [13] that there is no wait-free renaming algorithm when $M < 2n - 1$. As wait-free $(2n - 1)$-renaming algorithms have been designed, it follows that that $M = 2n - 1$ is a tight lower bound.

The previous discussion implicitly assumes the "worst case" scenario: all the processes participate in the renaming, and some of them crash during the algorithm execution. The net effect of crashes and asynchrony create "noise" that prevents the renaming space to be smaller than $2n - 1$. But it is not always the case that all the processes want to obtain a new name. (A simple example is when some processes crash before requiring a new name.) So, let p, $1 \leq p \leq n$, be the number of processes that actually participate in the renaming. A renaming algorithm guarantees *adaptive* name space if the size of the new name space is a function of p and not of n. Several adaptive wait-free algorithms have been proposed that are optimal as they provide $M = 2p - 1$ (e.g., [2,4,6]).

The question addressed in the paper. Let us assume that we have a solution to the consensus problem. In that case, it easy to design an adaptive renaming algorithm where $M = p$ (the number of participating processes). The solution is as follows. From consensus objects, the processes build a concurrent queue that provides them with two operations: a classical enqueue operation and a read operation that provides its caller with the current content of the queue (without modifying the queue). Such a queue object has a sequential specification and each operation can always be executed (they are *total* operations according to the terminology of [12]), from which it follows that this queue object can be wait-free implemented from atomic registers and consensus objects [12]. Now, a process that wants to obtain a new name does the following: (1) it deposits its initial name in the queue, (2) then reads the content of the queue, and finally (3) takes as its new name its position in the sequence of initial names read from queue. It is easy to see that if p processes participate, they obtain the new names from 1 to p, which means that consensus objects are powerful enough to obtain the smallest possible new name space.

The aim of the paper is to try circumventing the lower bound $M = 2p - 1$ associated with the adaptive wait-free renaming problem, by enriching the underlying read/write register system with appropriate objects. More precisely, given M with $p \leq M \leq 2p - 1$, which objects (when added to a read/write register system) allow designing an M-renaming wait-free algorithm (without allowing designing an $(M - 1)$-renaming algorithm). The previous discussion on consensus objects suggests to investigate k-set agreement objects to attain this goal, and to study the tradeoff relating the value of k with the new renaming space. The *k-set agreement* problem is a distributed coordination problem (k defines the coordination degree it provides the processes with) that generalizes the consensus problem: each process proposes a value, and any process that does not crash must decide a value in such a way that at most k distinct values are decided and any decided value is a proposed value. The smaller the coordination degree k, the more coordination imposed on the participating processes: $k = 1$ is the more constrained version of the problem (it is consensus), while $k = n$ means no coordination at all.

From k-set to $(2p - \lceil \frac{p}{k} \rceil)$*-renaming.* Assuming k-set agreement base objects, and $p \leq n$ participating processes, the paper presents an adaptive wait-free renaming algorithm providing a renaming space whose size is $M = (2p - \lceil \frac{p}{k} \rceil)$. Interestingly, when considering the two extreme cases we have the following: $k = 1$ (consensus) gives $M = p$ (the best that can be attained), while $k = n$ (no additional coordination power) gives $M = 2p - 1$, meeting the lower bound for adaptive renaming in read/write register systems.

The proposed algorithm follows Gafni's reduction style [9]. It is inspired by the adaptive renaming algorithm proposed by Borowsky and Gafni [6]. In addition to k-set objects, it also uses simple variants of base objects introduced in [6,7,10], namely, *strong k-set agreement* [7], *k-participating set* [6,10]. These objects can be incrementally built as follows: (1) base k-set objects are used to build k-participating set objects, and then (2) k-participating set objects, are used to solve $(2p - \lceil \frac{p}{k} \rceil)$-renaming.

The renaming algorithm that we obtain is surprisingly simple. It is based on a very well-known basic strategy: decompose a problem into independent subproblems, solve each subproblem separately, and finally piece together the subproblem results to produce the final result. More precisely, the algorithm proceeds as follows:

- (1) Using a k-participating set object, the processes are partitioned into independent subsets of size at most k.
- (2) In each partition, the processes compete in order to acquire new names from a small name space. Let h be the number of processes that belong to a given partition.They obtain new names in the space $[1..2h - 1]$.
- (3) Finally, the name spaces of all the partitions are concatenated in order to obtain a single name space $[1..M]$.

The key of the algorithm is the way it uses a k-participating set object to partition the p participating processes in such a way that, when the new names allocated in each partition are pieced together, the new name space is upper bounded by $M = (2p - \lceil \frac{p}{k} \rceil)$ Interestingly, the processes that belong to the same partition can use any wait-free adaptive renaming algorithm to obtain new names within their partition (distinct partitions can even use different algorithms). This noteworthy modularity property adds a generic dimension to the proposed algorithm.

From the oracle Ω^k *to k-set objects.* Unfortunately, k-set agreement objects cannot be wait-free implemented from atomic registers [7,13,17]. So, the paper investigates additional equipment the asynchronous read/write register system has to be enriched with in order k-set agreement objects can be implemented. To that aim, the paper investigates a family of leader oracles (denoted here $(\Omega^z)_{1\leq z\leq n}$), and presents a k-set algorithm based on read/write registers and any oracle of such a class Ω^k.

So, the paper provides reductions showing that adaptive wait-free $(2p-\lceil\frac{p}{k}\rceil)$-renaming can be reduced to the Ω^k leader oracle class. To our knowledge, this is the first time that oracles (failure detectors) are proposed and used to circumvent the $2p-1$ adaptive renaming space lower bound. Several problems remain open. The most crucial is the statement of the minimal information on process crashes that are necessary and sufficient for bypassing the lower bound $2p-1$.

Roadmap. The paper is made up of 5 sections. Section 2 presents the asynchronous computation model. Then, Section 3 describes the adaptive renaming algorithm. This algorithm is based on a k-participating set object. Section 4 visits Gafni's reduction land by showing how the k-participating set object can be built from a k-set object. Then, Section 5 describes an algorithm that constructs a k-set object in an asynchronous read/write system equipped with a leader oracle of the class Ω^k.

2 Asynchronous System Model

Process model. The system consists of n processes that we denote $p_1,\ldots,p_n$. The integer i is the index of p_i. Each process p_i has an initial name id_i such that $id_i \in [1..N]$. Moreover, a process does not know the initial names of the other processes; it only knows that no two processes have the same initial name. A process can crash. Given an execution, a process that crashes is said to be *faulty*, otherwise it is *correct* in that execution. Each process progresses at its own speed, which means that the system is asynchronous.

In the following, given a run of an algorithm, p denotes the number of processes that *participate* in that run, $1 \leq p \leq n$. (To participate, a process has to execute at least one operation on a shared object.)

Coordination model. The processes cooperate and communicate through two types of reliable objects: atomic multi-reader/single-write registers, and k-set objects. A k-set object KS provides the processes with a single operation denoted $\mathsf{kset\_propose}_k()$. It is a one-shot object in the sense that each process can invoke $KS.\mathsf{kset\_propose}_k()$ at most once. When a process p_i invokes $KS.\mathsf{kset\_propose}_k(v)$, we say that it "proposes v" to the k-set object KS. If p_i does not crash during that invocation, it obtains a value v' (we then say "p_i decides v'"). A k-set object guarantees the following two properties: a decided value is a proposed value, and no more than k distinct values are decided.

Notation. Identifiers with upper case letters are used to denote shared registers or shared objects. Lower case letters are used to denote local variables; in that case the process index appears as a subscript. As an example, $level_i[j]$ is a local variable of the process p_i, while $LEVEL[j]$ is an atomic register.

3 An Adaptive $(2p - \lceil \frac{p}{k} \rceil)$-Renaming Algorithm

3.1 Non-triviality

Let us observe that the trivial renaming algorithm where p_i takes i as its new name is not adaptive, as the renaming space would always be $[1..m]$, where m is the greatest index of a participating process (as an example consider the case where only p_1 and p_n are participating). To rule out this type of ineffective solution, we consider the following requirement for a renaming algorithm [5]:

- The code executed by process p_i with initial name id is exactly the same as the code executed by process p_j with initial name id.

This constraint imposes a form of anonymity with respect to the process initial names. It also means that there is a strong distinction between the index i associated with p_i and its original name id_i. The initial name id_i can be seen as a particular value defined in p_i's initial context. Differently, the index i can be seen as a pointer to the atomic registers that can be written only by p_i. This means that the indexes define the underlying "communication infrastructure".

3.2 k-Participating Set Object

The renaming algorithm is based on a *k-participating set* object. Such an object generalizes the *participating set* object defined in [6].

Definition. A k-participating set object PS is a one-shot object that provides the processes with a single operation denoted $\mathsf{participating\_set}_k()$. A process p_i invokes that operation with its name id_i as a parameter. The invocation $PS.\mathsf{participating\_set}_k(id_i)$ returns a set S_i to p_i (if p_i does not crash while executing that operation). The semantics of the object is defined by the following properties [6,10]:

- Self-membership: $\forall i$: $id_i \in S_i$.
- Comparability: $\forall i, j$: $S_i \subseteq S_j \ \vee \ S_j \subseteq S_i$.
- Immediacy: $\forall i, j$: $(id_i \in S_j) \Rightarrow (S_i \subseteq S_j)$.
- Bounded simultaneity: $\forall \ell : \ 1 \leq \ell \leq n$: $|\{j \ : \ |S_j| = \ell\}| \leq k$.

The set S_i obtained by a process p_i can be seen as the set of processes that, from its point of view, have accessed or are accessing the k-participating set object. A process always sees itself (self-membership). Moreover, such an object guarantees that the S_i sets returned to the process invocations can be totally ordered by inclusion (comparability). Additionally, this total order is not at all arbitrary: it ensures that, if p_j sees p_i (i.e., $id_i \in S_j$) it also sees all the processes seen by p_i (Immediacy). As a consequence if $id_i \in S_j$ and $id_j \in S_i$, we have $S_i = S_j$. Finally, the object guarantees that no more than k processes see the same set of processes (Bounded simultaneity). As we will see later (Section 3.2), such an object can be constructed from k-set objects.

Table 1. An example of k-participating object ($p = 10 \leq n$, $k = 3$)

level	stopped processes	S_i sets
10	p_5,p_9	$S_5 = S_9 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$
9		empty level
8	p_1, p_3, p_{10}	$S_1 = S_3 = S_{10} = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_{10}\}$
7		empty level
6		empty level
5	p_2, p_8	$S_2 = S_8 = \{p_2, p_4, p_6, p_7, p_8\}$
4		empty level
3	p_7	$S_7 = \{p_4, p_6, p_7\}$
2	p_4, p_6	$S_4 = S_6 = \{p_4, p_6\}$
1		empty level

Notation and properties. Let S_j be the set returned to p_j after it has invoked participating_set$_k(id_j)$, and $\ell = |S_j|$ (notice that $1 \leq \ell \leq n$). The integer ℓ is called the *level* of p_j, and we say "p_j is -or stopped- at *level* ℓ". If there is a process p_j such that $|S_j| = \ell$, we say "the level ℓ is not empty", otherwise we say "the level ℓ is empty". Let $\mathcal{L}$ be the set of non-empty levels ℓ, $|\mathcal{L}| = m \leq n$. Let us order the m levels of $\mathcal{L}$ according to their values, i.e., $\ell_1 < \ell_2 < \cdots < \ell_m$ (this means that the levels in $\{1, \ldots, n\} \setminus \{\ell_1, \ldots, \ell_m\}$ are empty).

$|S_j| = \ell$ (p_j stopped at level ℓ) means that, from p_j point of view, there are exactly ℓ processes that (if they do not crash) stop at the levels ℓ' such that $1 \leq \ell' \leq \ell$. Moreover, these processes are the processes that define S_j. (It is possible that some of them have crashed before stopping at a level, but this fact cannot be known by p_j.) We have the following properties:

- If p processes invoke participating_set$_k()$, no process stops at a level higher than p.
- $(|S_i| = |S_j| = \ell) \Rightarrow (Si = S_j)$ (from the comparability property).
- Let S_i and S_j such that $|S_i| = \ell_x$ and $|S_j| = \ell_y$ with $\ell_x < \ell_y$.
  - $S_i \subset S_j$ (from $\ell_x < \ell_y$, and the comparability property).
  - $|S_j \setminus S_i| = |S_j| - |S_i| = \ell_y - \ell_x$ (consequence of the set inclusion $S_i \subset S_j$).

A k-participating set object can be seen as "spreading" the $p \leq n$ participating processes on at most p levels ℓ. This spreading is such that (1) there are at most k processes per level, and (2) each process has a consistent view of the spreading (where "consistent" is defined by the self-membership, comparability and immediacy properties). As an example, let us consider Table 1 that depicts the sets S_i returned to $p = 10$ processes participating in a k-participating set object (with $k = 3$), in a failure-free run. As we can see some levels are empty. Two processes, p_2 and p_8, stopped at level 5; their sets are equal and contain exactly five processes, namely the processes that stopped at a level ≤ 5.

The next lemma captures an important property provided by a k-participating set object. Let $ST[\ell_x] = \{j \text{ such that } |S_j| = \ell_x\}$ (the processes that have stopped at the level ℓ_x). For consistency purpose, let $\ell_0 = 0$.

Lemma 1. $|ST[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$.

Proof. $|ST[\ell_x]| \leq k$ follows immediately from the bounded simultaneity property. To show $|ST[\ell_x]| \leq \ell_x - \ell_{x-1}$, let us consider two processes p_j and p_i such that p_j stops at the level ℓ_x while p_i stops at the level ℓ_{x-1}. We have:

1. $|S_j| = \ell_x$ and $|S_i| = \ell_{x-1}$ (definition of "a process stops at a level").
2. $ST[\ell_x] \subseteq S_j$ (from the self-membership and comparability properties),
3. $ST[\ell_x] \cap S_i = \emptyset$ (from $S_j \neq S_i$ and the immediacy and self-membership properties),
4. $ST[\ell_x] \subseteq S_j \setminus S_i$ (from the items 2 and 3),
5. $|S_j \setminus S_i| = \ell_x - \ell_{x-1}$ (previous discussion),
6. $|ST[\ell_x]| \leq \ell_x - \ell_{x-1}$ (from the items 4 and 5). $\Box_{Lemma\ 1}$

Considering again Table 1, let us assume that the processes p_1, p_3 and p_{10} have crashed while they are at level $\ell = 8$, and before determining their sets S_1, S_3 and S_{10}. The level $\ell = 8$ is now empty (as no process stops at that level), and the levels 10 and 5 are now consecutive non-empty levels. We have then $ST[10] = \{p_5, p_9\}$, $ST[5] = \{p_2, p_8\}$, and $|ST[10]| = 2 \leq \min(k, 10 - 5)$.

3.3 An Adaptive Renaming Protocol

The adaptive renaming algorithm is described in Figure 1. When a process p_i wants to acquire a new name, it invokes $\mathsf{new\_name}(id_i)$. It then obtains a new name when it executes line 05. Remind that p denotes the number of processes that participate in the algorithm.

Base objects. The algorithm uses a k-participating set object denoted PS, and a size n array of adaptive renaming objects, denoted $RN[1..n]$.

Each base renaming object $RN[y]$ can be accessed by at most k processes. It provides them with an operation denoted $\mathsf{rename}()$. When accessed by $h \leq k$ processes, it allows them to acquire new names within the renaming space $[1..2h-1]$. Interestingly, such adaptive wait-free renaming objects can be built from atomic registers (e.g., [2,4,6]). As noticed in the introduction, this feature provides the proposed algorithm with a modularity dimension as $RN[y]$ and $RN[y']$ can be implemented differently.

The algorithm: principles and description. The algorithm is based on the following (well-known) principle.

- Part 1. Divide for conquer.
  A process p_i first invokes $PS.\mathsf{participating\_set}_k(id_i)$ to obtain a set S_i satisfying the self-membership, comparability, immediacy and bounded simultaneity properties (line 01). It follows from these properties that (1) at most k processes obtain the same set S (and consequently belong to the same partition), and (2) there are at most p distinct partitions.

  An easy and unambiguous way to identify the partition p_i belongs to is to consider the level at which p_i stopped in the k-participating set object, namely, the level $\ell = |S_i|$. The $h \leq k$ processes in the partition $\ell = |S_i|$ compete then among themselves to acquire a new name. This is done by p_i invoking the appropriate renaming object, i.e., $RN\big[|S_i|\big].\mathsf{rename}(id_i)$ (line 03). As indicated before, these processes obtain new names in renaming space $[1..2h-1]$.

```
operation new_name($id_i$):
(1)        $S_i \leftarrow PS.\mathsf{participating\_set}_k(id_i)$;
(2)        $base_i \leftarrow (2 \times |S_i| - \lceil \frac{|S_i|}{k} \rceil)$;
(3)        $offset_i \leftarrow RN\big[|S_i|\big].\mathsf{rename}(id_i)$;
(4)        $myname_i \leftarrow base_i - offset_i + 1$;
(5)        return($myname_i$)
```

Fig. 1. Generic adaptive renaming algorithm (code for p_i)

- Part 2. Piece together the results of the subproblems.
  The final name assignment is done according to very classical *(base,offset)* rule. A base is attributed to each partition as follows. The partition $\ell = |S_i|$ is attributed the base $2 \times |S_i| - \lceil \frac{|S_i|}{k} \rceil$ (line 02). Let us notice that no two partitions are attributed the same base. Then, a process p_i in partition ℓ considers the new name obtained from $RN[\ell]$ as an offset (notice that an offset in never equal to 0). It determines its final new name from the base and offset values it has been provided with, considering the name space starting from the base and going down (line 04).

3.4 Proof of the Algorithm

Lemma 2. *The algorithm described in Figure 1 ensures that no two processes obtain the same new name.*

Proof. Let p_i be a process such that $|S_i| = \ell_x$. That process is one of the $|ST[\ell_x]|$ processes that stop at the level ℓ_x and consequently use the underlying renaming object $RN[\ell_x]$. Due to the property of that renaming object, p_i computes a value $offset_i$ such that $1 \leq offset_i \leq 2 \times |ST[\ell_x]| - 1$. Moreover, as $|ST[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$ (Lemma 1), the previous relation becomes $1 \leq offset_i \leq 2 \times \min(k, \ell_x - \ell_{x-1})$.

On another side, the renaming space attributed to the processes p_i of $ST[\ell_x]$ starts at the base $2\ell_x - \lceil \frac{\ell_x}{k} \rceil$ (included) and goes down until $2\ell_{x-1} - \lceil \frac{\ell_{x-1}}{k} \rceil$ (excluded). Hence the size of this renaming space is

$$2(\ell_x - \ell_{x-1}) - \big(\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil\big).$$

It follows from these observations that a sufficient condition for preventing conflict in name assignment is to have

$$2 \times \min(k, \ell_x - \ell_{x-1}) - 1 \leq 2(\ell_x - \ell_{x-1}) - \big(\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil\big).$$

We prove that the algorithm satisfies the previous relation by considering two cases according to the minimum between k and $\ell_x - \ell_{x-1}$. Let

$$\ell_x = q_x\ k + r_x \text{ with } 0 \leq r_x < k \quad (\text{i.e., } \lceil\frac{r_x}{k}\rceil \in \{0,1\}), \quad \text{and}$$

$$\ell_{x-1} = q_{x-1}\ k + r_{x-1} \text{ with } 0 \leq r_{x-1} < k \quad (\text{i.e., } \lceil\frac{r_{x-1}}{k}\rceil \in \{0,1\}),$$

$$\text{from which we have } \ \ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1}).$$

- Case $\ell_x - \ell_{x-1} \leq k$.
  In that case, the relation to prove simplifies and becomes $\lceil\frac{\ell_x}{k}\rceil - \lceil\frac{\ell_{x-1}}{k}\rceil \leq 1$, i.e., $(q_x + \lceil\frac{r_x}{k}\rceil) - (q_{x-1} + \lceil\frac{r_{x-1}}{k}\rceil) \leq 1$, that can be rewritten as $(q_x - q_{x-1}) + (\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil) \leq 1$.
  Moreover, from $\ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1})$ and $\ell_x - \ell_{x-1} \leq k$, we have $(q_x - q_{x-1})\ k + (r_x - r_{x-1}) \leq k$ from which we can extract two subcases:

  - Case $q_x - q_{x-1} = 1$ and $r_x = r_{x-1}$.
    In that case, it trivially follows from the previous formulas that $(q_x - q_{x-1}) + (\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil) \leq 1$, which proves the lemma for that case.
  - Case $q_x = q_{x-1}$ and $0 \leq r_x - r_{x-1} \leq k$.
    In that case we have to prove $\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil \leq 1$. As $\lceil\frac{r_x}{k}\rceil, \lceil\frac{r_{x-1}}{k}\rceil \in \{0,1\}$, we have $\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil \leq 1$, which proves the lemma for that case.

- Case $k < \ell_x - \ell_{x-1}$.
  After simple algebraic manipulations, the formula to prove becomes:

  $$(2k-1)(q_x - q_{x-1} - 1) + 2(r_x - r_{x-1}) - \big(\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil\big) \geq 0.$$

  Moreover, we have now $\ell_x - \ell_{x-1} = (q_x - q_{x-1})\ k + (r_x - r_{x-1}) > k$, from which, as $0 \leq r_x, r_{x-1} < k$, we can conclude $q_x - q_{x-1} \geq 1$. We consider two cases.

  - $q_x - q_{x-1} = 1$.
    The formula to prove becomes $2(r_x - r_{x-1}) \geq \lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil$.
    From $\ell_x - \ell_{x-1} > k$ we have:

    * $r_x > r_{x-1}$, from which (as r_x and r_{x-1} are integers) we conclude $2(r_x - r_{x-1}) \geq 2$.
    * $1 \geq \lceil\frac{r_x}{k}\rceil \geq \lceil\frac{r_{x-1}}{k}\rceil \geq 0$, from which we conclude $\lceil\frac{r_x}{k}\rceil - \lceil\frac{r_{x-1}}{k}\rceil \leq 1$.

    By combining the previous relations we obtain $2 \geq 1$ which proves the lemma for that case.

- $q_x - q_{x-1} > 1$. Let $q_x - q_{x-1} = 1 + \alpha$ (where α is an integer ≥ 1). The formula to prove becomes
$$(2k-1)\alpha + 2(r_x - r_{x-1}) - \big(\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil\big) \geq 0.$$
As $0 \leq r_x, r_{x-1} < k$, the smallest value of $r_x - r_{x-1}$ is $-(k-1)$. Similarly, the greatest value of $\lceil \frac{r}{k} \rceil - \lceil \frac{r_{\ -1}}{k} \rceil$ is 1.
It follows that, the smallest value of the left side of the formula is $(2k-1)\alpha - 2(k-1) - 1 = 2k\alpha - (2k+\alpha) + 1 = (2k-1)(\alpha-1)$. As $k \geq 1$ and $\alpha \geq 1$, it follows that the left side is never negative, which proves the lemma for that case.

$\Box_{Lemma}$ 2

Theorem 1. *The algorithm described in Figure 1 is a wait-free adaptive* $(2p - \lceil \frac{p}{k} \rceil)$*-renaming algorithm (where* $p \leq n$ *is the number of participating processes).*

Proof. The fact that the algorithm is wait-free is an immediate consequence of the fact that base k-set participating set object and the base renaming objects are wait-free. The fact that no two processes obtain the same new name is established in Lemma 2.

If p processes participate in the algorithm, the highest level at which a process stops is p (this follows from the properties of the k-set participating set object). Consequently, the largest base that is used (line 02) is $2p - \lceil \frac{p}{k} \rceil$, which establishes the upper bound on the renaming space. $\Box_{Theorem}$ 1

4 Visiting Gafni's Land: From *k*-Set to *k*-Participating Set

This section presents a wait-free transformation from a k-set agreement object to a k-participating set object. It can be seen as a guided visit to Gafni's reduction land [6,7,10]. Let us recall that a k-set object provides the processes with an operation $\mathsf{kset\_propose}_k()$.

4.1 From Set Agreement to Strong Set Agreement

Let us observe that, given a k-set object, it is possible that no process decides the value it has proposed. This feature is the "added value" provided by a *strong k-set agreement* object: it is a k-set object (i.e., at most k different values are decided) such that at least one process decides the value it has proposed [7]. The corresponding operation is denoted $\mathsf{strong\_kset\_propose}_k()$.

In addition to a k-set object KS, the processes cooperate by accessing an array $DEC[1..n]$ of one-writer/multi-reader atomic registers. That array is initialized to $[\bot, \ldots, \bot]$. $DEC[i]$ can be written only by p_i. The array is provided with a $\mathsf{snapshot}()$ operation. Such an operation returns a value of the whole array as if that value has been obtained by atomically reading the whole array [1]. Let us remind that such an operation can be wait-free implemented on top of atomic read/write base registers.

The construction (introduced in [7]) is described in Figure 2. A process p_i first proposes its original name to the underlying k-set object KS, and writes the value it obtains (an original name) into $DEC[i]$ (line 01). Then, p_i atomically reads the whole array (line 02). Finally, if it observes that some process has decided its original name id_i, p_i also decides id_i, otherwise p_i decides the original name it has been provided with by the k-set object (lines 03-04).

```
operation strong_kset_propose$_k(id_i)$ :
(1)  $DEC[i] \leftarrow KS$.kset_propose$_k(id_i)$;
(2)  $dec_i[1..n] \leftarrow$ snapshot$(DEC[1..n])$;
(3)  if $(\exists h : dec_i[h] = id_i)$ then $decision_i \leftarrow id_i$ else $decision_i \leftarrow dec_i[i]$ endif;
(4)  return$(decision_i)$
```

Fig. 2. Strong k-set agreement algorithm (code for p_i)

4.2 From Strong Set Agreement to *k*-Participating Set

The specification of a k-participating set object has been defined in Section 3.2. The present section shows how such an object PS can be wait-free implemented from an array of strong k-set agreement objects; this array is denoted $SKS[1..n]$. (This construction generalizes the construction proposed in [10] that considers $n = 3$ and $k = 2$.) In addition to the array $SKS[1..N]$ of strong k-set agreement objects, the construction uses an array of one-writer/multi-reader atomic registers denoted $LEVEL[1..n]$. As before only p_i can write $LEVEL[i]$. The array is initialized to $[n+1, \dots, n+1]$.

The algorithm is based on what we call *Borowski-Gafni's ladder*, a wait-free object introduced in [6]. It combines such a ladder object with a k-set agreement object in order to guarantee that no more than k processes, that do not crash, stop at the same step of the ladder.

Borowsky-Gafni's Ladder. Let us consider the array $LEVEL[1..n]$ as a ladder. Initially, a process is at the top of the ladder, namely, at level $n + 1$. Then it descends the ladder, one step after the other, according to predefined rules until it stops at some level (or crashes). While descending the ladder, a process p_i registers its current position in the ladder in the atomic register $LEVEL[i]$.

After it has stepped down from one ladder level to the next one, a process p_i computes a local view (denoted $view_i$) of the progress of the other processes in their descent of the ladder. That view contains the processes p_j seen by p_i at the same or a lower ladder level (i.e., such that $level_i[j] \leq LEVEL[i]$). Then, if the current level ℓ of p_i is such that p_i sees at least ℓ processes in its view (i.e., processes that are at its level or a lower level) it stops at the level ℓ of the ladder. This behavior is described by the following algorithm [6]:

```
repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
        for $j \in \{1, \dots, n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end_do;
        $view_i \leftarrow \{j : level_i[j] \leq LEVEL[i]\}$;
until $(|view_i| \geq LEVEL[i])$ end_repeat;
let $S_i = view_i$; return$(S_i)$.
```

This very elegant algorithm satisfies the following properties [6]. The sets S_i of the processes that terminate the algorithm, satisfy the self-membership, comparability and immediacy properties of the k-participating set object. Moreover, if $|S_i| = \ell$, then p_i stopped at the level ℓ, and there are ℓ processes whose current level is $\leq \ell$.

From a ladder to a k-participating set object. The construction, described in Figure 3, is nearly the same as the construction given in [10]. It uses the previous ladder algorithm as a skeleton to implement a k-participating set object. When it invokes $\mathsf{participating\_set}_k()$, a process p_i provides its original name as input parameter. This name will be used by the underlying strong k-participating set object. The array $INIT\_NAME[1..n]$ is initialized to $[\perp, \ldots, \perp]$. $INIT\_NAME[i]$ can be written only by p_i.

```
operation participating_set_k($id_i$)
(1)  $INIT\_NAME[i] \leftarrow id_i$;
(2)  repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
(3)     for $j \in \{1, \ldots, n\}$ do $level_i[j] \leftarrow LEVEL[j]$ end_do;
(4)     $view_i \leftarrow \{j : level_i[j] \leq LEVEL[i]\}$;
(5)     if $(LEVEL[i] > k) \wedge (|view_i| = LEVEL[i])$
(6)              then let $\ell = LEVEL[i]$;
(7)                   $ans_i \leftarrow SKS[\ell]$.strong_kset_propose_k($id_i$);
(8)                   $ok_i \leftarrow (ans_i = id_i)$
(9)              else $ok_i \leftarrow true$
(10)    endif
(11) until $(|view_i| \geq LEVEL[i]) \wedge ok_i$ end_repeat;
(12) let $S_i = \{id \mid \exists j \in view_i$ such that $INIT\_NAME[j] = id\}$;
(13) return($S_i$)
```

Fig. 3. k-participating set algorithm (code for p_i)

If, in the original Borowski-Gafni's ladder, a process p_i stops at a ladder level $\ell \leq k$, it can also stop at the same level in the k-set participating object. This follows from the fact that, as $|view_i| = \ell \leq k$ when p_i stops descending, we know from the ladder properties that at most $\ell \leq k$ processes are at the level ℓ (or at a lower level). So, when $LEVEL[i] \leq k$ (line 05), p_i sets ok_i to *true* (line 05). It consequently exits the repeat loop (line 11) and we can affirm that no more than k processes do the same, thereby satisfying the bounded simultaneity property.

So, the main issue of the algorithm is to satisfy the bounded simultaneity property when the level ℓ at which p_i should stop in the original Borowski-Gafni's ladder is higher than k. In that case, p_i uses the underlying strong k-set agreement object $SKS[\ell]$ to know if it can stop at that level (lines 07-08). This k-participating set object ensures that at least one (and at most k) among the participating processes that should stop at level ℓ in the original Borowski-Gafni's ladder, do actually stop. If a process p_i is not allowed to stop (we have then $ok_i = false$ at line 08), it is required to descend to the next step of the ladder (lines 11 and 01). When a process stops at a level ℓ, there are exactly ℓ processes at the levels $\ell' \leq \ell$. This property is maintained when a process steps down from ℓ to $\ell - 1$ (this follows from the fact that when a process is required to step down from $\ell > k$ to $\ell - 1$ because $\ell > k$, at least one process remains at the level ℓ due to the k-set agreement object $SKS[\ell]$).

5 From Ω^k to k-Set Objects

This section shows that a k-set object can be built from single-writer/multi-reader atomic registers and an oracle (failure detector) of the class Ω^k.

5.1 The Oracle Class Ω^k

The family of oracle classes $(\Omega^z)_{1 \leq z \leq n}$ has been introduced in [16]. That definition implicitly assumes that all the processes are participating. We extend here this definition by making explicit the notion of *participating* processes. More precisely, an oracle of the class Ω^z provides the processes with an operation denoted leader() that satisfies the following properties:

- Output size: each time it is invoked, leader() provides the invoking process with a set of at most *z participating* process identities (e.g., $\{id_{x_1}, \ldots, id_{x} \}$).
- Eventual multiple leadership: There is a time after which all the leader() invocations return forever the same set. Moreover, this set includes at least one *correct participating* process (if any).

It is important to notice that each instance of Ω^k is defined with respect to the *context* where it is used. This context is the set of participating processes. This means that if Ω^k is used to construct a given object, say a k-set object KS, the participating processes for that failure detector instance are the processes that invoke KS.kset_propose$_k$(). Let us remark that, during an arbitrary long period, the participating processes that invoke leader() can see different sets of leaders, and no process knows when this "anarchy" period is over. Moreover, nothing prevent faulty processes to be elected as permanent leaders.

When all the processes are assumed to participate, Ω^1 is nothing else than the leader failure detector denoted Ω introduced in [8], where it is shown that it is the weakest failure detector for solving the consensus problem in asynchronous systems. (Let us notice that the lower bound proved in [8], on the power of failure detectors, assumes implicitly that all the correct processes participate in the consensus algorithm.)

5.2 From Ω^k to k-Set Agreement

In addition to an oracle of the class Ω^k, the proposed k-set agreement algorithm is based on a variant, denoted KA, of a round-based object introduced in [11] to capture the safety properties of Paxos-like consensus algorithms [14]. The leader oracle is used to ensure the liveness of the algorithm. KA is used to abstract away the safety properties of the k-set problem, namely, at most k values are decided, and the decided values are have been proposed.

The KA object This object provides the processes with an operation denoted $\mathsf{alpha\_propose}_k(r_i, v_i)$. That operation has two input parameters: the value v_i proposed by the invoking process p_i (here its name id_i), and a round number r_i (that allows identifying the invocations). The KA object assumes that no two processes use the same round numbers, and successive invocations by the same process use increasing round numbers. Given a KA object, the invocations $\mathsf{alpha\_propose}_k()$ satisfy the following properties:

- Validity: the value returned by any invocation $\mathsf{alpha\_propose}_k()$ is a proposed value or $\perp$.
- Agreement: At most k different non-$\perp$ values can be returned by the whole set of $\mathsf{alpha\_propose}_k()$ invocations.
- Convergence: If there is a time after which the operation $\mathsf{alpha\_propose}_k()$ is invoked infinitely often, and these invocations are issued by an (unknown but fixed) set of at most k processes, then there is a time after which none of these invocations returns $\perp$.

An algorithm constructing a KA object from single-writer/multi-reader atomic registers is described in [15].

The k-set algorithm. The algorithm constructing a k-set object KS (accessed by at most n processes[1]), is described in Figure 4. As in previous algorithms, it uses an array $DEC[1..n]$ of one-writer/multi-reader atomic registers. Only p_i can write $DEC[i]$. The array is initialized to $[\perp, \ldots, \perp]$. The algorithm is very simple. If a value has already been decided ($\exists j : DEC[j] \neq \perp$), p_i decides it. Otherwise, p_i looks if it is a leader. If it is not, it loops. If it is a leader, p_i invokes $\mathsf{alpha\_propose}_k(r_i, v_i)$ and writes in $DEC[i]$ the value it obtains (it follows from the specification of KA that that value it writes is $\perp$ or a proposed value).

```
operation kset_propose_k($v_i$):
$r_i \leftarrow (i - n)$;
while ($\forall j : DEC[j] = \perp$) do
   if $id_i \in$ leader() then $r_i \leftarrow r_i + n$; $DEC[i] \leftarrow KA$.alpha_propose_k($r_i, v_i$) endif
end_while;
let $decided_i =$ any $DEC[j] \neq \perp$;
return($decided_i$)
```

Fig. 4. An Ω^k-based k-set algorithm (code for p_i)

It is easy to see that no two processes use the same round numbers, and each process uses increasing round numbers. It follows directly from the agreement property of the KA object, that at any time, the array $DEC[1..n]$ contains at most k values different from $\perp$. Moreover, due the validity property of KA, these values have been proposed.

[1] Let us remind that the construction of each $SKS[\ell]$ object used in Figure 3 is based on an underlying k-set object KS object.

It is easy to see that, as soon as a process has written a non-$\bot$ value in $DEC[1..n]$, any kset_propose(v_i) invocation issued by a correct process terminates. So, in order to show that the algorithm is wait-free, we have to show that at least one process writes a non-$\bot$ value in $DEC[1..n]$. Let us assume that no process deposits a value in this array. Due to the eventual multiple leadership property of Ω^k, there is a time τ after which the same set of $k' \leq k$ participating processes are elected as permanent leaders, and this set includes at least one correct process. It follows from the algorithm that, after τ, at most k processes invoke KA.alpha_propose$_k$(), and one of them is correct. It follows from the convergence property of the KA object, that there is a time $\tau' \geq \tau$ after which no invocation returns $\bot$. Moreover, as at least one correct process belongs to the set of elected processes, that process eventually obtains a non-$\bot$ value from an invocation, and consequently deposits that non-$\bot$ value in $DEC[1..n]$. The algorithm is consequently wait-free.

References

1. Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.
2. Afek Y. and Merritt M., Fast, Wait-Free $(2k-1)$-Renaming. *Proc. 18th ACM Symp. on Principles of Dist. Comp. (PODC'99)*, ACM Press, pp. 105-112, 1999.
3. Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.
4. Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2k-1)$-Renaming. *Proc. Symp. on Dist. Comp. (DISC'00)*, LNCS #1914, pp. 149-163, 2000.
5. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004.
6. Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symp. on Principles of Distr. Comp. (PODC'93)*, pp. 41-51, 1993.
7. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM STOC*, pp. 91-100, 1993.
8. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
9. Gafni E., Read/Write Reductions. *DISC/GODEL presentation given as introduction to the 18th Int'l Symposium on Distributed Computing (DISC'04)*, 2004.
10. Gafni E., Rajsbaum R., Raynal M. and Travers C., The Committee Decision Problem. *Proc. 8th LATIN*, LNCS #3887, pp. 502-514, 2006.
11. Guerraoui R. and Raynal M., The Alpha of Asynchronous Consensus. *The Computer Journal.* To appear.
12. Herlihy M.P., Wait-Free Synchronization. *ACM TOPLAS*, 13(1):124-149, 1991.
13. Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923,, 1999.
14. Lamport L., The Part-Time Parliament. *ACM TOCS*, 16(2):133-169; 1998.
15. Mostefaoui M., Raynal M., and Travers C., Exploring Gafni's Reduction land: from Ω^k to Wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set Agreement. *Tech Report #1676*, IRISA, Université de Rennes (France), 2006.
16. Neiger G., Failure Detectors and the Wait-Free Hierarchy. *Proc. 14th Int'l ACM Symp. on Principles of Dist. Comp. (PODC'95)*, ACM Press, pp. 100-109, 1995.
17. Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.

Renaming in Message Passing Systems with Byzantine Failures

Michael Okun and Amnon Barak

Department of Computer Science, The Hebrew University of Jerusalem
{mush, amnon}@cs.huji.ac.il

Abstract. We study the renaming problem in a fully connected synchronous network with Byzantine failures. We show that when faulty processors are able to cheat about their original identities, this problem cannot be solved in an a priori bounded number of rounds for $t \geq (n + n \bmod 3)/3$, where n is the size of the network and t is the number of failures. This result also implies a $t \geq (n+n \bmod 4)/2$ bound for the case of faulty processors that are not able to falsify their original identities. In addition, we present several Byzantine renaming algorithms based on distinct approaches, each providing a different tradeoff between its running time and the solution quality.

1 Introduction

In the renaming problem, n processors have to cooperatively select for themselves new names from a namespace whose size depends only on n, in a way that guarantees that each correct processor has a distinct new name. In the crash failure case, this problem was extensively investigated in both the message passing and the shared memory models. So far the renaming problem was not studied in the Byzantine failure case.

Consider for example a group of (client) processes that wish to access a data set or a service which is replicated on a large number of servers. For best performance, each process should access a different server. When some of the processes experience Byzantine failures [13], e.g., they are controlled by a malicious adversary, this might not be possible since the faulty processes may access any number of servers (in the worst case each faulty process accesses every server). However, even in such cases, the correct processes benefit from reducing the contention among themselves. This paper deals with the general problem demonstrated by the above example, in which the the processes are required to solve an instance of the renaming problem in the presence of Byzantine failures.

From the theoretical point of view, renaming belongs to the class of symmetry breaking problems, which can be regarded as the simplest nontrivial distributed computing task [9]. As we shall see, in the Byzantine failure case, similarly to crash failures, renaming raises a number of questions that are not addressed by other well known problems (such as consensus). In order to understand the implications of Byzantine failures for renaming, it is convenient (at least at the

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 16–30, 2006.

beginning) to consider the problem in the context of standard well known models, despite the fact that some aspects of the scenario described in the above example will not be accurately represented.

Specifically, in this paper we consider a synchronous network of an a priori known size n, in which every pair of processors is connected by a communication channel (link). Each processor has a unique identifier (*processorID*), which is originally known only to its owner (if the identifiers of the processors are globally known, the renaming task has a trivial solution). The communication between processors is performed by message passing, which satisfies one of the following conditions (see also [16]):

($\mathcal{M}_L$) A faulty processor may send messages with arbitrary identifiers. Messages from different processors can be distinguished according to the *link* through which they are received.

($\mathcal{M}_I$) The true unique *identifier* of each processor is included in any message it sends. [1]

The renaming problem can be formally defined by the following conditions [3,6,20]:

(*Termination*) Each correct processor eventually decides on a new name.

(*Uniqueness*) All the new names belong to the target namespace, and no two correct processors decide on the same name.

The above conditions also have stronger versions:

(*Strong termination* [10]) Each correct processor decides on a new name during $r = r(n)$ rounds, where r depends only on the size of the system.

(*Order preserving*) The new names of the correct processors preserve the order imposed by their original identifiers.

The paper has two main parts. Section 2 deals with impossibility results for Byzantine renaming. We show that in the $\mathcal{M}_L$ model, strong termination is not possible if $n + (n \bmod 3) \leq 3t$, where t is the number of faulty processors. The same proof also implies impossibility of strongly terminating Byzantine renaming in the $\mathcal{M}_I$ model, when $n + (n \bmod 4) \leq 2t$.

Section 3 focuses on efficient algorithms for the Byzantine renaming problem. We show three algorithms, each representing a different approach. In the first algorithm renaming is achieved by using Byzantine agreement (solving an instance of group membership problem). This approach allows to solve the order-preserving renaming problem using a small target namespace, in $O(n)$ time. The second algorithm uses *splitting* to solve the non-order preserving problem directly, in $O(\log n)$ rounds. Its target namespace, however, is $\Theta(\text{poly}(n))$. The third approach is to adapt the original algorithm of Attiya et al. [3] to the Byzantine failure case, to get an algorithm that works in the asynchronous case.

[1] This model is closer to the typical practical case, in which there are no dedicated communication links for each pair of processors. Such an assumption is necessary in order to prevent a single Byzantine processor from attacking a system by counterfeiting multiple identities (the Sybil attack [11]).

1.1 Related Previous Work

The renaming problem was originally introduced in [3] for asynchronous message passing system with crash failures. This landmark paper presented a simple renaming algorithm with a target namespace of size $(n - t/2)(t + 1)$, followed by a more intricate algorithm with a target namespace of size $(n + t)$, and an order preserving algorithm with a target namespace of size $2^t(n - t + 1) - 1$. The last result was also shown to be tight. The wait-free renaming problem in the synchronous message passing model with crash failures was studied in [9], which presents an $O(\log n)$-round algorithm and a proof that for comparison based algorithms this result is optimal.

The renaming problem was also studied in the shared memory model, first in the original *one-shot* setting [7,8], and then in the *long-lived* version, where processors request and release the new names dynamically [15]. In [15], the *splitter* object was used to solve the problem, an approach which was subsequently adopted in several follow up papers. More recently, both the one-shot and the long-lived versions of the problem were studied in the *adaptive* setting, where the number of participating processors k, is not known in advance [1,2,4,5]. In this setting the goal is to develop efficient wait-free algorithms whose target namespace and complexity depend only on k.

The question of the minimum possible target namespace was settled by the groundbreaking work of Herlihy and Shavit, as a special variant of their Asynchronous Computability Theorem [12]. They have shown that $(n + t)$ is indeed the smallest possible namespace for tolerating t failures in an asynchronous environment, for both the shared memory and the message passing models.

A review of the Byzantine agreement problem [19,13] is beyond the scope of this paper. A presentation of this topic can be found in [14,6].

2 Impossibility Results for Byzantine Renaming

This section deals with impossibility results for Byzantine renaming. First, observe that the Byzantine renaming problem (in a synchronous system) can be solved for any $n > t$, if strong termination is not required. A possible solution (assuming w.l.o.g. that processor ids are natural numbers), that is also order preserving, is presented in Fig. 1. The running time of this algorithm depends on the *values* of the ids, and thus can be arbitrarily long. We show that in model $\mathcal{M}_L$, for $n + (n \ mod \ 3) \leq 3t$ this is the best possible solution, i.e., there exists no deterministic strongly terminating renaming algorithm.

We start by proving that strongly terminating renaming is impossible in a system with three processors one of which can be faulty, when the identifiers of the processors belong to some infinite set $\mathfrak{I}$. For the proof, assume that the port numbering is as shown in Fig. 2.

Suppose by contradiction that there exists a deterministic algorithm $\mathcal{A}$ that solves the renaming problem in r rounds when one processor can arbitrarily fail. Further, assume that $\mathcal{A}$ is a full-information algorithm in which each correct processor distributes all its current information in every round.

```
Initial setup:
$M := \{1, \ldots, n\}$
In rounds $1, \ldots, processorID - 1$:
for every link $i$ do
    if $m$ is the first ever value received via link $i$ then
        $M := M \setminus \{m\}$
In round $processorID$:
decide on $min(M)$
send $min(M)$ to all
```

Fig. 1. A Byzantine renaming algorithm with eventual (weak) termination only

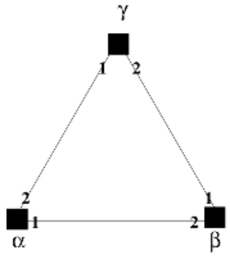


Fig. 2. A symmetric 3-processor system

To begin with, due to the symmetry of the port numbers in Fig. 2, each processor knows the port numbers through which the other two processors communicate with it. Therefore, the only new information available to a correct processor by the end of the first round is the identifiers sent by the other two processors. Generally, by the end of r rounds, the processor receives r identifiers from each port.

Let $\mathcal{A}(\alpha, \beta_1, ..., \beta_r, \gamma_1, ..., \gamma_r)$ be the output of $\mathcal{A}$ executed on a processor whose *processorID* is α, when it gets in round $1 \leq s \leq r$ identifier β_s via port 1 and identifier γ_s via port 2. Then algorithm $\mathcal{A}$ must satisfy:

$$\begin{gathered} \alpha \neq \beta_1 \Rightarrow \\ \forall \beta_{r+1}, \gamma_r \;\; \mathcal{A}(\alpha, \beta_1, ..., \beta_r, \gamma_1, ..., \gamma_r) \neq \mathcal{A}(\beta_1, \beta_2, ..., \beta_{r+1}, \alpha, \gamma_1, ..., \gamma_{r-1}), \end{gathered} \tag{1}$$

otherwise in an execution shown in Fig. 3 the correct processors α and β_1 decide on the same new id.

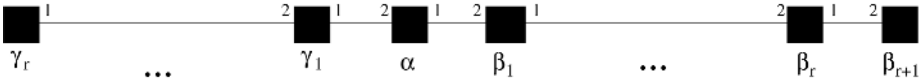


Fig. 3. A scenario for r-round renaming

To complete the proof we show that such a *function* cannot exist (since $\mathfrak{I}$ is assumed to be infinite, it is sufficient to prove this claim for $\mathfrak{I} = \mathbb{Z}$). In order to state the combinatorial theorem which implies this result, we introduce the following definitions.

Let $\widetilde{\mathbb{Z}}^n = \{(a_1, ..., a_n) \in \mathbb{Z}^n | \forall 1 \leq i, j \leq n \ \ i \neq j \Rightarrow a_i \neq a_j\}$, i.e., $\widetilde{\mathbb{Z}}^n$ contains only the points of $\mathbb{Z}^n$ that have n distinct coordinate values.

Let $L_k(a_1, ..., a_{k-1}, a_{k+1}, ..., a_n)$ denote a line in $\widetilde{\mathbb{Z}}^n$ parallel to the k-th axis:

$$\begin{aligned}&L_k(a_1, ..., a_{k-1}, a_{k+1}, ..., a_n) = \\ &\{(a_1, ..., a_{k-1}, z, a_{k+1}, ..., a_n) \,|z \in \mathbb{Z} \setminus \{a_1, ..., a_{k-1}, a_{k+1}, ..., a_n\}\} \,,\end{aligned}$$

where $a_1, ..., a_{k-1}, a_{k+1}, ..., a_n$ are any distinct integers[2] and $1 \leq k \leq n$.

For any $\sigma \in S_n$ (a permutation on n elements) and a point $\alpha = (a_1, ..., a_n)$ in $\widetilde{\mathbb{Z}}^n$, the point $\sigma(\alpha) \in \widetilde{\mathbb{Z}}^n$ is defined by $\sigma(\alpha) = (a_{\sigma^{-1}(1)}, a_{\sigma^{-1}(2)}, ..., a_{\sigma^{-1}(n)})$. A function $C : \widetilde{\mathbb{Z}}^n \to \mathcal{C}$, where $\mathcal{C}$ is a finite set, is called a (finite) coloring of $\widetilde{\mathbb{Z}}^n$.

For any $S \subseteq \widetilde{\mathbb{Z}}^n$, let $\sigma(S)$ and $C(S)$ denote the sets $\{\sigma(\alpha) \,|\alpha \in S\} \subseteq \widetilde{\mathbb{Z}}^n$ and $\{C(\alpha) \,|\alpha \in S\} \subseteq \mathcal{C}$, respectively. These definitions imply:

$$\begin{aligned}&\sigma \left(L_k(a_1, ..., a_{k-1}, a_{k+1}, ..., a_n)\right) = \\ &L_{\sigma(k)}(a_{\sigma^{-1}(1)}, a_{\sigma^{-1}(2)}, ..., a_{\sigma^{-1}(\sigma(k)-1)}, a_{\sigma^{-1}(\sigma(k)+1)}, ..., a_{\sigma^{-1}(n)}).\end{aligned} \tag{2}$$

Theorem 1. *For any $n \geq 2$, $n \geq k \geq 1$ and any cyclic permutation $\sigma \in S_n$, there exists no finite coloring C of $\widetilde{\mathbb{Z}}^n$ such that for every line L parallel to the k-th axis of $\widetilde{\mathbb{Z}}^n$*

$$C(L) \cap C(\sigma(L)) = \emptyset. \tag{3}$$

Before proving the theorem, we show that it implies the non-existence of a renaming algorithm. The key observation is that $\mathcal{A}$ can be considered as a coloring $\mathcal{A} : \widetilde{\mathbb{Z}}^{2r+1} \to \mathcal{C}$, where $\mathcal{C}$ is the finite target namespace of $\mathcal{A}$. For the permutation $\sigma = (1(r+2)(r+3)...(2r)(2r+1)(r+1)r...2)$, condition (1) implies that for any line L parallel to the $(2r+1)$-th axis, $\mathcal{A}(L) \cap \mathcal{A}(\sigma(L)) = \emptyset$. However, according to Theorem 1 no such finite coloring exists.

The theorem is proved by showing that a finite coloring of $\widetilde{\mathbb{Z}}^n$ (which satisfies the required condition), if it existed, could have been used to define a coloring of $\widetilde{\mathbb{Z}}^{n-1}$ with a larger (but finite) set of colors. In the context of renaming this means that an algorithm that runs in r rounds could have been used to construct an algorithm with larger target namespace that runs in $r-1$ rounds.

Proof of Theorem 1. Assume that the theorem is false for $n = 2$ and let $C : \widetilde{\mathbb{Z}}^2 \to \mathcal{C}$ be a coloring that satisfies (3) for $\sigma = (12)$. W.l.o.g. assume that $k = 2$. Let $L_2(a)$ and $L_2(b)$ be two different lines, such that $C(L_2(a)) = C(L_2(b))$. Observe that $L_2(a) \cap \sigma(L_2(b)) = (a, b)$. Therefore, $C(a, b) \in C(L_2(a)) = C(L_2(b))$ and $C(a, b) \in C(\sigma(L_2(b)))$, which is a contradiction.

Suppose that the theorem is false and let $n > 2$ be the minimal dimension for which the theorem is incorrect. W.l.o.g., assume that $k = n$. Let $C : \widetilde{\mathbb{Z}}^n \to \mathcal{C}$ be a coloring that satisfies (3) for $\sigma = (\sigma_1...\sigma_n)$. W.l.o.g. suppose that $n = \sigma_{n-1}$. Let

[2] In the sequel this coordinate numbering will be more convenient than the more obvious $a_1, ..., a_{n-1}$.

$\pi = (\sigma_1 ... \sigma_{n-2}\sigma_n)$ be a cyclic permutation on $n-1$ elements. Define a coloring $D_1 : \widetilde{\mathbb{Z}}^{n-1} \to 2^{\mathcal{C}}$ by $D_1(a_1, ..., a_{n-1}) = C\left(L_n(a_1, ..., a_{n-1})\right)$. In addition, let D_2 be a finite coloring that satisfies $D_2(\alpha) \neq D_2(\pi(\alpha))$ for any $\alpha \in \widetilde{\mathbb{Z}}^{n-1}$. Next, we show that the finite coloring D defined by $D(\alpha) = (D_1(\alpha), D_2(\alpha))$ satisfies (3) for every line in $\widetilde{\mathbb{Z}}^{n-1}$ which is parallel to the $\sigma^{-1}(n)$-th axis, with respect to the cyclic permutation π. Since this contradicts the choice of n, this completes the proof.

Let $L = L_{\sigma^{-1}(n)}(a_1, ..., a_{\sigma^{-1}(n)-1}, a_{\sigma^{-1}(n)+1}, ..., a_{n-1})$. Since $\forall i \neq \sigma(n)\ \pi^{-1}(i) = \sigma^{-1}(i)$, $\pi^{-1}(\sigma(n)) = \sigma^{-1}(n)$ and $\pi(\sigma^{-1}(n)) = \sigma(n)$, by applying (2) we get:

$$\pi(L) = L_{\sigma(n)}(a_{\sigma^{-1}(1)}, a_{\sigma^{-1}(2)}, ..., a_{\sigma^{-1}(\sigma(n)-1)}, a_{\sigma^{-1}(\sigma(n)+1)}, ..., a_{\sigma^{-1}(n-1)}).$$

If $D(L) \cap D(\pi(L)) \neq \emptyset$, there must exist two integers $a_{\sigma^{-1}(n)}$ and a_n such that D assigns the same value to the points

$$\alpha = (a_1, ..., a_{\sigma^{-1}(n)-1}, a_{\sigma^{-1}(n)}, a_{\sigma^{-1}(n)+1}, ..., a_{n-1}) \in L$$

and

$$\beta = (a_{\sigma^{-1}(1)}, ..., a_{\sigma^{-1}(\sigma(n)-1)}, a_{\sigma^{-1}(\sigma(n))}, a_{\sigma^{-1}(\sigma(n)+1)}, ..., a_{\sigma^{-1}(n-1)}) \in \pi(L).$$

Moreover $a_{\sigma^{-1}(n)} \neq a_n$, since otherwise it holds that $\pi(\alpha) = \beta$, which implies $D_2(\alpha) \neq D_2(\beta)$ and thus $D(\alpha) \neq D(\beta)$ as well.

From $D_1(\alpha) = C(L_n(a_1, ..., a_{n-1})) = C(L_n(a_{\sigma^{-1}(1)}, ..., a_{\sigma^{-1}(n-1)})) = D_1(\beta)$, it follows that there must exist a point $\gamma \in L_n(a_1, ..., a_{n-1})$ such that

$$C(\gamma) = C(a_{\sigma^{-1}(1)}, ..., a_{\sigma^{-1}(n)}).$$

However the point $(a_{\sigma^{-1}(1)}, ..., a_{\sigma^{-1}(n)})$ belongs also to the line $\sigma(L_n(a_1, ..., a_{n-1}))$, and thus we found a line for which C does not satisfy (3), contrary to the assumption. □

This completes the proof of the impossibility of strongly terminating renaming in model $\mathcal{M}_L$ in a 3-processor system with 1 failure. The same proof holds for a 4-processor system with 2 faults in model $\mathcal{M}_I$, since in the particular case in which each correct processor sees a different faulty processor, the views of the correct processors are exactly as above.

Next we would like to extend the impossibility result to a system with n processors, to show that strongly terminating renaming in presence of $\lceil n/3 \rceil$ faulty processors is impossible. Usually this can be done by the following simple simulation argument (e.g., see [13]). If there were an algorithm for a system with n processors that tolerates $\lceil n/3 \rceil$ faults, it would be possible to divide the n processors to three nearly equal sets, and let each processor in a 3-processor system simulate one of the sets. This would give an algorithm for the case $n = 3$, $t = 1$, which was already shown to be impossible. However, in the current case such a simulation argument cannot be applied straightforwardly, since according to the model definition, the algorithms executed by the processors must be completely *identical.* Thus, a more careful simulation argument is required.

For $n \equiv 0 \; mod \; 3$ there exist port labeling schemes of K_n (a complete graph on n nodes) which divide its nodes to 3 isomorphic sets of $n/3$ nodes each, such that these sets are interconnected in a symmetrical way. An algorithm $\mathcal{A}$ that solves the Byzantine renaming problem in K_n with such a labeling scheme in the presence of $n/3$ failures, can be simulated in a 3-processor system presented in Fig. 2. This would give an algorithm for renaming in this 3-processor system, which is a contradiction. Therefore, no such $\mathcal{A}$ exists.

Next, suppose $n \equiv 1 \; mod \; 3$. Consider a labeling of K_n in which there exists one special node that is seen by all the other nodes via their $(n-1)$ link, while all the other nodes are divided to three isomorphic sets connected as in the previous case. The special node is assumed to be crashed from the very beginning, and in addition 1/3 of the remaining nodes can experience a Byzantine failure. Even for this specific case, there exists no strongly terminating renaming algorithm, since it can be turned into an algorithm for 3-processor system in Fig. 2, as in the $n \equiv 0 \; mod \; 3$ case.

The case $n \equiv 2 \; mod \; 3$ is similar, except that now two special nodes are required. Therefore, we can only prove that there exists no renaming algorithm that tolerates $(n-2)/3+2$ failures. Together the three cases imply the following:

Theorem 2. *In model $\mathcal{M}_L$, in a system of n processors of which t may fail, where $n + (n \; mod \; 3) \leq 3t$, there exists no algorithm that solves the strongly terminating Byzantine renaming problem.*

Since a similar difficulty arises when trying to extend the impossibility result in the 4-processor case to a system with n processors in model $\mathcal{M}_I$, we can only prove that there exists no strongly terminating algorithm for $n+(n \; mod \; 4) \leq 2t$.

3 Algorithms

In this section we study the algorithmic aspect of the Byzantine renaming problem, by considering several different approaches for constructing such algorithms. Section 3.1 describes a simple renaming algorithm based on Byzantine agreement. Section 3.2 presents an algorithm that solves the problem by first iteratively splitting the processors into smaller groups until at least one correct processor gets a unique new name, and then using it to assign all the processors unique new names. In Section 3.3 we briefly discuss solutions based on the original algorithms of Attiya et al. [3]. All the presented algorithms assume $n > 3t$ and work for model $\mathcal{M}_L$.

3.1 Renaming Using Byzantine Agreement

Using Byzantine agreement it is possible to solve the renaming problem in a simple and natural way. Moreover, it provides a “high quality” solution: the target namespace can be made small and the original order of the ids is automatically preserved. We note that Byzantine agreement was studied mainly under the assumption of a priori acquainted processors, which does not hold here. Yet, such an assumption is not necessary [16,17,18].

The idea of our renaming algorithm is rather simple: first each correct processor p computes two sets of ids, where the first set (J_p) contains all the processor ids known to p, and the second set (I_p) contains only the ids of the "well behaved" processors. In addition, these sets satisfy $I_p \subseteq J_q$ for any two correct processors p and q. Next, an instance of Byzantine agreement protocol is executed for every id, to decide if it should be taken into account. A correct processor p participates only in instances of ids that appear in J_p. The decisions are consistent because whenever an id belongs to the I-set of some correct processor, all the correct processors participate in its instance, and if an id does not belong to the I-set of any correct processor, the decision is guaranteed to be 0 (i.e., to drop the id), even if some correct processors do not participate. All the correct processors end up with the same set of ids (that contains the ids of all the correct processors), then each processor decides on the rank of its own id in this set.

The algorithm and the formal proof of its correctness can be found in [18].

3.2 Fast Byzantine Renaming

The renaming algorithm presented in the previous section relies on an underlying Byzantine agreement protocol. However, it is well known that renaming is "easier" than consensus. For example, in the crash failure case, synchronous wait-free renaming can be performed in $O(\log n)$ rounds [9] (consensus requires $\Omega(n)$ rounds), and it can also be solved asynchronously in the presence of up to $\lfloor (n-1)/2 \rfloor$ failures [3] (consensus is not solvable asynchronously even if a single crash failure is possible). These observations suggest that it would be interesting to find a Byzantine renaming algorithm that does not use Byzantine agreement.

Fig. 4 presents an $O(\log n)$ round renaming algorithm that does not rely on Byzantine agreement. As in [9] and several shared memory algorithms [15,5], the idea is to split the processors into smaller and smaller groups. Every processor starts with an empty string and iteratively extends this string according to the position of its identifier in the set of identifiers of all the processors whose string is equal to its own. This approach works for the crash failure case (see [9]), since eventually each group contains at most one processor. However, it is insufficient for Byzantine failures, because in some groups the faulty processors may become (an arbitrarily large) majority, in which case they can prevent the group from further splitting.

Nevertheless, there always exists at least one group which does not have this problem, so that eventually at least one correct processor p_0 chooses a unique new name. Moreover, by using the technique of *echo* messages [21], it is possible to ensure that no faulty processor is able to claim multiple new names, which in particular implies that the new name of p_0 is not shared by any faulty processor. The method used for Byzantine failures is to treat the new names as *namespaces* in which a name is assigned to every processor by the namespace owner. Renaming can be solved this way because at least one namespace is completely controlled by a correct processor.

```
Initial setup (1st round): /* initialization phase, round 1 */
send $(processorID, \lambda)$ to all
Initial setup (2nd round): /* initialization phase, round 2 */
for $1 \leq i \leq n$ do
    if $(\alpha, \lambda)$ message was received from link $i$ in round 1 then
        $lnk[i].ID := \alpha$
        send $echo(\alpha)$ to all
receive messages
for $1 \leq i \leq n$ do
    if $echo(lnk[i].ID)$ messages were received from $n - t$ distinct processors then
        $lnk[i].str := \lambda$
    else
        $lnk[i].str := \perp$
In rounds $2k + 1$, where $1 \leq k \leq K$: /* splitting phase $k$, round 1 */
let $I :=$
    $\{lnk[i].ID \,|1 \leq i \leq n \ \wedge \ lnk[i].str = lnk[n].str\}$
let $d :=$
    $\begin{cases} 1 \ if \ |I|/4 \geq rank \ (processorID) \geq 1 \\ 2 \ if \ |I|/2 \geq rank \ (processorID) > |I|/4 \\ 3 \ if \ 3|I|/4 \geq rank \ (processorID) > |I|/2 \\ 4 \ if \ rank \ (processorID) > 3|I|/4 \end{cases}$
append to $lnk[n].str$: (i) $d$; (ii) the 7 most significant digits in the decimal representation of $|I|$
send $(processorID, lnk[n].str)$ to all
receive messages
for $1 \leq i \leq n$ do
    if $(lnk[i].ID, \sigma)$ was received from link $i$ and $\sigma$ is a legal string for phase $k$ and $lnk[i].str$ is
    a prefix of $\sigma$ then
        $lnk[i].str := \sigma$
    else
        $lnk[i].str := \perp$
In rounds $2k + 2$, where $1 \leq k \leq K$: /* splitting phase $k$, round 2 */
for $1 \leq i \leq n$ do
    send $echo(lnk[i].ID, lnk[i].str)$ to all
receive messages
for $1 \leq i \leq n$ do
    if $echo(lnk[i].ID, lnk[i].str)$ were received from $< n - t$ distinct processors then
        $lnk[i].str := \perp$
In round $2K + 3$: /* decision phase, round 1 */
for $1 \leq i \leq n$ do
    send $(processorID, lnk[i].ID, lnk[n].str, i)$ to all
In round $2K + 4$: /* decision phase, round 2 */
for $1 \leq i \leq n$ do
    if $(lnk[i].ID, \gamma, lnk[i].str, j)$ was received via link $i$ and there is no $1 \leq i' \leq n$ s.t. $lnk[i'].ID \neq$
    $lnk[i].ID \ \wedge \ lnk[i'].str = lnk[i].str$ then
        send $echo(lnk[i].ID, \gamma, lnk[i].str, j)$ to all
In round $2K + 5$: /* decision phase, round 3 */
for $1 \leq i \leq n$ do
    if $echo(lnk[i].ID, \gamma, lnk[i].str, j)$ were received from $n - t$ distinct processors and there are no
    $\gamma' \neq \gamma$ and $1 \leq i' \leq n$ s.t. $echo(lnk[i'].ID, \gamma', lnk[i].str, j)$ were received from $n - t$ distinct
    processors then
        send $ACK(lnk[i].ID, \gamma, lnk[i].str, j)$ to all
receive messages
let $1 \leq i_0 \leq n$ be a minimal number s.t. $ACK(lnk[i_0].ID, processorID, lnk[i_0].str, j)$ were
received from $n - t$ distinct processors
decide on $(lnk[i_0].str, j)$
```

Fig. 4. A fast Byzantine renaming algorithm

The renaming algorithm in Fig. 4 consists of an initialization phase (the first two rounds), $O(\log n)$ *splitting* phases (each taking two rounds), and a final phase (the last three rounds) in which a processor allocates new names in its namespace (computed during the splitting phases) for all the processors in the system (round $2K + 3$), and then picks some "legal" namespace together with the name that it was assigned inside that namespace (round $2K + 5$).

In Fig. 4 we denote an empty string by λ, and the $NULL$ string, which by definition cannot be a prefix of any other string, by $\perp$. For convenience, it is assumed that the links are labeled $1, ..., n$, where link n is a self-loop. The number of splitting phases is denoted by $K = O(\log n)$ (the exact value follows from the proof). $rank_I(\alpha)$ denotes the *rank* of an id α in the set I, i.e., the place of α in a list of elements in I sorted in ascending order.

For a string σ, let $C_k(\sigma)$ denote the set of (identifiers of) the correct processors whose $lnk[n].str$ variable at the end of phase $k \geq 0$ (i.e., at the end of round $2k+2$), is σ. Note that the variable $lnk[n].str$ holds the string that the processor has chosen for itself (so far), which follows from the convention that the nth link is a self loop.

In addition to $C_k(\sigma)$, we define the set $F_k(\sigma)$ of the faulty identifiers that correspond to the string σ. Formally, an identifier β belongs to $F_k(\sigma)$ iff one of the following two conditions is satisfied:

(i) $\beta \notin C_k(\sigma)$ and there exists a correct processor p and $1 \leq i \leq n-1$ s.t. $(lnk[i].ID)_p = \beta$ and $(lnk[i].str)_p = \sigma$ at the end of phase k.

(ii) $\beta \in C_k(\sigma)$ and there exist $n-2t$ correct processors $p_1, ..., p_{n-2t}$ s.t. $\forall\, 1 \leq j \leq n-2t$, $\exists i_1, i_2$ with $(lnk[i_1].ID)_p \ = (lnk[i_2].ID)_p \ = \beta$ and $(lnk[i_1].str)_p \ = (lnk[i_2].str)_p \ = \sigma$.

Intuitively, $F_k(\sigma)$ contains all the ids that are associated with the string σ which do not belong to any correct processor in $C_k(\sigma)$, together with ids of the correct processors associated with σ that have a faulty duplicate that is also associated with σ.

Below we sketch the proof of the correctness of the renaming algorithm in Fig. 4. A formal proof appears in Appendix A.

The first step is to show that $|F_0(\lambda)| < 2t$, which implies that from the beginning the number of faulty ids that satisfy (i) or (ii) is smaller than the number of the correct processors. Since in the following stages of the algorithm both the correct and the faulty ids are only splitted and no new ids can be introduced, the above inequality implies that for any $1 \leq k \leq K$ there exists a string σ such that $|C_k(\sigma)| > |F_k(\sigma)|$.

Next we show that a group $C_k(\sigma)$ of correct processors that satisfies $|C_k(\sigma)| > |F_k(\sigma)|$ is splitted in the $(k+1)$ phase into at least two groups (unless it is already of size 1). Moreover, when $|C_k(\sigma)|$ is higher than some (constant) threshold, each of the groups into which $C_k(\sigma)$ splits is smaller than $C_k(\sigma)$ by some constant fraction. This result implies that after $K = O(\log n)$ splitting phases, there exists a string σ_0, such that $C_K(\sigma_0) = \{\alpha_0\}$ and $F_K(\sigma_0) = \emptyset$.

Based on the above, the correctness of the algorithm can be proved as follows: First we show that two correct processors cannot decide on the same new name. Then we show that since there exists at least one legal namespace from which new names can be chosen (the namespace σ_0, controlled by correct processor with id α_0), every correct processor is able to decide on a new name. Finally, since in each of the $O(\log n)$ splitting phases the strings are extended by a constant number of symbols, the target namespace is of size $O(\text{poly}(n))$.

3.3 Asynchronous Byzantine Renaming

In [3], which introduced the renaming problem for the asynchronous message passing environment with crash failures, the basic algorithm operates by continuously exchanging all the ids that each processor has discovered so far. It is shown that eventually the exchanged vectors of ids must converge to *stable* vectors, i.e., vectors shared by the majority of the processors. Since the stable vectors are totally ordered by the inclusion relation, they can be used for renaming: a new name which consists of the size of a stable vector and of the rank of processor's original id in a stable vector is guaranteed to be unique.

In the Byzantine failure case it is a common practice to use *echo* messages to verify every message received [21]. In our case, if an id is added to the vector only after it was acknowledged by at least $n - t$ *echo* messages from distinct processors, the basic renaming algorithm of [3] is transformed into an asynchronous algorithm that tolerates $\lfloor (n-1)/3 \rfloor$ Byzantine failures.

In [3], two more complex renaming algorithms are presented: a renaming algorithm with a target namespace of size $n+t$, and an order preserving renaming algorithm. Both algorithms rely on the fact that processors do not cheat in a more subtle ways than the basic algorithm. Thus, these algorithms cannot be made Byzantine fault tolerant in a similarly straightforward manner.

4 Conclusions

This paper considered the renaming problem in a totally connected synchronous network. It was shown that when faulty processors are able to falsify their names (the $\mathcal{M}_L$ model), the problem cannot be solved in an a priori bounded number of rounds, for $n + (n \bmod 3) \leq 3t$. For the case of faulty processors that cannot cheat about their names (the $\mathcal{M}_I$ model), this bound implies that renaming cannot be solved when $n + (n\ mod\ 4) \leq 2t$. We also presented three algorithms for solving the Byzantine renaming problems for $n > 3t$, each using a different well known paradigm. One of the algorithms works in the asynchronous model as well.

The Byzantine renaming problem offers a number of interesting open questions. The most important one is to find the maximal number of failures that can be tolerated by a renaming algorithm. This mainly concerns the $\mathcal{M}_I$ model, since the bound that was obtained in Section 2 for $\mathcal{M}_L$ is almost tight (a gap of one fault remains in case $n \equiv 2 \bmod 3$). Another direction is to find fast algorithms with a small (possibly even linear) size of the target namespace.

The above questions are also interesting in the asynchronous case of the Byzantine renaming problem. An additional open question in this case is the size of the minimum possible target namespace. We note that in the synchronous system the last question can be settled for $n > 3t$ by exploiting the possibility of consensus, in which case for both the $\mathcal{M}_I$ and the $\mathcal{M}_L$ models the new namespace can be of size n (the best possible).

Acknowledgements

This research was supported in part by Israeli Council for Higher Education, the Ministry of Defense and by a grant from Dr. and Mrs. Silverston, Cambridge, UK.

References

1. Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *PODC*, pages 91–103, 1999.
2. Yehuda Afek and Michael Merritt. Fast, wait-free $(2k-1)$-renaming. In *PODC*, pages 105–112, 1999.
3. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
4. Hagit Attiya and Arie Fouren. Polynominal and adaptive long-lived $(2k$-1)-renaming. In *DISC*, pages 149–163, 2000.
5. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
6. Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.
7. Amotz Bar-Noy and Danny Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Mathematical Systems Theory*, 26(1):21–39, 1993.
8. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In *PODC*, pages 41–51, 1993.
9. Soma Chaudhuri, Maurice Herlihy, and Mark R. Tuttle. Wait-free implementations in message-passing systems. *Theor. Comput. Sci.*, 220(1):211–245, 1999.
10. Edsger W. Dijkstra. On weak and strong termination. *Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag*, pages 355–357, 1982.
11. John R. Douceur. The Sybil attack. In *IPTPS*, pages 251–260, 2002.
12. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
13. Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
14. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
15. Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.
16. Michael Okun. Agreement among unacquainted Byzantine generals. In *DISC*, pages 499–500, 2005.
17. Michael Okun and Amnon Barak. Efficient algorithms for anonymous Byzantine agreement. To appear in *Theory Comput. Syst.*
18. Michael Okun and Amnon Barak. On anonymous Byzantine agreement. *Leibniz Center TR 2004-2, The Hebrew University*, 2004.
19. Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
20. Michel Raynal. An introduction to the renaming problem. In *PRDC*, pages 121–124, 2002.
21. T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

A Algorithm for Fast Byzantine Renaming

Below we prove the correctness of the renaming algorithm in Fig. 4. The proof proceeds as described in Section 3.2. For convenience we often refer to a correct processor directly by its id. To denote the id of the correct processor to which a variable belongs subscripts are used.

Lemma 1. $|F_0(\lambda)| < 2t$.

Proof. Recall that from the definitions in Section 3.2 it follows that $C_0(\lambda)$ is the set of ids of all the correct processors. Consider some $\beta \in F_0(\lambda)$. If $\beta \notin C_0(\lambda)$ then there exists a correct processor α such that at the end of the second round $(lnk[i].ID)_\alpha = \beta$ and $(lnk[i].str)_\alpha = \lambda$. This is possible iff α received $n - t$ $echo(\beta)$ messages, i.e., at least $n - 2t$ correct processors received (β, λ) message in the first round. If $\beta \in C_0(\lambda)$, then in the first round at least $n - 2t$ correct processors received (β, λ) message from at least two different links.

In both cases β accounts for $n-2t$ links between correct and faulty processors (in the 1st round). The total number of such links is $(n - t)t$. Thus $|F_0(\lambda)| \leq (n - t)t/(n - 2t) < 2t$, where the last inequality follows from $n > 3t$. □

Since $|C_0(\lambda)| = n - t$, the above lemma in particular implies $|C_0(\lambda)| > |F_0(\lambda)|$.

Lemma 2. *For every $K \geq k \geq 0$, a string σ which is valid for phase k and a string $\sigma\tau$ ($\sigma\tau$ denotes the concatenation of strings σ and τ) valid for phase $k + 1$, it holds that $\cup_\tau C_{k+1}(\sigma\tau) = C_k(\sigma)$ and $\cup_\tau F_{k+1}(\sigma\tau) \subseteq F_k(\sigma)$.*

Proof. For most cases the property follows directly from the definitions of the sets together with the fact that all the string variables in the algorithm are modified only by appending of new suffixes. The only non trivial case is when some $\beta \in F_{k+1}(\sigma\tau)$ s.t. $\beta \notin C_{k+1}(\sigma\tau)$ belongs to $C_k(\sigma)$. In this case there exists a correct processor α s.t. at the end of phase $k + 1$ $(lnk[i].str = \sigma\tau)_\alpha$ and $(lnk[i].ID = \beta)_\alpha$ (for some i). This is possible iff α received in round $2(k+1)+2$ $n-t$ $echo(\sigma\tau, \beta)$ messages, which implies that during phase $k+1$ there are $n-2t$ correct processors that have two channels associated with the id β, one of which is also associated with the string $\sigma\tau$. Thus $\beta \in F_k(\sigma)$. □

Together, Lemma 1 and Lemma 2 imply that for every k there exists a string σ such that $|C_k(\sigma)| > |F_k(\sigma)|$.

The following property follows directly from the definitions and the algorithm:

Property 1. If $\alpha \in C_k(\sigma)$, then the set I_α computed by the correct processor α in the beginning of phase $k + 1$ satisfies $C_k(\sigma) \subseteq I_\alpha \subseteq C_k(\sigma) \cup F_k(\sigma)$.

Lemma 3. *If $10^6 \geq |C_k(\sigma)| > \max\{|F_k(\sigma)|, 1\}$, then $|C_k(\sigma)| > |C_{k+1}(\sigma\tau)|$ for any non-empty string τ.*

Proof. Property 1 implies that when the sizes of the sets $C_k(\sigma)$ and $F_k(\sigma)$ are bounded by 10^6, the size of the set I computed in the beginning of phase $k + 1$ (see line 12 in Fig. 4) is entirely expressed by the digits appended to the string (line 14). If the size of the set I is not the same among all the processors in $C_k(\sigma)$, then the lemma follows immediately.

Otherwise, let $\alpha_1 = \min C_k(\sigma)$ and $\alpha_2 = \max C_k(\sigma)$. Let $S = |C_k(\sigma)|$ and let $S + s = |I_{\alpha_1}| = |I_{\alpha_2}|$, where I_{α_1} and I_{α_2} are the I sets computed by α_1 and α_2 in phase $k+1$. Property 1 implies $S > s$.

To prove the lemma, consider first the case $S-1 > s$. In this case, in I_{α_1} there are $S-1$ elements higher than α_1 (namely $C_k(\sigma) \setminus \{\alpha_1\}$), and at most s elements lower than α_1. It follows that in phase $(k+1)$ $d_{\alpha_1} \leq 2$. At the same time, in I_{α_2} there are $S-1$ elements lower than α_2 (namely $C_k(\sigma) \setminus \{\alpha_2\}$), and at most s elements higher than α_2. Therefore $d_{\alpha_2} \geq 3$, which completes the proof for this case.

In the second case $S - 1 = s$. It follows that $I_{\alpha_1} = C_k(\sigma) \cup F_k(\sigma)$, since otherwise $|C_k(\sigma)| > |F_k(\sigma)|$ does not hold. Similarly $I_{\alpha_2} = C_k(\sigma) \cup F_k(\sigma)$. Since the distance between α_1 and α_2 is s, while $|C_k(\sigma) \cup F_k(\sigma)| = 2s+1$, α_1 and α_2 cannot belong to the same quarter of $C_k(\sigma) \cup F_k(\sigma)$, i.e., $d_{\alpha_1} < d_{\alpha_2}$. □

Lemma 4. *If $|C_k(\sigma)| > |F_k(\sigma)|$ and $|C_k(\sigma)| \geq 32$, then for any non-empty string τ it holds that $31\,|C_k(\sigma)|\,/32 \geq |C_{k+1}(\sigma\tau)|$.*

Proof. As before, let S denote the size of the set $C_k(\sigma)$. Suppose the lemma is incorrect, i.e., there exists τ_0 such that $|C_{k+1}(\sigma\tau_0)| > 31S/32$. Observe that Property 1 implies that for any $\alpha \in C_k(\sigma)$ the I-set computed in phase $k+1$ satisfies $2S > |I_\alpha| \geq S$. It follows that for any $\alpha \in C_{k+1}(\sigma\tau_0)$ $S + s + \epsilon S > |I_\alpha| \geq S + s$, where $s = \min_{\alpha \in C\ _{+1}(\sigma\tau_0)} |I_\alpha| - S$, and ϵ depends on the number of the most significant digits of the size of the I-set that the algorithm appends to the string. When 7 digits in decimal representation are appended (line 14), $\epsilon < 10^{-6}$.

Consider two possible cases:

(i) $s \leq 7S/8$. In this case, for any $\alpha \in C_{k+1}(\sigma\tau_0)$ $|I_\alpha|/2 < (15/16 + \epsilon/2)S < 31S/32$. It follows that the processor with the highest id in $C_{k+1}(\sigma\tau_0)$ assigns in phase $k+1$ its d variable the value 3 or 4. The processor with the lowest id in $C_{k+1}(\sigma\tau_0)$ assigns in phase $k+1$ its d variable the value 1 or 2, which is a contradiction.

(ii) $s > 7S/8$. In this case, for any $\alpha, \alpha_1, \alpha_2 \in C_{k+1}(\sigma\tau_0)$ it holds that

$$\left| rank_{I\ _1}(\alpha) - rank_{I\ _2}(\alpha) \right| < S/8 + \epsilon S. \tag{4}$$

W.l.o.g. suppose that in phase $k+1$ all the processors in $C_{k+1}(\sigma\tau_0)$ choose to assign their d variable the value 2, i.e., for any $\alpha \in C_{k+1}(\sigma\tau_0)$ $|I_\alpha|/2 \geq rank_{I\ }(\alpha) > |I_\alpha|/4$, which implies $(S + s + \epsilon S)/2 \geq rank_{I\ }(\alpha) > (S+s)/4$. By applying (4) we get that for any $\alpha, \alpha_0 \in C_{k+1}(\sigma\tau_0)$ it holds that $(S+s+\epsilon S)/2 + S/8 + \epsilon S \geq rank_{I\ _0}(\alpha) > (S+s)/4 - S/8 - \epsilon S$. This implies $\forall \alpha \in C_{k+1}(\sigma\tau_0)$ $34S/32 + 3\epsilon S/2 \geq rank_{I\ _0}(\alpha) > 11S/32 - \epsilon S$, which is an obvious contradiction (a set of size $31S/32$ cannot have its ranks in an interval of length smaller than $31S/32$). □

Lemma 5. *No two correct processors end up with the same new name.*

Proof. Assume by contradiction that two correct processors, α_1 and α_2 decide on the same name (σ, j). It follows that there exist β, β' such that in round $2K+5$ processor α_1 received the message $ACK(\beta, \alpha_1, \sigma, j)$ from $n-t$ distinct processors, and processor α_2 received the message $ACK(\beta', \alpha_2, \sigma, j)$ from $n-t$ distinct processors. Therefore there must exist a correct processor that in round $2K+5$ sends both these messages. This is a contradiction, since the rule for sending ACK messages (lines 34, 35) forbids it. □

Lemma 6. *If by the end of the K splitting phases $F_K(\sigma) = \emptyset$ and $C_K(\sigma) = \{\alpha_0\}$, then every correct processor decides on a new name.*

Proof. Consider the correct processor α_0. In round $2K+3$ α_0 sends (to all) a message $(\alpha_0, \alpha, \sigma, j)$, for any id α that belongs to a correct processor (typically j is the label of the link which connects α_0 to α, but it can also be a label of a link between α_0 and a faulty processors that presents itself as α). Since $F_K(\sigma) = \emptyset$, the condition on line 31 implies that all the correct processors send in round $2K+4$ an *echo*$(\alpha_0, \alpha, \sigma, j)$ message to all. Next consider the last round. If all the correct processors send an $ACK(\alpha_0, \alpha, \sigma, j)$ message, the proof is complete. Otherwise, there exists a correct processor that does not send $ACK(\alpha_0, \alpha, \sigma, j)$, despite receiving *echo*$(\alpha_0, \alpha, \sigma, j)$ messages from $n-t$ distinct processors. From the condition on line 34 it follows that there exist $n-2t$ correct processors that in round $2K+4$ sent an *echo*$(\beta, \gamma', \sigma, j)$ message, where $\gamma' \neq \alpha$, in addition to the *echo*$(\alpha_0, \alpha, \sigma, j)$ message. This is impossible if $\beta \neq \alpha_0$ (see the condition on line 31). However, if $\beta = \alpha_0$ each one of these $n-2t$ correct processors must have two different links associated with the id α_0 and the string σ, i.e., $\alpha_0 \in F_K(\sigma)$. This contradicts the assumption. □

Theorem 3. *The algorithm in Fig. 4 solves the Byzantine renaming problem in model $\mathcal{M}_L$ for $n > 3t$ in $O(\log n)$ rounds, with target namespace of $O(poly(n))$ size.*

Proof. By Lemmas 1, 2, 3 and 4 it follows that after appropriately chosen $K = O(\log n)$ splitting phases, there exists some σ such that $C_K(\sigma) = \{\alpha_0\}$, and $F_K(\sigma) = \emptyset$. For such a K Lemma 5 and Lemma 6 imply that renaming is achieved.

In every splitting phase the string of each processor grows by a constant number of bits. Thus, the strings are $O(\log n)$ bits long. Since the final name consists of one such string and a number between 1 and n, the new namespace is of size $O(\text{poly}(n))$. (To get the best possible size, one needs to optimize the parameters in Lemmas 3 and 4.) □

Built-In Coloring for Highly-Concurrent Doubly-Linked Lists
(Extended Abstract)

Hagit Attiya and Eshcar Hillel

Department of Computer Science, Technion

Abstract. This paper presents a novel approach for lock-free implementations of concurrent data structures, based on dynamically maintaining a *coloring* of the data structure's items. Roughly speaking, the data structure's operations are implemented by acquiring virtual locks on several items of the data structure and then making the changes atomically; this simplifies the design and provides clean functionality. The virtual locks are managed with CAS or DCAS primitives, and helping is used to guarantee progress; virtual locks are acquired according to a coloring order that decreases the length of waiting chains and increases concurrency. Coming back full circle, the legality of the coloring is preserved by having operations correctly update the colors of the items they modify.

The benefits of the scheme are demonstrated with new nonblocking implementations of doubly-linked list data structures: A DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere, and CAS-based implementations in which removals are allowed only at the ends of the list (insertions can occur anywhere).

The implementations possess several attractive features: they do not bound the list size, they do not leave accessible chains of garbage nodes, and they allow operations to proceed concurrently, without interfering with each other, if they are applied to non-adjacent nodes in the list.

1 Introduction

Many core problems in asynchronous multiprocessing systems revolve around the coordination of access to shared resources and can be captured as *concurrent data structures*—abstract data structures that are concurrently accessed by asynchronous processes. A prominent example is provided by list-based data structures: A *double-ended queue* (*deque*) supports operations that insert and remove items at the two ends of the queue; it can be used as a producer-consumer job queue [3]. A *priority queue* can be implemented as a doubly-linked list where removals are allowed only at the ends, while items can be inserted anywhere at the queue; it can be used to queue process identifiers for scheduling purposes. Finally, a generic *doubly-linked list* (hereafter, often called simply a *linked list*) allows insertions and removals anywhere in the linked list.

Concurrent data structures are implemented by applying *primitives*—provided by the hardware or the operating system—to memory locations. *Lock-free implementations* do not rely on mutual exclusion, thereby avoiding the inherent problems associated with locking—deadlock, convoying, and priority-inversion. Lock-free implementations must

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 31–45, 2006.

rely on strong primitives [15], e.g., CAS (*compare and swap*) and its multi-location variant, kCAS.

Lock-free implementations are often complex and hard to get right; even for relatively simple, key data structures, like deques, they suffer from significant drawbacks: Some implementations may contain garbage nodes [14], others statically limit the data structure's size [16] or do not allow concurrent operations on both ends of the queue [21]. Even when DCAS (i.e., 2CAS) is used, existing implementations either are inherently sequential [11, 12] or allow to access chains of garbage nodes [9].

Implementing concurrent data structures is fairly simple if an arbitrary number of locations can be accessed atomically. For example, removing an item from a doubly-linked list is easy if one can atomically access three items—the item to be removed and the two items before and after it (cf. [9]).

Since no multiprocessor supports primitives that access more than two locations atomically, it is necessary to simulate them in software using CAS or DCAS. This can be done using methods such as *software transactional memory* [22] or the so-called *locking without blocking* techniques [25,7]. The basic idea of these methods is to use CAS in order to acquire *virtual* locks on the items—one item at a time, and *help* processes that hold virtual locks on desired items until they are released. This guarantees that the simulation is *nonblocking* [15], namely, in any infinite execution, some pending operation completes within a finite number of steps. Unfortunately, the resulting implementations may have long waiting chains, creating interference among operations and reducing the implementation's throughput.

Attiya and Dagan [4] suggest an alternative implementation of binary operations that reduces interference by using *colors* (from a small set). This *color-based virtual locking* scheme starts by legally coloring the items it is going to access, so that neighboring items have distinct colors. Then, the algorithm acquires the virtual locks in increasing order of colors, thereby avoiding long waiting chains. Afek et al. [1] extended this implementation to arbitrary k-ary operations.

To evaluate whether operations that access disjoint parts of the data structure, or are widely separated in time, do not interfere with each other, Afek et al. [1] define two measures. These definitions rely on the familiar notion of a *conflict graph*, whose nodes are the data items and there is an edge between two items if they are accessed by the same operation. Roughly speaking, the *distance* between operations in the conflict graph is the length of the shortest path between their data items. An implementation has *d-local step complexity* if only operations in distance less than or equal to d in the conflict graph can delay each other; it has *d-local contention* if only operations in distance less than or equal to d in the conflict graph can access the same locations simultaneously.[1] In particular, when there is no path in the conflict graph between the data items accessed by two operations, they do not delay each other or access the same memory locations; thus, d-local step complexity and contention extend and generalize *disjoint-access parallelism* [19].

The implementations [1,4] have $O(\log^* n)$-local step complexity and contention, and they are rather complicated, making them infeasible for fundamental linked list-based

[1] Attiya and Dagan [4] used a more complicated measure called *sensitivity*, which is not discussed in this extended abstract.

data structures. The major reason for the cost and complication of these implementations is the need to color memory locations at the beginning of each operation, since operations access arbitrary and unpredictable sets of memory locations.

When operations are applied on a specific data structure, however, they access its constituent items in a predictable, well-organized manner; e.g., linked list operations access two or three consecutive items. In this case, why color the accessed items from scratch, each time an operation is invoked? After all, the implementation initializes the data structure and provides operations that are the only means for manipulating it. If the colors are built into the items, then an operation can rely on them to guide its locking order, without coloring them first. In return, the operation needs to guarantee that the modifications it applies to the data structure preserve the legality of the items' coloring.

We demonstrate this approach with two new doubly-linked list algorithms: A CAS-based implementation in which removals are allowed only at the ends of the list (and insertions can occur anywhere), and a DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere.

The CAS-based implementation, allowing insertions anywhere and removals at the ends, is based on a 3-coloring of the linked list items. It has 4-local contention and 4-local step complexity.; namely, an operation only contends with operations on items close to its own items on the linked list, and it is delayed only due to such operations. When insertions are also limited to occur at the ends, the analysis can be further refined to show 2-local contention and 2-local step complexity; this means that operations at the two ends of a deque containing three data items (or more) never interfere with each other.

Handling removals from the middle of the linked list is more difficult: removing an item might entail recoloring one of its neighbors, requiring to make sure its neighbor's color is not changed concurrently. Thus, a remove operation has to lock *three* consecutive items; under a legal coloring it is possible that two of these items (necessarily non-consecutive) have the same color. We employ a DCAS operation to lock these two nodes atomically, thereby avoiding hold-and-wait chains. This algorithm has 6-local contention and 2-local step complexity. To the best of our knowledge, this is the first nonblocking implementation of a doubly-linked list from realistic primitives, which allows insertions and removals anywhere in the list, and has low interference.

In our algorithms, an operation has constant *obstruction-free step complexity* [10]; namely, an operation completes within $O(1)$ steps in an execution suffix in which it is running solo. Another attractive feature of our implementations is that it does not leave accessible chains of stale "garbage" nodes.

In recent years, a flurry of papers proposed implementations of dynamic linked list data structures, yet none of them provided all the features of our algorithms.

Harris [14] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously removed from the linked list, possibly yielding an unbounded chain of uncollected garbage nodes. Michael [20] handled these memory management issues. Elsewhere [21], Michael proposed an implementation of a deque; in his algorithm, a single word (called *anchor*) holds the head and tail pointers, causing all operations to interfere with each other, thereby making the implementation inherently sequential.

Sundell and Tsigas [24] avoid the use of a single anchor, allowing operations on the two ends to proceed concurrently. They extend the algorithm to allow insertions and removals in the middle of the list [23]; in the latter algorithm, a long path of overlapping removals may cause interference among distant operations; moreover, during intermediate states, there can be a consecutive sequence of inconsistent backward links, causing part of the list to behave as singly-linked. An *obstruction-free* deque, providing a liveness property weaker than nonblocking, was proposed by Herlihy et al. [16]; besides blocking when there is even a little contention, this array-based implementation bounds the deque's size.

Greenwald [11, 12] suggests to use DCAS to simplify the design of implementations of many data structures. His implementations of deques, singly-linked and doubly-linked lists synchronize via a single designated memory location, resulting in a strictly sequential execution of operations. Agesen et al. [2] present the first DCAS-based nonblocking, dynamically-sized deque implementation that supports concurrent access to both ends of the deque, and has 1-local step complexity; this algorithm does not allow insertions or removals in the middle of the linked list. The SNARK algorithm [8] is an attempt for further improvement that uses only a single DCAS primitive per operation in the best case, instead of two. Unfortunately, SNARK is incorrect [9]; the corrected version allows removed nodes to be accessed from within the deque, thus preventing the garbage collector from reclaiming long chains of unused nodes. Doherty et al. [9] even argue that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and efficient nonblocking implementations of concurrent data structures.

The rest of this paper is organized as follows. Section 2 presents the model of a asynchronous shared-memory system, while Section 3 defines local contention and local step complexity in a dynamic setting. Most of the paper describes the DCAS-based implementation of a doubly-linked list allowing insertions and removals anywhere (Section 4). Section 6 outlines the modifications needed to obtain the CAS-based implementation that does not allow removals from the middle. The complete code and proof of correctness for both algorithms appear in the full version of this paper [5].

2 Preliminaries

We consider a standard model for a shared memory system [6] in which a finite set of *asynchronous processes* $p_1, \ldots, p_n$ communicate by applying *primitive* operations to m shared *memory locations*, $l_1, \ldots, l_m$.

A *configuration* is a vector $C = (q_1, \ldots, q_n, v_1, \ldots, v_m)$, where q_i is the local state of p_i and v_j is the value of memory location l_j.

An *event* is a computation step by a process, p_i, consisting of some local computation and the application of a primitive to the memory. We allow the following primitives: READ(l_j) returns the value v_j in location l_j; WRITE(l_j, v) sets the value of location l_j to v; CAS$(l_j, \mathit{exp}, \mathit{new})$ writes the value *new* to location l_j if its value is equal to *exp*, and returns a success or failure flag; DCAS is similar to CAS, but operates on two independent memory locations.

An *execution interval* α is a (finite or infinite) alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \ldots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the unique initial configuration.

A *data structure* of type T supports a set of operations that provide the only means to manipulate it. Each data structure has a *sequential specification*, which indicates how it is modified when operations are applied in a serial manner (in isolation).

An *implementation* of a data structure T provides a specific data-representation for T's instances as a set of memory locations, and protocols that processes must follow to carry out T's operations, defined in terms of primitives applied to memory locations. We require the implementation to be *linearizable* [17].

This paper considers a *doubly-linked list* data structure, composed of *nodes*, each with link pointers to its left and right neighboring nodes. Two special *anchor* nodes serve as the first (leftmost) and last (rightmost) nodes in the doubly-linked list; they cannot be removed from it, and hold no left link or no right link, respectively. A node is *valid* in configuration C if it is either an anchor, or both its left link and right link pointers are not null.

We concentrate on the *InsertRight*, *InsertLeft* and *Remove* operations applied to some $source$ node in the linked list. Our description of their effects follows the description of the deque operations in [2]:

insertRight(nd). If $source$ is a valid node other than the right anchor, then insert nd to the right of $source$ and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

insertLeft(nd). If $source$ is a valid node other than the left anchor, then insert nd to the left of $source$ and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

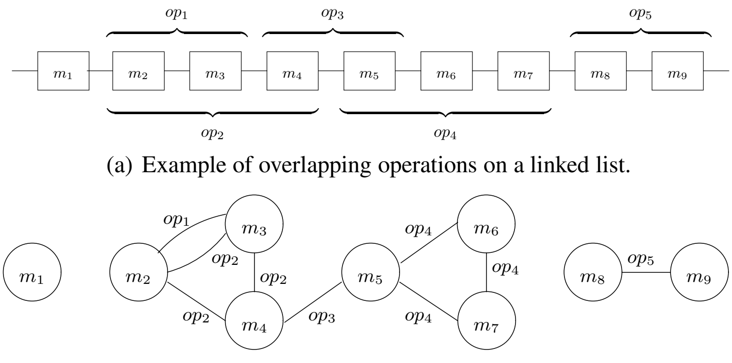
remove(). If $source$ is a valid node other than an anchor, then remove $source$ from the linked list and return SUCCESS; otherwise, return INVALID and the linked list is unchanged.

In order to apply an operation op_i to the data structure, process p_i executes the associated protocol. The *interval of an operation op*, denoted I_{op}, is the execution interval between the first and last events of the process executing *op*'s protocol; if the operation does not terminate, its interval is infinite. Two operations *overlap* if their intervals overlap. The *interval of a set of operations OP*, denoted I_{OP}, is the minimal execution interval that contains all intervals, $\{I_{op}\}_{op \in OP}$.

3 Locality Properties

The *reference lock-based implementation* of a data structure T atomically locks all the memory locations that it accesses; these are called the *lock set* of the operation. The lock set of an operation op_i applied in state s is denoted $\mathcal{LS}_s(op_i)$. Different lock-based implementations may have different lock sets. Since we aim for highly concurrent implementations, we choose a reference implementation that locks as few data items as possible; for a linked list data structure this number is a constant.

When operations are concurrent, the state of the data structure at a configuration C is not necessarily unique. A state s of the data structure is *possible* in configuration C,



(a) Example of overlapping operations on a linked list.

(b) The corresponding conflict graph $G(C)$.

Fig. 1. A simple conflict graph

if it is the result of some linearization that includes all operations that complete before C and a subset of the operations that are pending in C. The set of all possible states in C is denoted *state*(C).

Intuitively, the data set of an operation includes all the data items the operation accesses. When the data structure is dynamic, however, the data set changes over time and it is unknown when the operation is invoked. For this reason, we need to consult the reference implementation regarding the data items it locks with respect to all the states of the data structure during the operation's interval. Formally, the *data set* of an operation op_i in configuration C is defined as $\mathcal{DS}_C(op_i) = \bigcup_{s \in state(C)} \mathcal{LS}_s(op_i)$, i.e., the union of all the sets of data items the operation locks (under the reference implementation) when the state of the data structure is in $state(C)$. $\mathcal{DS}(op_i) = \bigcup_{C \in I} \quad \mathcal{DS}_C(op_i)$; namely, the union of $\mathcal{DS}_C(op_i)$ over all configurations during op_i's execution interval.

The *conflict graph* of a configuration C, denoted $G(C)$, occurring in some execution, is an undirected graph that captures the distance between overlapping operations. If C is in the execution interval of an operation op_i, and v and u are data items in $\mathcal{DS}_C(op_i)$, then the conflict graph includes an edge between the respective vertices m_v and m_u, labeled op_i. The conflict graph of an execution interval α is the graph $\bigcup_{C \in \alpha} G(C)$. For example, Figure 1(a) depicts the data set of several overlapping operations; op_1, op_3, and op_5 insert a new node to the right of m_2, m_4, and m_8, respectively, while op_2 and op_4 remove m_3 and m_6 respectively. Figure 1(b) depicts the corresponding conflict graph; the new node, omitted from the figure, is also in the operation's data set.

The *conflict distance* (in short *distance*) between two operations, op_i, op_j, in a conflict graph is the length (in edges) of the shortest path between some vertex m_i in $\mathcal{DS}(op_i)$ and some (possibly the same) vertex m_j in $\mathcal{DS}(op_j)$. In particular, if $\mathcal{DS}(op_i)$ intersect $\mathcal{DS}(op_j)$, then the distance between op_i and op_j is zero. The distance is ∞, if there is no such path. In the conflict graph of Figure 1(b), the distance between op_1 and op_2 is zero, the distance between op_1 and op_3 is one, the distance between op_1 and op_4 is two, and the distance between op_1 and op_5 is ∞.

We use this dynamic version of a conflict graph in the definitions of locality measures suggested by Afek et al. [1]:

Definition 1. *An algorithm has* d-local step complexity *if the number of steps performed by process* p *during the operation interval* I_{op} *is bounded by a function of the number of operations at distance smaller than or equal to* d *from op in the conflict graph of its operation interval* I_{op}.

Definition 2. *An algorithm has* d-local contention *if in every execution interval for any two operations,* $I_{\{op_1,op_2\}}$, op_1 *and* op_2 *access the same memory location only if their distance in the conflict graph of* $I_{\{op_1,op_2\}}$ *is smaller than or equal to* d.

4 DCAS-Based Doubly-Linked List Algorithm

We demonstrate our approach with a nonblocking implementation, DCAS-CHROMO, of a doubly-linked list with insertions and removals anywhere. At the heart of our methodology is an enhancement of the colored-based virtual locking scheme. We first review this scheme, and then describe our algorithm.

The Color-Based Virtual Locking Scheme: Data structures can be implemented by the nonblocking *virtual locking* scheme [7,22,25]. A concurrent implementation is systematically derived from any lock-based algorithm: an operation starts by acquiring *virtual locks* on the data items in its data set (LOCK phase); then, the appropriate changes are applied on these data items (APPLY phase); finally, the operation releases the virtual locks (UNLOCK phase). Similar to a lock-based solution, while a data item is locked by an operation, other operations can neither lock nor modify it. This means the algorithm is relieved of handling inconsistent states due to contention.

An operation is *blocked* if a data item in its data set is locked by another, *blocking* operation. In order to make the scheme nonblocking, the process executing the blocked operation *op* helps the blocking operation *op*$'$ to complete and release its data set. Several processes may execute an operation; the process that invokes the operation is its *initiator*, while the *executing processes* are processes helping the initiator to complete or the initiator itself. CAS primitives are used to guarantee that only one of the executing processes performs each step of the operation, and others have no effect.

This scheme induces *recursive* helping, in which one process helps another process to help a third process and so on, possibly causing long helping chains. For example, assume the nodes in Figure 1(a) are locked in ascending order. Consider an execution α in which op_2, op_3 and op_4 concurrently lock their left-most data items successfully, and then op_1 tries to lock its data items while the other operations are delayed. Since m_2 is locked by op_2, op_1 has to help op_2; since m_4 is locked by op_3, op_1 has to help op_3; and since m_5 is locked by op_4, op_1 has to help op_4. Thus op_1 is delayed by operations on a path in α's conflict graph, from some vertex in $\mathcal{DS}(op_1)$. In general, op_1 can be delayed by any operation within finite distance from it, implying that the locality is high.

Shavit and Touitou [22] overcome this problem by helping only an immediate neighbor in the conflict graph. Nevertheless, the number of steps a process performs depends on the length of the longest path from its data set in the conflict graph. Consider again

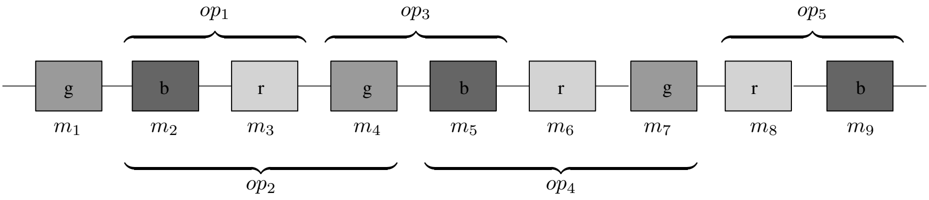


Fig. 2. 3-Coloring of the linked list in Figure 1(a)

an execution that starts with op_2, op_3 and op_4 locking their low-address data items successfully, then op_1 fails to lock m_2, op_2 fails to lock m_4, and op_3 fails to lock m_5; each operation then helps its (immediate) neighbor. Prior to helping, op_2 and op_3, relinquish their locks and fail, thus op_1 and op_2 discover their help is unnecessary. Assume that op_4 completes, and again op_1, op_2 and op_3 try to lock their data sets. It is possible that op_2 and op_3 lock their low-address data items, and op_1 tries, in vain, to help op_2, which releases its locks due to op_3, etc. As the length of the path of overlapping operations increases, the number of times op_1 futilely helps op_2 increases as well.

A *color-based* virtual locking scheme [4] bounds the length of helping chains by *M-coloring* the data items with an ordered set of colors, $c_1 < c_2 < \ldots < c_M$. An operation acquires locks on data items in an increasing order of colors; after it locks all c_i-colored data items, we say the operation *locked color* c_i. In this scheme, op helps op' only if op' already locked a higher color.

Figure 2 presents a 3-coloring of the linked list in Figure 1(a) using the colors $r(red) < g(green) < b(blue)$. Assume op_3 locks m_4 and then tries to lock m_5, with color *b*. If the lock on m_5 is already held by op_4, then op_3 has to help op_4. Note however, that *b* is the largest color, which means that op_4 already locked all the nodes in data set. This means that op_3 will only have to apply op_4's changes, and op_3 is not required to recursively help additional operations. Along these lines, it is possible to prove that the length of helping chains is bounded by the number of colors, M, and the number of times an operation helps other operations is bounded by a function of the number of operations within distance M [4].

Originally [1, 4], colors were assigned to nodes from scratch each time an operation starts. This is done in a DECISION phase, which obtains information about operations (and their data sets) at non-constant distance; thus, the DECISION phase has non-constant locality properties.

Our Approach: We achieve constant locality properties by employing two complementary algorithmic ideas: The first is to maintain the data structure legally colored at all times, and the second is to atomically lock all data items with the same color.

The key idea of our approach is to keep the coloring legal while the operation is in its APPLY phase, rendering the DECISION phase obsolete. That is, the colors are built into the nodes, and the operation updates the colors so that nodes remain legally colored.

These changes are limited to the nodes in the operation's data set, and bypass the need to re-compute a legal coloring from scratch each time an operation is invoked.

The second idea avoids long helping chains due to symmetric color assignments. For example, consider a long legally colored linked list of nodes with alternating colors: $b, r, b, r, b, r, \ldots$. Assume a set of concurrent operations, each of which is trying to remove a different r-colored node, by first locking the node and its two b-colored neighbors. An implementation that locks these two b-colored nodes one at a time, e.g., first the left neighbor, can lead to a configuration in which an operation holds its left lock, and needs to help all operations to its right.

It is tempting to extend the notion of a legal coloring and require that any triple of neighboring nodes is assigned distinct colors. This certainly will allow to follow the color-based virtual locking scheme, but how can we preserve this extended coloring property? In particular, when a node is removed, it is necessary to lock *four* nodes in order to legally re-color the remaining three nodes; this requires to further extend the coloring property to any four consecutive nodes, which in turn requires to lock *five* consecutive nodes and so on.

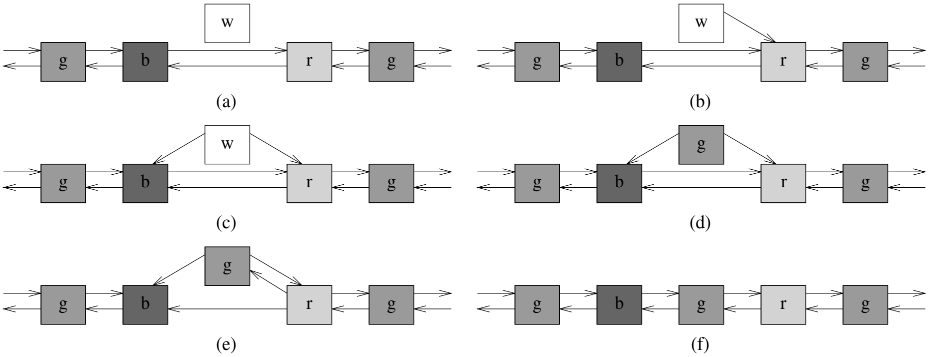
Locking equally-colored nodes atomically provides an escape from this vicious circle, by avoiding this situation altogether. An operation accesses at most three consecutive nodes, which are legally colored, thus at most two of these nodes have the same color, and a DCAS suffices for locking them. For example, in the scenario described above, locking the two b-colored nodes atomically breaks the symmetry. This guarantees that the LOCK phase has $O(1)$-local step complexity.

Another aspect of our algorithm is in handling the complications due to dynamically-changing data structures. Previous implementations of the virtual locking scheme handle static transactions [22] and multi-location operations [1, 4]; in both cases, an operation accesses a pre-determined static data set.

Our algorithm addresses this problem, in a manner similar to [13], using a *data set memento*, which holds a view of the data set when the operation starts. If, while locking, a node and its memento are inconsistent, the operation skips the APPLY phase to the UNLOCK phase where it releases all the locks it holds. If, on the other hand, the operation completes its LOCK phase, then the locked data set memento is consistent with the operation data set and the operation can continue with the APPLY phase as in a static virtual locking scheme.

Detailed Description of Algorithm DCAS-CHROMO*:* First, we describe how operations apply their changes to the data structure, and give some intuition of how the legal coloring is preserved; then we describe the helping mechanism that is responsible for the nonblocking and locality properties.

The lock-based implementation we use as a reference has the following lock sets: An *InsertRight* operation locks the new node to be inserted, the $source$ node (to which the operation is applied) and its right neighbor; an *InsertLeft* operation is symmetric; a *Remove* operation locks the $source$ node and both its left and right neighbors. After locking, the operations apply changes to the respective set of left and right links as described by the following code:



(a)

(b)

(c)

(d)

(e)

(f)

Fig. 3. An example of an *InsertRight* operation - op_1 in Figure 2

```
InsertRight::applyChanges() {
    $newNode$.right ← $source$.right
    $newNode$.left ← $source$
    $source$.right.left ← $newNode$
    $source$.right ← $newNode$
}
```

```
Remove::applyChanges() {
    $source$.left.right ← $source$.right
    $source$.right.left ← $source$.left
    $source$.right ← $\perp$
    $source$.left ← $\perp$
}
```

Since our algorithm employs a virtual locking scheme, each operation proceeds in exclusion in a manner similar to the lock-based one. Our implementation, however, also needs to maintain the nodes legally colored. This requires adding one step to the *InsertRight* operation (see Figure 3), and two steps to the *Remove* operation (see Figure 4). To ensure that the coloring is legal at all times, we use a temporary color $c_0 < c_1$ during the algorithm as described bellow. In the example figures, c_0 is $w(white)$.

***InsertRight* operation.** Figure 3(a) presents the nodes m_1, m_2, m_3, m_4 from Figure 2, and the new node, m, that op_1 inserts to the right of m_2. Before m is inserted to the linked list, it is colored with the temporary color, w. op_1 locks the nodes in its data set, m_2 and m_3 (and effectively, also m), and then applies its changes as follows: update right neighbor of m (Figure 3(b)); update left neighbor of m—now, m is legally colored, since its neighbors m_2 and m_3 have colors different than w (Figure 3(c)); m is assigned with a non-temporary color different than its neighbors m_2 and m_3 (Figure 3(d)); update left neighbor of m_3 (Figure 3(e)); update right neighbor of m_2 (Figure 3(f)).

***Remove* operation.** Figure 4(a) presents the nodes m_1, m_2, m_3, m_4, m_5 from Figure 2, op_2 removes the node m_3. op_2 locks the nodes in its data set, m_2, m_3 and m_4, before it applies its changes as follows: m_4 is assigned with the temporary color, w—now, m_4 is legally colored, since its neighbors m_3 and m_5 have colors different than w (Figure 4(b)); update right neighbor of m_2 (Figure 4(c)); update left neighbor of m_4 (Figure 4(d)); set right and left neighbors of m_3 to null (Figure 4(e)); m_4 is assigned with a non-temporary color different than its neighbors m_2 and m_5 so it is legally colored (Figure 4(f)).

Both an *InsertRight* operation and a *Remove* operation access three consecutive nodes in the data set, however each operation only changes the color of a single node.

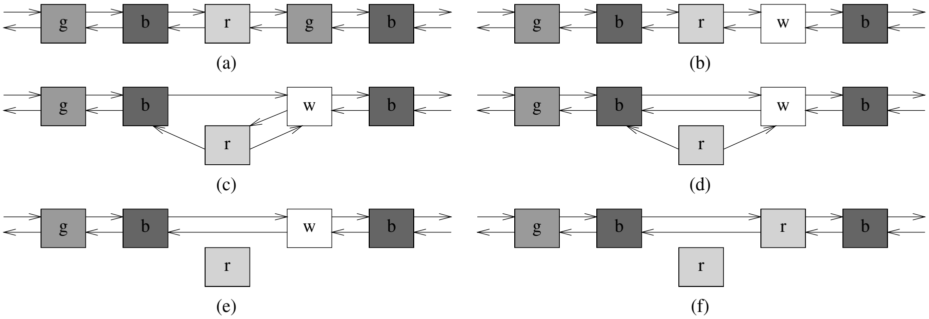


Fig. 4. An example of a *remove* operation - op_2 in Figure 2

An *InsertRight* operation changes the color of the middle node, and a *Remove* operation changes the color of the right node. The color of the left node in the data set of an operation is not modified. This ensures that no two adjacent nodes change their color concurrently even if they belong to the data sets of two adjacent concurrent operations.

We now detail the color-based locking and helping mechanisms. An operation is partitioned into *invocations*. To initiate an invocation, the initiator process generates the operation's data set memento, which traces inconsistencies in the data set due to changes applied by concurrent operations. If the operation locks its data set and applies its changes then the invocation *completes successfully* and the operation will not be re-invoked. Otherwise, the invocation *fails* and the operation restarts a new invocation.

The state of an operation is a tuple ⟨seq,phase,result⟩: *seq* is an integer, initially 0, incremented every time the operation fails and the initiator process reinvokes it; *phase* indicates the locking scheme phase within the invocation, set to INIT at the beginning of every invocation; *result* holds the result of the current invocation execution, set to NULL at the beginning of every invocation.

Figure 5 shows the state transition diagram of an operation's invocation. The dashed line indicates re-invocation, increasing the sequence number of the operation. The state transitions of an invocation in a best-case execution, encountering no contention, appear at the top. If an operation discovers, while initiating an invocation, that another operation removed the source node then it need not apply its changes, and it skips to the FINAL phase with an INVALID result; this operation will not be re-invoked. If an operation discovers that a node in its data set other than the source node is invalid, then the operation needs to re-evaluate its data set. In such a case, the invocation fails and a new invocation is restarted. Another scenario in which an invocation fails is if the operation detects inconsistency with the data set memento while locking the data set. In this case, the operation releases the locks it already acquired and restarts a new invocation.

When an operation *op* fails to lock color *c* it may discover that a node in its data set is locked by another, blocking operation op'. In such a case, we follow the standard recursive helping mechanism, i.e., *op* helps op'. Before helping op', the executing process of *op* verifies (again) that the nodes are consistent with their mementos. This is crucial for maintaining the locality properties of the algorithm. If after an operation fails to lock the nodes it discovers that none of them is locked by another operation, it simply retries to

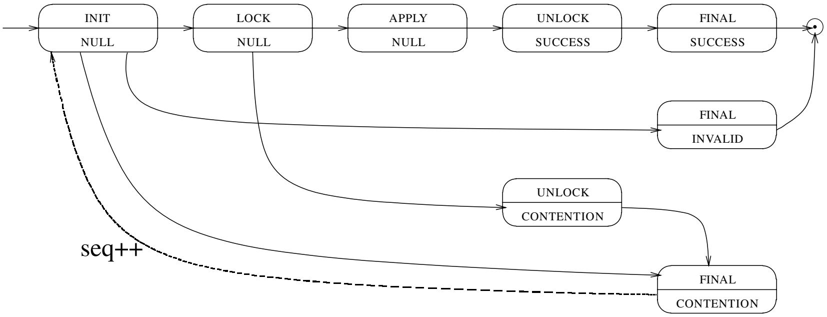


Fig. 5. Diagram of an operation state transitions model; the lower part of the state is the value of *result*

acquire their locks. Finally, when an operation discovers that its source node is invalid (as described above), it helps the operation that removes this node before skipping to its FINAL phase, to preserve the correct order in which the operations complete.

Since an operation may be invoked more than once, its execution is composed of an alternating sequence of acquiring and releasing locks. Having more than one process executes the operation requires special care. Specifically, a process may acquire locks of previous invocations or release locks acquired in a later invocation. Together with the CAS primitives, the state is used to synchronize between the executing processes of an operation. Before acquiring a lock the process verifies that the operation's sequence number is equal to the invocation it is executing. Furthermore, to prevent a process from releasing locks acquired in a later invocation, the operation stamps any lock it acquires with its sequence number. Before a process releases a lock, it verifies that the sequence number stamped on the lock is equal to the invocation it is executing.

Some Implementation Details: We use object-oriented terminology and define operations as objects, whose structure and behavior are defined in the *Operation* hierarchy.

A process initializes an operation object with all the data required for its execution, specifically the source node from the linked list on which the operation is applied. Algorithm 1. outlines the generic protocol for an operation execution. The execution starts with the execute method (line ex1) and as long as it suffers from contention and is unable to complete, the process repeatedly tries to re-invoke the operation (lines ex3-ex4): First it generates the new data set memento (line t5); then it "helps" itself to follow the locking scheme (line t7); lock nodes in its data set (line h2), apply its changes (line h4), and releas the data set (line h6). Concrete operations, such as *InsertRight* and *Remove*, extend the *Operation* structure and refine its protocols for cloning and manipulating the data set with respect to their specifications. (The full pseudocode appears in [5].)

It is well-known that CAS primitives suffer from the ABA problem [18]: a process p may read a value A from some memory location l, then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed. The simplest way to avoid this problem is to associate each attribute with a monotonically increasing counter. The attribute and the counter are manipulated

Algorithm 1. Algorithm DCAS-CHROMO: Execution outline

```
Result Operation::execute() {
    do
        initiate new invocation
        try()
    while state.result = CONTENTION
    return state.result
}
```

```
Operation::try() {
    if source is invalid then
        helpBlocking(source.lock)
        transition to FINAL-INVALID state
    clone data set
    transition to LOCK state
    help(state.seq)
    transition to FINAL state
}
```

```
Operation::help(int seq) {
    lock data set // by ascending colors
    if state.phase = APPLY then
        apply changes
        transition to UNLOCK state
    unlock data set
}
```

```
Operation::helpBlocking(Lock lock) {
    if lock != ⊥ then
        op, opseq ← get blocking info
        op.help(opseq)
}
```

atomically; the counter is incremented whenever the attribute is updated. Assuming that the counter has enough bits, the CAS succeeds only if the counter has not changed since the process read the attribute. Other methods prevent the ABA problem without the use of a per-attribute counters, and may be applied also to our algorithm.

It is assumed that an automatic garbage collection reclaims unreferenced objects such as nodes and operation objects. Long chains of garbage and garbage cycles do not form since the links of removed nodes are nullified. The ABA prevention counter allows a removed node to be inserted into a linked list immediately (after setting its color to c_0) without harming the correctness of the algorithm. However, this would violate the local contention property of the algorithm, so it is assumed that once a node is removed from one linked list it is not reused until reclaimed by the garbage collector.

5 Correctness Proof (Outline)

The safety properties of the implementation, and in particular, its linearizability, hinge on showing that the executing processes preserve the correct transition of the operation between phases—locking, changing and releasing nodes in accordance with the operations' phases. Most importantly, items in the data set are changed only while all of them are locked. As mentioned before, this is somewhat more complicated than in previous work [1,4,7,22,25], since the data set is dynamic.

Proving the progress and locality properties is more involved. One key is to show that the color of an item causing a blocked operation to help, increases with every recursive call. This implies that the depth of the recursion is bounded by the number of colors, M. Moreover, we argue that in every locking attempt of an executing process, may it be a successful or a futile one, some "nearby" operation makes progress, ensuring that the algorithm is nonblocking and that the step complexity of an operation depends only on the number of operations in its close neighborhood.

The detailed correctness proof appears in the full version of the paper [5], showing:

Theorem 1. *Algorithm* DCAS-CHROMO *is a nonblocking implementation of a doubly-linked list, allowing insertions and removals anywhere, with 2-local step complexity and 6-local contention complexity.*

6 CAS-Based Doubly-Linked List Algorithm

In this section we discuss Algorithm CAS-CHROMO, a CAS-based variation of Algorithm DCAS-CHROMO, allowing insertions everywhere and removals only at the ends.

We reuse the core implementation of insert and remove operations from Algorithm DCAS-CHROMO and add the operations *InsertFirst*, *RemoveFirst*, *InsertLast* and *RemoveLast* for manipulating the ends of the linked list, with the obvious functionality.

We discuss the operations applied on the first (left) end of the linked list; the two operation on the last (right) end are symmetric. *InsertFirst* and *RemoveFirst* operations are closely related to the *InsertRight* and *Remove* operations, except that they implicitly take the left anchor as their source node. The most crucial modification is in the locking protocol, which no longer uses a DCAS primitive when locking its data set. However, nodes with the same color are locked according to their order in the list, from left to right; this allows to prove that the algorithm is nonblocking. In fact, this can also show that operations help only along paths with $O(1)$ length, which can be used to prove that the algorithm has good locality properties. The details of the algorithm, as well as its correctness proof, appear in the full version of the paper [5].

Theorem 2. *Algorithm* CAS-CHROMO *is a nonblocking implementation of a doubly-linked list, allowing insertions anywhere and removals at the ends, with 4-local step complexity and 4-local contention complexity.*

An implementation of deque data structure requires operations only at the ends. In this case, the analysis can be further improved to show that the algorithm has 2-local step complexity and 2-local contention complexity.

7 Discussion

This paper presents a new approach for designing nonblocking and high-throughput implementations of linked list data structures; our scheme may have other applications, e.g., for tree-based data structures.

We show a DCAS-based implementation of insertions and removals in a doubly-linked list; when nodes are removed only from the ends, the implementation is modified to use only CAS. These implementations are intended only as a proof-of-concept and leave open further optimizations. It is also necessary to implement a *search* mechanism in order to support the full functionality of priority queues and lists.

Acknowledgments. We thank David Hay, Danny Hendler, Gadi Taubenfeld and the referees for helpful comments.

References

1. Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. PODC 1997, pp. 111–120..
2. O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS-based concurrent deques. *Theory Comput. Syst.*, 35(3):349–386, 2002.
3. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.
4. H. Attiya and E. Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5):1013–1037, 2001.
5. H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. Available from `http://www.cs.technion.ac.il/~hagit/publications/`, 2006.
6. H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations and Advanced Topics*. John Wiley& Sons, second edition, 2004.
7. G. Barnes. A method for implementing lock-free shared-data structures. SPAA 1993, pp. 261–270.
8. D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. DISC 2000, pp. 59–73.
9. S. Doherty, D. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS is not a silver bullet for nonblocking algorithm design. SPAA 2004, pp. 216–224.
10. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free step complexity: Lock-free dcas as an example (brief announcement). DISC 2005, pp. 493–494.
11. M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
12. M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. PODC 2002, pp. 260–269.
13. T. Harris and K. Fraser. Language support for lightweight transactions. OOPSLA 2003, pp. 388–402.
14. T. Harris. A pragmatic implementation of non-blocking linked-lists. DISC 2001, pp. 300–314.
15. M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
16. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. ICDCS 2003, pp. 522–529.
17. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
18. IBM. *IBM System/370 Extended Architecture, Principle of Operation*, 1983. IBM Publication No. SA22-7085.
19. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. PODC 1994, pp. 151–160.
20. M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002, pp. 73–82. ACM Press.
21. M. Michael. CAS-based lock-free algorithm for shared deques. Euro-Par 2003, pp. 651–660.
22. N. Shavit and D. Touitou. Software transactional memory. *Dist. Comp.*, 10(2):99–116, 1997.
23. H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
24. H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. OPODIS 2004, pp. 240–255.
25. J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. PODS 1992, pp. 212–222.

Fault-Tolerant and Self-stabilizing Mobile Robots Gathering

— Feasibility Study —

Xavier Défago[1,⋆], Maria Gradinariu[2],
Stéphane Messika[3], and Philippe Raipin-Parvédy[2,4]

[1] School of Information Science, JAIST, Ishikawa, Japan
defago@jaist.ac.jp
[2] IRISA/Université de Rennes 1, France
mgradina@irisa.fr
[3] LRI/Université Paris Sud, France
messika@lri.fr
[4] France Telecom R&D, France
philippe.raipin@orange-ft.com

Abstract. Gathering is a fundamental coordination problem in cooperative mobile robotics. In short, given a set of robots with arbitrary initial location and no initial agreement on a global coordinate system, gathering requires that all robots, following their algorithm, reach the exact same but not predetermined location. Gathering is particularly challenging in networks where robots are oblivious (i.e., stateless) and the direct communication is replaced by observations on their respective locations. Interestingly any algorithm that solves gathering with oblivious robots is inherently self-stabilizing.

In this paper, we significantly extend the studies of deterministic gathering feasibility under different assumptions related to synchrony and faults (crash and Byzantine). Unlike prior work, we consider a larger set of scheduling strategies, such as bounded schedulers, and derive interesting lower bounds on these schedulers. In addition, we extend our study to the feasibility of probabilistic gathering in both fault-free and fault-prone environments. To the best of our knowledge our work is the first to address the gathering from a probabilistic point of view.

1 Introduction

Many applications of mobile robotics envision groups of mobile robots self-organizing and cooperating toward the resolution of common objectives. In many cases, the group of robots is aimed at being deployed in adverse environments, such as space, deep sea, or after some natural (or unnatural) disaster. It results that the group must self-organize in the absence of any prior infrastructure (e.g., no global positioning), and ensure coordination in spite of faulty robots and unanticipated changes in the environment.

⋆ Work supported by MEXT Grant-in-Aid for Young Scientists (A) (Nr. 18680007).

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 46–60, 2006.

The *gathering problem*, also known as the *Rendez-Vous* problem, is a fundamental coordination problem in cooperative mobile robotics. In short, given a set of robots with arbitrary initial location and no initial agreement on a global coordinate system, gathering requires that all robots, following their algorithm, reach the exact same location—one not agreed upon initially—within a *finite* number of steps, and remain there.

Similar to the Consensus problem in conventional distributed systems, gathering has a simple definition but the existence of a solution greatly depends on the synchrony of the systems as well as the nature of the faults that may possibly occur. In this paper, we investigate some of the fundamental limits of deterministic and probabilistic gathering in the face of different synchrony and fault assumptions.

To study the gathering problem, we consider a system model first defined by Suzuki and Yamashita [1], and some variants with various degrees of synchrony. In this model, robots are represented as points that evolve on a plane. At any given time, a robot can be either idle or active. In the latter case, the robot observes the locations of the other robots, computes a target position, and moves toward it. The time when a robot becomes active is governed by an activation daemon (scheduler). In the original definition of Suzuki and Yamashita, called the ATOM model, activations (i.e., look–compute–move) are atomic, and the scheduler is assumed to be fair and distributed, meaning that each robot is activated infinitely often and that any subset of the robots can be active simultaneously. In the CORDA model of Prencipe [2], activations are completely asynchronous, for instance allowing robots to be seen while moving.

Suzuki and Yamashita [1] proposed a gathering algorithm for non-oblivious robots in ATOM model. They also proved that gathering can be solved with three or more oblivious robots, but not with only two.[1] Prencipe [3] studied the problem of gathering in both ATOM and CORDA models. He showed that the problem is impossible without additional assumptions such as being able to detect the multiplicity of a location (i.e., knowing the number of robots that may simultaneously occupy that location). Flocchini *et al.* [4] proposed a gathering solution for oblivious robots with limited visibility in CORDA model, where robots share the knowledge of a common direction as given by some compass. Based on that work, Souissi *et al.* [5] consider a system in which compasses are not necessarily consistent initially. Ando *et al.* [6] propose a gathering algorithm for ATOM model with limited visibility. Cohen and Peleg [7] study the problem when robots' observations and movements are subject to some errors.

None of the previously mentioned works addressed the gathering feasibility in fault-prone environments. One of the first steps in this direction was done by Agmon and Peleg [8]. They prove that gathering of correct robots (referred in this paper *weak gathering*) can be achieved in the ATOM model even in the

[1] With two robots, all configurations are symmetrical and may lead to robots endlessly swapping their positions. In contrast, with three or more robots, an algorithm can be made such that, at each step, either the robots remain symmetrical and they eventually reach the same location, or symmetry is broken and this is used to move one robot at a time.

face of the crash of a single robot. Furthermore, they prove that no deterministic gathering algorithm exists in ATOM model that can tolerate a Byzantine[2] robot. Finally, they consider a stronger daemon, called fully synchronous, in which all robots are always activated simultaneously, and show that weak gathering can be solved in that model when the number of Byzantine robots is less than one third of the system.

Contribution. In this paper, we further study the limits of gathering feasibility in both fault-free and fault prone environments, by considering centralized schedulers[3] (i.e., activations in mutual exclusion) and k-bounded schedulers, that is, schedulers ensuring that between any two consecutive activations of a robot, no other robot is activated more than k times.

The main results we obtain are as follows. Firstly, we strengthen the impossibility results of Agmon and Peleg [8] since we show that, even in strictly stronger models, their impossibility result holds. Secondly, we outline the essential limits where Byzantine and crash-tolerant gathering become possible. In particular, we propose interesting lower bounds on the value that k (the scheduler bound) must take for the problem to become possible. Thirdly, we show in what situations randomized algorithms can help solve the problem, and when they cannot. To the best of our knowledge our work is the first to study the feasibility of probabilistic gathering in both fault-free and fault-prone systems. Additionally we evaluate the convergence time of our probabilistic gathering algorithms under fair schedulers using the coupling technique developed in [9]. The convergence time of our algorithms is polynomial in the size of the network in both fault-free and crash-prone environments under fair bounded schedulers. We conjecture that our bounds are optimal and hold for the case of Byzantine-prone systems.

Structure of the paper. The rest of the paper is structured as follows. Section 2 describes the robots network and system model. Section 3 formally defines the gathering problem. Section 4 propose possibility and impossibility results for deterministic and probabilistic gathering in fault-free environments. Section 5.1 and 5.2 extend the study in Section 4 to crash and Byzantine prone environments. Due to space limitations, most of the proofs are omitted, but they are included in the full version [10].

2 Model

2.1 Robots Networks

Most of the notions presented in this section are borrowed from [1,2,8]. We consider a network of a finite set of robots arbitrarily deployed in a geographical

[2] A Byzantine robot is a faulty robot that can behave arbitrarily, possibly in a way to prevent the other robots from gathering in a stable way.

[3] The rationale for considering a centralized daemon is that, with communication facilities, the robots can synchronize by running a mutual exclusion algorithm, such as token passing.

area. The robots are devices with sensing, computational and motion capabilities. They can observe (sense) the positions of other robots in the plane and based on these observations they perform some local computations. Furthermore, based on the local computations robots may move to other locations in the plane.

In the case robots are able to sense the whole set of robots they are referred as robots with *unlimited visibility*; otherwise robots have limited visibility. In this paper, we consider that robots have unlimited visibility.

In the case robots are able to distinguish if there are more than one robot at a given position they are referred as robots with *multiplicity knowledge*.

2.2 System Model

A network of robots that exhibit a discrete behaviour can be modeled with an I/O automaton [11]. A network of robots that exhibit a continous behaviour can be modeled with a hybrid I/O automaton [12]. The actions performed by the automaton modeling a robot are as follows:

- *Observation (input type action).*
  An observation returns a snapshot of the positions of all the robots in the visibility range. In our case, this observation returns a snapshot of the positions of all the robots;
- *Local computation (internal action).*
  The aim of a local computation is the computation of a destination point;
- *Motion (output type action).*
  This action commands the motion of robots towards the destination location computed in the local computation action.

The local state of a robot at time t is the state of its input/output variables and the state of its local variables and registers. A network of robots is modeled by the parallel composition of the individual automata that model one per one the robots in the network. A configuration of the system at time t is the union of the local states of the robots in the system at time t. An execution $e = (c_0, \ldots, c_t, \ldots)$ of the system is an infinite sequence of configurations, where c_0 is the initial configuration[4] of the system, and every transition $c_i \rightarrow c_{i+1}$ is associated to the execution of a subset of the previously defined actions.

Schedulers. A scheduler decides at each configuration the set of robots allowed to perform their actions. A scheduler is fair if, in an infinite execution, a robot is activated infinitely often. In this paper we consider the fair version of the following schedulers:

- *centralized*: at each configuration a single robot is allowed to perform its actions;
- *k-bounded*: between two consecutive activations of a robot, another robot can be activated at most k times;

[4] Unless stated otherwise, this paper makes no specific assumption regarding the respective positions of robots in initial configurations.

- *bounded regular*: between two consecutive activations of a robot, all the robots in the system perform their actions once and only once.
- *arbitrary*: at each configuration an arbitrary subset of robots is activated.

Faults. In this paper, we address the following failures:

- *crash failures*: In this class, we further distinguish two subclasses: (1) robots physically disappear from the network, and (2) robots stop all their activities, but remain physically present in the network;
- *Byzantine failures*: In this case, robots may have an arbitrary behavior.

2.3 Computational Models

The literature proposes two computational models: ATOM and CORDA. The ATOM model was introduced by Suzuki and Yamashita [1]. In this model each robot performs, once activated by the scheduler, a *computation cycle* composed of the following three actions: observation, computation and motion. The atomic action performed by a robot in this model is a computation cycle. The execution of the system can be modeled as an infinite sequence of rounds. In a round one or more robots are activated and perform a computation cycle. The ATOM model was refined by Agmon and Peleg [8]. The authors distinguish the case of hyperactive systems where all robots are activated simultaneously and non-hyperactive systems where a strict subset of robots are simultaneously activated.

The CORDA model was introduced by Prencipe [2]. This model refines the atomicity of the actions performed by each robot. Hence, robots may perform in a decoupled fashion, the atomic actions of a computation cycle. They may be interrupted by the scheduler in the middle of a computation cycle. Moreover, while a robot performs an action A, where A can be one of the following atomic actions: observation, local computation or motion, another robot may perform a totally different action B.

In this paper, we consider both models, refined with the scheduling strategies presented above. Moreover, we consider that robots are oblivious (i.e., stateless). That is, robots do not conserve any information between two computational cycles.[5] We also assume that all the robots in the system have unlimited visibility.

3 The Gathering Problem

A network of robots is in a *terminal (legitimate) configuration* with respect to the gathering requirement if all the robots share the same position in the plane. Let denote by $\mathcal{P}_{Gathering}$ this predicate.

An algorithm solves the gathering problem in an oblivious system if the following two properties are verified:

[5] One of the major motivation for considering oblivious robots is that, as observed by Suzuki and Yamashita [1], any algorithm designed for that model is inherently self-stabilizing.

- **Convergence** Any execution of the system starting in an arbitrary configuration reaches in a finite number of steps a configuration that satisfies $\mathcal{P}_{Gathering}$.
- **Termination** Any execution starting in a terminal configuration with respect to the $\mathcal{P}_{Gathering}$ predicate contains only legitimate configurations.

Gathering is difficult to achieve in most of the environments. Therefore, weaker forms of gathering were studied so far. An interesting version of this problem requires robots to *converge* toward a single location rather than reach that location in a finite time. The convergence is however considerably easier to deal with. For instance, with unlimited visibility, convergence can be achieved trivially by having robots moving toward the barycenter of the network [1].

Note that an algorithm that solves the gathering problem with oblivious or stateless robots is self-stabilizing [13].

4 Gathering in Fault-Free Environments

In this section, we refine results showing the impossibility of gathering [3,8] by proving first that these results hold even under more restrictive schedulers than the ones considered so far [3,8]. Interestingly, we also prove that some of these impossibility results hold even in probabilistic settings. Additionally, to circumvent these impossibility results, we propose a probabilistic algorithm that solves the fault-free gathering in both ATOM and CORDA models, under a special class of schedulers, known as k-bounded schedulers. In short, a k-bounded scheduler is one ensuring that, during any two consecutive activations of any robot, no other robot is activated more than k times.

4.1 Synchronous Robots – ATOM Model

Note 4.1. Prencipe [3] proved that there is no deterministic algorithm that solves gathering in ATOM and CORDA models without additional assumptions, such as the ability to detect multiplicity.

The following lemma shows that the impossibility result of Prencipe [3] holds even under a weaker scheduler—the centralized fair bounded regular scheduler. Intuitively, a schedule of this particular scheduler is characterized by two properties: each robot is activated infinitely often and between two executions of a robot every robot in the network executes its actions exactly once. Moreover, in each configuration a single robot is allowed to execute its actions.

Lemma 4.1. *There is no deterministic algorithm that solves gathering in the ATOM model for $n \geq 3$ under a centralized fair bounded regular scheduler, without additional assumptions (e.g., multiplicity knowledge).*

Note that the deterministic gathering of two oblivious robots was proved impossible by Suzuki and Yamashita [1]. The scenario is the following: the two robots

Algorithm 4.1 Probabilistic gathering for robot p.

Functions:

observe_neighbors :: returns the set of robots within visibility range of robot p (the set of p's neighbors). Note that, in a system with unlimited visibility, *observe_neighbors* returns all the robots in the network.

Actions:

```
$\mathcal{A}_1$ :: $true \longrightarrow$
        $\mathcal{N}_p = observe\_neighbors()$;
        with probability $\alpha = \frac{1}{|\mathcal{N} \bigcup\{p\}|}$ do
            select a robot $q \in \mathcal{N}_p \bigcup\{p\}$;
            move towards $q$;
        Remark: with probability $1 - \alpha$, the position remains unchanged;
```

are always activated simultaneously. Consequently, they continuously swap positions, and the system never converges. In the following, we prove that, for the case of two robots, there exists a probabilistic solution for gathering in the ATOM model, under any type of scheduler. Algorithm 4.1 describes the probabilistic strategy of a robot. When chosen by the scheduler, a robot decides, with probability α, whether it will actually compute a location and move whereas, with probability $1 - \alpha$, the robot will remain stationary. The following lemma shows that Algorithm 4.1 reaches a terminal configuration with probability 1.

Lemma 4.2. *Algorithm 4.1 probabilistically solves the 2-gathering problem in the ATOM model under an arbitrary scheduler. The algorithm converges in 2 steps in expectation.*

The next lemma extends the impossibility result proved in Lemma 4.1 to probabilistic algorithms under unfair schedulers.

Lemma 4.3. *There is no probabilistic algorithm that solves the n-gathering problem, for $n \geq 3$, in ATOM model, under a fair centralized scheduler without additional assumptions (e.g., multiplicity knowledge).*

The key issue leading to the above impossibility is the freedom that the scheduler has in selecting a robot r until its probabilistic local computation allows r to actually move. The scenario can however no longer hold with systems in which the scheduler is k-bounded. That is, in systems where a robot cannot be activated more than k times before the activation of another robot. In this type of game robots win against the scheduler and the system converges to a terminal configuration.

Lemma 4.4. *Algorithm 4.1 probabilistically solves the n-gathering problem, $n \geq 3$, in the ATOM model under a fair k-bounded scheduler and without multiplicity knowledge.*

Lemma 4.5. *The convergence time of Algorithm 4.1 under fair bounded schedulers is n^2 rounds*[6] *in expectation.*

Proof. In the following, we use the coupling technique developed in [9]. Algorithm 4.1 can be seen as a Markov chain. Let's call it $\mathcal{A}$ hereafter. A coupling for Algorithm 4.1, is a Markov chain $(X_t, Y_t)_{t=1}^{\infty}$ with the following properties: (1) each of the variables (X_t), (Y_t) is a copy of the Markov chain $\mathcal{A}$ (given initial configurations $X_0 = x$ and $Y_0 = y$); and (2) if $X_t = Y_t$ then $X_{t+1} = Y_{t+1}$. Intuitively, the coupling time is the expected time for the two processes X_t and Y_t to reach the agreement property ($X_t = Y_t$). As shown in Theorem 1 [9] the coupling time is also an upper bound for the hitting time or convergence time of a self-stabilizing algorithm.

Assume (X_t) and (Y_t) are two copies of the Markov chain modeling Algorithm 4.1. Let us denote by $\delta(X_t, Y_t)$ the distance between X_t and Y_t (the number of robots that do not share identical positions in X_t and Y_t). In the worst case, $\delta(X_t, Y_t) = n$ (where n is the number of robots in the network). In the following we show that, with positive probability, the distance between X_{t+1} and Y_{t+1} decreases. Assume that the scheduler chooses robot p at instant t, and assume that p does not share the same position in X_t and Y_t. With positive probability, $X_{t+1}(p) = Y_{t+1}(p)$. Assume that the scheduler chooses two or more robots in t. Since the scheduler is bounded, within a round of size R, $\delta(X_{t+R}, Y_{t+R}) \leq \delta(X_t, Y_t) - 1$. Following the result proved in Theorem 2 [9], the coupling time for this chain is bounded from above by $\frac{B}{1-\beta}$. Where B is the maximal value of the distance metric (in our case this value is n) and β is the constant such that for all (X_t, Y_t) we have $E[\delta(X_{t+1}, Y_{t+1})] \leq \beta\delta(X_t, Y_t)$. In our case, $\beta = \frac{n-1}{n}$. So, the hitting (convergence) time for Algorithm 4.1 is n^2 rounds in expectation. □

4.2 Asynchronous Robots – CORDA Model

In the following, we analyze the feasibility of gathering in a stronger model, namely, CORDA. Obviously, all the impossibility results proved in the ATOM model hold for CORDA [14].

The next lemma states that 2-gathering, while probabilistically feasible in ATOM model, is impossible in the CORDA model under an arbitrary scheduler.[7] We recall that, in the CORDA model, robots can be interrupted by the scheduler during a computation cycle.

Lemma 4.6. 2*-gathering is impossible in the CORDA model under an arbitrary scheduler.*

Now, instead of an arbitrary scheduler, we consider a k-bounded scheduler, and obtain the following possibility result.

[6] A round is the longest fragment of an execution between two successive actions of the same process. Following the variant of the chosen k-bounded scheduler a round can have k steps or kn steps.

[7] Note that 2-gathering is trivially possible under a centralized scheduler.

Lemma 4.7. *Algorithm 4.1 probabilistically solves the n-gathering problem, $n \geq 2$, in the CORDA model under a k-bounded scheduler and without multiplicity knowledge.*

5 Fault Tolerant Gathering

5.1 Crash Tolerant Gathering

In the following we extend the study of the gathering feasibility to fault-prone environments. In this section (n, f) denotes a system with n correct robots but f and the considered faults are the crash failures. As mentioned in the model, Section 2, in an (n, f) crash-prone system there are two types of crashes: (1) the crashed robots completely disappear from the system, and (2) the crashed robots are still physically present in the system, however they stop the execution of any action. In the sequel we analyze both situations.

Lemma 5.1. *In a crash-prone system, $(3, 1)$-gathering is deterministically possible under a fair centralized regular scheduler.*

The following lemma proves that the previous result does not hold in systems with more than three robots. More precisely, this lemma expands the impossibility results proved in Lemma 4.1 and 4.3 to crash-prone environments.

Lemma 5.2. *In a crash-prone system, there is no deterministic algorithm that solves the $(n, 1)$-gathering problem, $n \geq 4$, under a fair bounded regular centralized scheduler without additional assumptions (e.g, multiplicity knowledge).*

Lemma 5.3. *In a crash-prone system, there is no probabilistic algorithm that solves the $(n, 1)$-gathering problem, $n \geq 3$, under a fair centralized scheduler without additional assumptions (e.g., multiplicity knowledge).*

The key argument in the previous impossibility proof is that the scheduler has the possibility to choose a robot until that robot is allowed to move (by its probabilistic algorithm). In some sense, the scheduler managed to derandomize the system. However, the process of derandomization is no longer possible with a bounded scheduler. The following lemma proves that $(n, 1)$-gathering is probabilistically possible under a bounded scheduler and without additional assumptions.

Lemma 5.4. *In a crash-prone system, Algorithm 4.1 is a probabilistic solution for the gathering problem in systems with n correct robots but one and under a bounded scheduler.*

In the following, we extend our study to systems with more than one faulty robot. Hereafter, (n, f)-gathering refers to the gathering problem in a system with n correct robots but f. If the faulty robots disappear from the system, then the problem trivially reduces to the study of a fault-free gathering with n–f correct robots. In contrast, in systems where faulty robots remain physically present in the network after crashing, the problem is far from being trivial. Obviously, gathering all the robots including the faulty ones, is impossible since faulty robots may possibly have crashed at different locations.

From this point on, we study the feasibility of a weaker version of gathering, referred to as *weak gathering.* The (n, f)-*weak gathering* problem requires that, in a terminal configuration, only the *correct* robots must share the same position. The following lemma proves the impossibility of deterministic and probabilistic weak gathering under centralized bounded and fair schedulers and without additional assumptions.

Lemma 5.5. *In a crash-prone system, there is neither a probabilistic nor a deterministic algorithm that solves the* (n, f)*-weak gathering problem,* $n \geq 3$ *and* $f \geq 2$*, under a fair centralized regular scheduler without additional assumptions.*

Algorithm 5.1 Deterministic fault-tolerant weak gathering for robot p

```
Functions:
observe_neighbors :: returns the set of robots within the vision range of robot p (the
set of p's neighbors);
maximal_multiplicity :: returns a robot in the group with the maximal multiplicity;
or, if several such groups exists, makes an arbitrary choice among them;

Actions:
  $\mathcal{A}_1$ :: true $\longrightarrow$
                  $\mathcal{N}_p$ = observe_neighbors();
                  $q$ = maximal_multiplicity($\mathcal{N}_p$);
                  move towards $q$;
```

An immediate consequence of the previous lemma is the necessity of an additional assumption (e.g., multiplicity knowledge), even for probabilistic solutions under bounded schedulers.

In the sequel, we identify the conditions under which the weak gathering accepts deterministic and probabilistic solutions. Algorithm 5.1 proposes a deterministic solution for the weak gathering that works under both centralized and bounded schedulers. The idea of the algorithm is the following: a robot, once chosen by the scheduler, moves to the group with the maximal multiplicity – "attraction action". In case that all groups have the same multiplicity, the chosen robot will go to the location of another robot – "unbalanced action". The attraction action helps the convergence while the unbalanced action breaks the symmetry.

Lemma 5.6. *In a crash-prone system, Algorithm 5.1 deterministically solves the* (n, f)*-weak gathering problem,* $f \geq 2$*, under a centralized scheduler if robots are aware of the system multiplicity.*

Algorithm 5.2 Probabilistic fault-tolerant gathering for robot p with multiplicity knowledge

```
Functions:
$observe\_neighbors$ :: returns the set of robots within the vision range of robot $p$ (the set of $p$'s neighbors);
$maximal\_multiplicity$ :: returns the set of robots with the maximal multiplicity;

Actions:
  $\mathcal{A}_1$ :: $true \longrightarrow$
      $\mathcal{N}_p = observe\_neighbors()$;
      if $p \in maximal\_multiplicity(\mathcal{N}_p) \wedge |maximal\_multiplicity\ (\mathcal{N}_p)| > 1$ then
          with probability $\frac{1}{|maximal\_multiplicity(\mathcal{N}\ )|}$ do
              select a robot $q \in maximal\_multiplicity(\mathcal{N}_p)$;
              move towards $q$;
      else
          select a robot $q \in maximal\_multiplicity(\mathcal{N}_p)$;
          move towards $q$;
```

In the following we show that (n, f)-weak gathering can be solved under arbitrary schedulers using a probabilistic algorithm, Algorithm 5.2, and multiplicity knowledge. Algorithm 5.2 works as follows. When a robot is chosen by the scheduler it moves to the group with maximal multiplicity. When all groups have the same size, then the robot tosses a coin to decide if it moves or holds the current position.

Lemma 5.7. *In a crash-prone system, Algorithm 5.2 probabilistically solves the (n, f)-weak gathering problem, $f \geq 2$, under an unfair scheduler if robots are aware of the system multiplicity.*

5.2 Byzantine Tolerant Gathering

In the following we study the gathering feasibility in systems prone to Byzantine failures. In the sequel (n, f) denotes a system with n correct robots but f. Agmon and Peleg [8] proved that gathering in Byzantine environments is impossible in ATOM and CORDA models for the case $(3, 1)$. The impossibility proof is given for the case of the ATOM model and algorithms that are not hyperactive. The following lemma proves the $(3, 1)$-gathering impossibility under the weakest scheduler, in particular the centralized, fair and regular.

Lemma 5.8. *In a Byzantine-prone system, there is no deterministic algorithm that solves $(3, 1)$-weak gathering under a fair, centralized and bounded regular scheduler without additional assumptions.*

Note 5.1. Note that Algorithm 5.1 solves the Byzantine $(3, 1)$-weak gathering under a centralized regular scheduler and multiplicity knowledge. The cycle created in the impossibility proof is broken because the Byzantine robot cannot play the attractor role.

The following lemma shows that if the scheduler is relaxed, the $(3, 1)$-weak gathering becomes impossible even if robots are aware of the system multiplicity.

Lemma 5.9. *In a Byzantine-prone system, there is no deterministic algorithm that solves the* $(3, 1)$*-weak gathering, even when robots are aware of the system multiplicity, under a centralized fair* k*-bounded scheduler with* $k \geq 2$.

Note 5.2. Byzantine $(n, 1)$-weak gathering for any odd $n > 4$ is possible under any fair centralized scheduler and multiplicity knowledge. The algorithm is trivial: a robot moves to the group with maximal multiplicity.

The following lemma establishes a lower bound for the bounded centralized scheduler that prevents the deterministic gathering.

Lemma 5.10. *In a Byzantine-prone system, there is no deterministic algorithm that solves* $(n, 1)$*-weak gathering, with* $n \geq 2$ *even, under a centralized* k*-bounded scheduler for* $k \geq (n - 1)$*. This result holds even when robots are aware of the system multiplicity.*

Corollary 5.1. *Byzantine* $(n, 1)$*-weak gathering is possible under a centralized scheduler:*

- *in systems where* $n \geq 4$ *is odd, robots have multiplicity knowledge and the scheduler is fair, or*
- *in systems where* $n \geq 2$ *is even and the scheduler is* k*-bounded with* $k \leq (n - 2)$.

The following lemma states the lower bound for a bounded scheduler that prevents deterministic gathering.

Lemma 5.11. *In Byzantine-prone systems, there is no deterministic algorithm that solves* (n, f)*-weak gathering,* $f \geq 2$*, under a centralized* k*-bounded scheduler with* $k \geq \left\lceil \frac{n-f}{f} \right\rceil$ *when* n *is even, and with* $k \geq \left\lceil \frac{n-f}{f-1} \right\rceil$ *when* n *is odd, even when the robots can detect multiplicity.*

Proof.

- **Even case.** Similar to the $(n, 1)$ case above, assume that the system starts in an initial configuration in which all robots are arranged in two groups. Assume the same scheduler as in the $(n, 1)$ case: for each move of a correct robot the scheduler chooses a Byzantine robot. The Byzantine robot will try to balance the system equilibrium hence it will move towards the old location of the correct robot. In order to win the game the Byzantine robots need to move each time a correct robot moves. Since there are $n-f$ correct robots in the system, the scheduler has to be bounded by no less than $\left\lceil \frac{n-f}{f} \right\rceil$ for the Byzantine team to win.
- **Odd case.** For the odd case assume an initial configuration where robots but one (a Byzantine one) are arranged in two groups. When chosen by the scheduler the Byzantine robot not member of a group moves such that the equilibrium between the two groups does not change. Let denote G_1 and

G_2 the two groups. Consider the following schedule. Every time a correct robot, member of G_i, moves, a Byzantine robot moves as well in the opposite direction. Hence the system equilibrium does not change. The game is similar to the even case. The only difference is that the number of Byzantine robots that influence the faith of the game is $f-1$. Therefore, in order to win the game, the Byzantine team needs a k-bounded scheduler bounded by $k \geq \left\lceil \frac{n-f}{f-1} \right\rceil$. □

Lemma 5.12. *In systems with Byzantine faults, Algorithm 5.2 probabilistically solves the* (n, f)*-weak gathering,* $n \geq 3$*, problem under a bounded scheduler and multiplicity detection.*

6 Conclusion

The results presented here extend that of prior work on the possibility and impossibility of gathering in fault-free and both crash-prone and Byzantine-prone systems. For instance, we strengthen several prior impossibility results by showing that they still hold against weaker schedulers, and under various failure models. We also mark out more accurately the limit between possibility and impossibility by deriving appropriate upper and lower bounds.

To the best of our knowledge, this is actually the first study that considers probabilistic solutions to solve the gathering problem. Here, we identify conditions under which a probabilistic solution exists, as well as conditions for which not even a probabilistic solution exists.

The main results of the paper are summed up in Table 1 for fault-free systems; in Table 2 for crash-prone systems; and in Table 3 for the weak gathering problem in Byzantine-prone systems.

As an open question, some of the impossibility proofs only consider the use of randomization for determining whether a robot takes actions or not when it is activated. One can argue that using randomization in a different way may

Table 1. Summary of the main results in fault-free environments

ATOM	CORDA	mult.	no mult.	centralized	regular	k-bounded	arbitrary	unfair	Conditions	Solution	Source
●			●				●	○	–	*Impossible*	Prencipe [3] (Note 4.1)
●	○		●	●	●	○	○	○	$n \geq 3$	*No deterministic*	Lemma 4.1
●		●		○	○	○	●		$n = 2$	Probabilistic	Lemma 4.2
●	○		●	●					$n \geq 3$	*No probabilistic*	Lemma 4.3
●		○	●		○	●			$n \geq 3$	Probabilistic	Lemma 4.4
	●	●					●	○	$n = 2$	*Impossible*	Lemma 4.6
○	●	○	●		○	●			–	Probabilistic	Lemma 4.7
"●" means explicit; "○" means implicit; negative results are in italic											

Table 2. Summary of the main results in crash-prone systems

ATOM	CORDA	mult.	no mult.	centralized	regular	k-bounded	arbitrary	unfair	Conditions	Solution	Source
●		●		●	●				$n = 3,\ f = 1$	Deterministic	Lemma 5.1
●	○		●	●	●	○	○	○	$n \geq 4,\ f \geq 1$	*No deterministic*	Lemma 5.2
●	○		●	●			○	○	$n \geq 3,\ f \geq 1$	*No probabilistic*	Lemma 5.3
●		●			●	●			$f = 1$	Probabilistic	Lemma 5.4
●	○		●	●	●	○	○	○	$n \geq 3,\ f \geq 2$, weak	*Impossible*	Lemma 5.5
●		●		●					$f \geq 2$, weak	Deterministic	Lemma 5.6
●		●		○	○	○	○	●	$f \geq 2$, weak	Probabilistic	Lemma 5.7

'●" means explicit; "○" means implicit; negative results are in italic

Table 3. Summary of the main results in Byzantine-prone systems

ATOM	CORDA	mult.	no mult.	centralized	regular	k-bounded	arbitrary	unfair	Conditions	Solution	Source
●	○		●				●	○	$n = 3,\ f = 1$	*No deterministic*	Agmon–Peleg [8]
●	○		●	●	●	○	○	○	$n = 3,\ f = 1$	*No deterministic*	Lemma 5.8
●		●		●	●				$n = 3,\ f = 1$	Deterministic	Note 5.1
●	○	●	○	●		●	○	○	$n = 3,\ f = 1,\ k \geq 2$	*No deterministic*	Lemma 5.9
●		●		●					n odd, $n > 4,\ f = 1$	Deterministic	Note 5.2
●	○	●	○	●		●	○	○	n even $n \geq 2,\ f = 1,\ k \geq n-1$	*No deterministic*	Lemma 5.10
●	○	●	○	●		●	○	○	$f \geq 2,\ k \geq \begin{cases} \frac{n-f}{f} & \text{if } n \text{ even} \\ \frac{n-f}{f-1} & \text{if } n \text{ odd} \end{cases}$	*No deterministic*	Lemma 5.11
●		●			○	●			$n \geq 3$	Probabilistic	Lemma 5.12

"●" means explicit; "○" means implicit; negative results are in italic

possibly change some of the lower bounds presented here. We conjecture that the bounds will hold even if randomization is used differently.

References

1. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal of Computing **28**(4) (1999) 1347–1363
2. Prencipe, G.: Corda: Distributed coordination of a set of autonomous mobile robots. In: Proc. 4th European Research Seminar on Advances in Distributed Systems (ERSADS'01), Bertinoro, Italy (2001) 185–190

3. Prencipe, G.: On the feasibility of gathering by autonomous mobile robots. In Pelc, A., Raynal, M., eds.: Proc. Structural Information and Communication Complexity, 12th Intl Coll., SIROCCO 2005. Volume 3499 of LNCS., Mont Saint-Michel, France, Springer (2005) 246–261
4. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous mobile robots with limited visibility. Theoretical Computer Science **337** (2005) 147–168
5. Souissi, S., Défago, X., Yamashita, M.: Eventually consistent compasses for robust gathering of asynchronous mobile robots with limited visibility. Research Report IS-RR-2005-010, JAIST, Ishikawa, Japan (2005)
6. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Trans. on Robotics and Automation **15**(5) (1999) 818–828
7. Cohen, R., Peleg, D.: Convergence of autonomous mobile robots with inaccurate sensors and movements. In Durand, B., Thomas, W., eds.: 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS'06). Volume 3884 of LNCS., Marseille, France, Springer (2006) 549–560
8. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. In: Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004), New Orleans, LA, USA (2004) 1070–1078
9. Fribourg, L., Messika, S., Picaronny, C.: Coupling and self-stabilization. Distributed Computing **18**(3) (2006) 221–232
10. Défago, X., Gradinariu, M., Messika, S., Raipin-Parvédy, P.: Fault-tolerant and self-stabilizing mobile robots gathering: Feasibility study. Tech. Rep. PI-1802, IRISA, Rennes, France (2006)
11. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CA, USA (1996)
12. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata. Information and Computation **185**(1) (2003) 105–157
13. Dolev, S.: Self-Stabilization. MIT Press (2000)
14. Prencipe, G.: The effect of synchronicity on the behavior of autonomous mobile robots. Theory of Computing Systems **38**(5) (2005) 539–558

Fast Computation by Population Protocols with a Leader

Dana Angluin[1], James Aspnes[1,⋆], and David Eisenstat[2]

[1] Yale University, Department of Computer Science
[2] Princeton University, Department of Computer Science

Abstract. Fast algorithms are presented for performing computations in a probabilistic population model. This is a variant of the standard population protocol model—in which finite-state agents interact in pairs under the control of an adversary scheduler—where all pairs are equally likely to be chosen for each interaction. It is shown that when a unique leader agent is provided in the initial population, the population can simulate a virtual register machine in which standard arithmetic operations like comparison, addition, subtraction, and multiplication and division by constants can be simulated in $O(n \log^4 n)$ interactions with high probability. Applications include a reduction of the cost of computing a semilinear predicate to $O(n \log^4 n)$ interactions from the previously best-known bound of $O(n^2 \log n)$ interactions and simulation of a LOGSPACE Turing machine using the same $O(n \log^4 n)$ interactions per step. These bounds on interactions translate into $O(\log^4 n)$ time per step in a natural parallel model in which each agent participates in an expected $\Theta(1)$ interactions per time unit. The central method is the extensive use of epidemics to propagate information from and to the leader, combined with an epidemic-based phase clock used to detect when these epidemics are likely to be complete.

1 Introduction

The **population protocol** model of Angluin *et al.* [3] consists of a population of finite-state agents that interact in pairs, where each interaction updates the state of both participants according to a transition function based on the participants' previous states and the goal is to have all agents eventually converge to a common output value that represents the result of the computation, typically a predicate on the initial state of the population. A population protocol that always converges to the correct output is said to perform **stable computation** and a predicate that can be so computed is called **stably computable**.

In the simplest version of the model, any pair of agents may interact, but which interaction occurs at each step is under the control of an adversary, subject to a fairness condition that essentially says that any continuously reachable global configuration is eventually reached. The class of stably computable predicates in this model is now very well understood: it consists precisely of the **semilinear predicates** (those predicates on counts of input agents definable in first-order **Presburger arithmetic** [23]), where

[⋆] Supported in part by NSF grants CNS-0305258 and CNS-0435201.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 61–75, 2006.

semilinearity was shown to be sufficient in [3] and necessary in [5]. However, the fact that a protocol will eventually converge to the correct value of a semilinear predicate says little about how long such convergence will take.

Our fundamental measure of convergence is the total number of pairwise interactions until all agents have the correct output value, considered as a function of n, the number of agents in the population. We may also consider models in which reactions occur in parallel according to a Poisson process (as assumed in e.g. [18, 17]); this gives an equivalent distribution over sequences of reactions but suggests a measure of **time** based on assuming each each agent participates in an expected $\Theta(1)$ interactions per time unit. It is not hard to see that this time measure is asymptotically equal to the number of interactions divided by n.

To bound these measures, it is necessary to place further restrictions on the adversary: a merely fair adversary may wait an arbitrary number of interactions before it allows a particular important interaction to occur. In the present work, we consider the natural probabilistic model, proposed in [3], in which each interaction occurs between a pair of agents chosen uniformly at random. In this model, it was shown in [3] that any semilinear predicate can be computed in $\Theta(n^2 \log n)$ expected interactions using a protocol based on leader election in which the leader communicates the outcome by interacting with every other agent. Protocols were also given to simulate randomized LOGSPACE computations with polynomial slowdown, allowing an inverse polynomial probability of failure.

We give a new method for the design of probabilistic population protocols, based on controlled use of self-timed epidemics to disseminate control information rapidly through the population. This method organizes a population as an array of registers that can hold values linear in the size of the population. The simulated registers support the usual arithmetic operations, including addition, subtraction, multiplication and division by constants, and comparison, with implementations that complete with high probability in $O(n \log^4 n)$ interactions and polylogarithmic time per operation. As a consequence, any semilinear predicate can be computed without error by a probabilistic population protocol that converges in $O(n \log^4 n)$ interactions with high probability, and randomized LOGSPACE computation can be simulated with inverse polynomial error with only polylogarithmic slowdown. These bounds are optimal up to polylogarithmic factors, because $\Omega(n \log n)$ interactions are necessary to ensure that every agent has participated in at least one interaction with high probability.

However, in order to achieve these low running times, it is necessary to assume a leader in the form of some unique input agent. This is a reasonable assumption in sensor network models as a typical sensor network will have some small number of sensors that perform the specialized task of communicating with the user and we can appoint one of these as leader. Assuming the existence of a leader does not trivialize the problem; for example, any protocol that requires that the leader personally visit every agent in the population runs in expected number of interactions at least $\Omega(n^2 \log n)$.

If a leader is not provided, it is in principle possible to elect one; however, the best known expected bounds for leader election in a population protocol is still the $\Theta(n^2)$ interactions or $\Theta(n)$ time of a naive protocol in which candidate leaders drop out only on encountering other leaders. It is an open problem whether a leader can be elected significantly faster. There must also be a way to reinitialize the simulation protocol once all but one of the candidates drops out. We discuss these issues further in Section 7.

In building a register machine from agents in a population protocol, we must solve many of the same problems as hardware designers building register machines from electrons. Thus the structure of the paper roughly follows the design of increasing layers of abstraction in a CPU. We present the underlying physics of the world—the population protocol model—in Section 2. Section 3 gives concentration bounds on the number of interactions to propagate the epidemics that take the place of electrical signals and describes the phase clock used to coordinate the virtual machine's instruction cycle. Section 4 describes the microcode level of our machine, showing how to implement operations that are convenient to implement but hard to program with. More traditional register machine operations are then built on top of these microcode operations in Section 5, culminating in a summary of our main construction in Theorem 2. Applications to simulating LOGSPACE Turing machines and computing semilinear predicates are described in Section 6. Some directions for future work are described in Section 7. Due to space limitations, most proofs are omitted from this extended abstract.

Many of our results are probabilistic, and our algorithms include tuning parameters that can be used to adjust the probability of success. For example, the algorithm that implements a given register machine program is designed to run for n^k instructions for some k, and the probability of failure for each instruction must be bounded by a suitable inverse polynomial in n. We say that a statement holds **with high probability** if for any constant c there is a setting of the tuning parameters that cause the statement to hold with probability at least $1 - n^{-c}$. The cost of achieving a larger value of c is a constant factor slowdown in the number of interactions (or time) used by the algorithms.

1.1 Related Work

The population protocol model has been the subject of several recent papers. Diamadi and Fischer introduced a version of the probabilistic model to study the propagation of trust in a social network [15], and a related model of urn automata was explored in [2]. One motivation for the basic model studied in [3] was to understand the computational capabilities of populations of passively mobile sensors with very limited computational power. In the simplest form of the model, any agent may interact with any other, but variations of the model include limits on which pairs of agents may interact [3, 1, 4], various forms of one-way and delayed communication [6], and failures of agents [14]. The properties computed by population protocols have also been extended from predicates on the initial population to predicates on the underlying interaction graph [1], self-stabilizing behaviors [7], and stabilizing consensus [8].

Similar systems of pairwise interaction have previously been used to model the interaction of small molecules in solution [18,19] and the propagation in a human population of rumors [12] or epidemics of infectious disease [10]. Epidemic algorithms have also been used previously to perform multicast operations, e.g. by Birman *et al.* [11].

The notion of a "phase clock" as used in our protocol is common in the self-stabilizing literature, e.g. [20]. There is a substantial stream of research on building self-stabilizing synchronized clocks dating back to to the work of Arora *et al.* [9]. Recent work such as [16] shows that it is possible to perform self-stabilizing clock synchronization in traditional distributed systems even with a constant fraction of Byzantine faults; however, the resulting algorithms require more network structure and computational capacity at each agent that is available in a population protocol. An intriguing protocol of Daliot *et al.* [13] constructs a protocol for the closely-related problem of pulse synchronization inspired directly by biological models. Though this protocol also exceeds the finite-state limits of population protocols, it may be possible to construct a useful phase clock for our model by adapting similar techniques.

2 Model

In this paper we consider only the complete all-pairs interaction graph, so we can simplify the general definition of a probabilistic population protocol as follows. A **population protocol** consists of a finite set Q of states, of which a nonempty subset X are the initial states (thought of as inputs), a deterministic transition function $(a, b) \mapsto (a', b')$ that maps ordered pairs of states to ordered pairs of states, and an output function that maps states to an output alphabet Y. The **population** consists of agents numbered 1 through n; agent identities are not visible to the agents themselves, but facilitate the description of the model. A **configuration** C is a map from the population to states, giving the current state of every agent. An **input configuration** is a map from the population to X, representing an input consisting of a multiset of elements of X. C can reach C' in one interaction, denoted $C \rightarrow C'$, if there exist distinct agents i and j such that $C(i) = a$, $C(j) = b$, the transition function specifies $(a, b) \mapsto (a', b')$ and $C'(i) = a'$, $C'(j) = b'$ and $C'(k) = C(k)$ for all k other than i and j. In this interaction, i is the **initiator** and j is the **responder** – this asymmetry of roles is an assumption of the model [4].

An **execution** is a sequence $C_1, C_2, \ldots$ of configurations such that for each i, $C_i \rightarrow C_{i+1}$. An execution **converges** to an output $y \in Y$, if there exists an i such that for every $j \geq i$, the output function applied to every state occurring in C_j is y. In general, individual agents may not know when convergence to a common output has been reached, and protocols are generally designed not to halt. An execution is **fair** if for any C_i and C_j such that $C_i \rightarrow C_j$ and C_i occurs infinitely often in the execution, C_j also occurs infinitely often in the execution. A protocol **stably computes** a predicate P on multisets of elements of X if for any input configuration C, every fair execution of the protocol starting with C converges to 1 if P is true on the multiset of inputs represented by C, and converges to 0 otherwise. Note that a fixed protocol must be able to handle populations of arbitrary finite size – there is no dependence of the number of states on n, the population size.

For a **probabilistic population protocol**, we stipulate a particular probability distribution over executions from a given configuration C_1 as follows. We generate C_{k+1} from C_k by drawing an ordered pair (i, j) of agents independently and uniformly, applying the transition function to $(C_k(i), C_k(j))$, and updating the states of i and j accordingly to obtain C_{k+1}. (Note that an execution generated this way will be fair with probability 1.) In the probabilistic model we consider both the random variable of the number of interactions until convergence and the probabilities of various error conditions in our algorithms.

3 Tools

Here we give the basic tools used to construct our virtual machine. These consist of concentration bounds on the number of interactions needed to spread epidemics through the population (Section 3.1), which are then used to construct a phase clock that controls the machine's instruction cycle (Section 3.2). Basic protocols for duplication (Section 3.3), cancellation (Section 3.4), and probing (Section 3.5) are then defined and analyzed.

3.1 Epidemics

By a **one-way epidemic** we denote the population protocol with state space $\{0, 1\}$ and transition rule $(x, y) \mapsto (x, \max(x, y))$. Interpreting 0 as "susceptible" and 1 as "infected," this protocol corresponds to a simple epidemic in which transmission of the infection occurs if and only if the initiator is infected and the responder is susceptible. In the full paper, we show, using a reduction to coupon collector and sharp concentration results of [21], that the number of interactions for the epidemic to finish (that is, infect every agent) is $\Theta(n \log n)$ with high probability.

It will be useful to have a slightly more general lemma that bounds the time to infect the first k susceptible agents. Because of the high variance associated with filling the last few bins in the coupon collection problem, we consider only $k \geq n^{\epsilon}$ for $\epsilon > 0$.

Lemma 1. *Let $T(k)$ be number of interactions before a one-way epidemic starting with a single infected agent infects k agents. For any fixed $\epsilon > 0$ and $c > 0$, there exist positive constants c_1 and c_2 such that for sufficiently large n and any $k > n^{\epsilon}$, $c_1 n \ln k \leq T(k) \leq c_2 n \ln k$ with probability at least $1 - n^{-c}$.*

3.2 The Phase Clock

The core of our construction is a **phase clock** that allows a leader to determine when an epidemic or sequence of triggered epidemics is likely to have finished. In essence, the phase clock allows a finite-state leader to count off $\Theta(n \log n)$ total interactions with high probability; by adjusting the constants in the clock, the resulting count is enough to outlast the $c_2 n \ln n$ interactions needed to complete an epidemic by Lemma 1. Like physical clocks, the phase clock is based on a readily-available natural phenomenon with the right duration constant. A good choice for this natural phenomenon, in a probabilistic population protocol, turns out to be itself the spread of an epidemic. Like the one-way epidemic of Section 3.1, the phase clock requires only one-way communication.

Here is the protocol: each agent has a state in the range $0 \ldots m-1$ for some constant m that indicates which phase of the clock it is infected with. (The value of m will be chosen independent of n, but depending on c, where $1 - n^{-c}$ is the desired success probability.) Up to a point, later phases overwrite earlier phases: a responder in phase i will adopt the phase of any initiator in phases $i+1 \bmod m$ through $i+m/2 \bmod m$, but will ignore initiators in other phases. This behavior completely describes the transition function for non-leader responders.

New phases are triggered by a unique leader agent. When the leader encounters an initiator with its own phase, it spontaneously moves to the next phase. The leader ignores interactions with initiators in other phases. The initial configuration of the phase clock has the leader in phase 0 and all other agents in phase $m-1$. A **round** consists of m phases. A new round starts when the leader enters phase 0.

The normal operation of the phase clock has all the agents in a very few adjacent states, with the leader in the foremost one. When that state becomes populated enough for the leader to encounter another agent in that state, the leader moves on to the next state (modulo m) and the followers are pulled along. Successive rounds should be $\Theta(n \log n)$ interactions apart with high probability; the lower bound allows messages sent epidemically to reach the whole population, and the upper bound is essential for the overall efficiency of our algorithms.

Analysis. We wish to show that for appropriate constants c and m, any epidemic (running in parallel with the phase clock) that starts in phase i completes by the next occurrence of phase $(i+c) \mod m$ with high probability. To simplify the argument, we first consider an infinite-state version of the phase clock with state space $\mathbb{Z} \times \{\text{leader}, \text{follower}\}$ and transition rules

$$\begin{aligned}
(x,b),(y,\text{follower}) &\mapsto (x,b),(\max(x,y),\text{follower})\\
(x,b),(x,\text{leader}) &\mapsto (x,b),(x+1,\text{leader})\\
(x,b),(y,\text{leader}) &\mapsto (x,b),(y,\text{leader}) \quad [y \neq x]
\end{aligned}$$

We assume the initial configuration (at interaction 0) has the leader in state 0 and each follower in state -1. This infinite-state protocol has the useful invariant that every agent has a phase less than or equal to that of the leader. We define phase i as starting when the leader agent first adopts phase i. This result bounds the probability that a phase “ends too early” by $n^{-1/2}$.

Lemma 2. *Let phase i start at interaction t. Then there is a constant a such that for sufficiently large n, phase $i+1$ starts before interaction $t + an\ln n$ with probability at most $n^{-1/2}$.*

Observing that several phases must “end too early” in order for a round to “end too early” allows us to go from a failure probability of $n^{-1/2}$ for a phase to n^{-c} for a round.

Corollary 1. *Let phase i start at interaction t. Then for any $c>0$ and $d>0$, there is a constant k such that for sufficiently large n, phase $i+k$ starts before $t + dn\ln n$ interactions with probability at most n^{-c}.*

The following theorem gives probabilistic guarantees for a polynomial number of rounds of the phase clock. In the proof the probability of failure due to a “straggler” (agent so far behind that it appears to be ahead modulo m) must be also be appropriately bounded, to ensure that m may be a constant independent of n.

Theorem 1. *For any fixed $c, d > 0$, there exists a constant m such that, for all sufficiently large n, the finite-state phase clock with parameter m, starting from an initial* *state consisting of one leader in phase* 0 *and* $n-1$ *followers in phase* $m-1$*, completes* n^c *rounds of* m *phases each, where the minimum number of interactions in any of the* n^c *rounds is at least* $dn \ln n$ *with probability at least* $1-n^{-c}$.

Proof. The essential idea is to apply Corollary 1 twice: once to show that with high probability the number of interactions between phase $i+1$ and phase $i+m/2$ is long enough for any old phase-i agents to be eaten up (thus avoiding any problems with wrap-around), and once to show the lower bound on the length of a round.

To show that no agent is left behind, consider, in the infinite-state protocol, the fate of agents in phase i or lower once at least one agent in phase $i+1$ or higher exists. If we map all phases i or lower to 0 and all phases $i+1$ or higher to 1, then encounters between agents have the same effect after the mapping as in a one-way epidemic. By Lemma 1, there is a constant c_2 such that all n agents are infected by interaction $c_2 n \ln n$ with probability at least $1-n^{-3c}$. By Corollary 1, there is a constant k_1 such that phase $i+k_1+1$ starts at least $c_2 n \ln n$ interactions after phase $i+1$ with probability at least $1-n^{-3c}$. Setting $m > 2(k_1+1)$ then ensures that all phase i (or lower) agents have updated their phase before phase $i+m/2$ with probability at least $1-2n^{-3c}$. If we sum the probability of failure over all mn^c phases in the first n^c rounds, we get a probability of at most $2mn^{-2c}$ that some phase i agent survives long enough to cause trouble.

Assuming that no such trouble occurs, we can simulate the finite-state phase clock by mapping the phases of the infinite-state phase clock mod m. Now by Corollary 1 there is a constant k_2 such that the number of interactions to complete k_2 consecutive phases is at least $dn \ln n$ with probability at least $1-n^{-3c}$. Setting $m \geq k_2$ thus gives that all n^c rounds take at least $dn \ln n$ interactions with probability at least $1-n^c n^{-3c} = 1-n^{-2c}$. Thus the total probability of failure is bounded by $2mn^{-2c}+n^{-2c} < n^{-c}$ for sufficiently large n as claimed.

3.3 Duplication

A **duplication** protocol has state space $\{(1,1),(0,1),(0,0)\}$ and transition rules:

$$\begin{aligned}(1,1),(0,0) &\mapsto (0,1),(0,1)\\ (0,0),(1,1) &\mapsto (0,1),(0,1)\end{aligned}$$

with all other encounters having no effect.

When run to convergence, a duplication protocol starting with a "active" agents in state $(1,1)$ and the rest in the null state $(0,0)$ converges to $2a$ "inactive" agents in state $(0,1)$, provided $2a$ is less than n; otherwise it converges to a population of mixed active and inactive agents with no unrecruited agents left in the null state. The invariant is that the total number of 1 tokens is preserved while eliminating as many double-token agents as possible. We do not consider agents in a $(1,0)$ state as they can be converted to $(0,1)$ immediately at the start of the protocol.

When the initial number of active agents a is close to $n/2$, duplication may take as much as $\Theta(n^2)$ interactions to converge, as the last few active agents wait to encounter the last few null agents. But for smaller values of a the protocol converges more quickly.

Lemma 3. *Let $2a + b \leq n/2$. The probability that a duplication protocol starting with a active agents and b inactive agents, has not converged after $(2c+1)n \ln n$ interactions is at most n^{-c}.*

3.4 Cancellation

A **cancellation** protocol has states $\{(0,0), (1,0), (0,1)\}$ and transition rules:

$$(1,0), (0,1) \mapsto (0,0), (0,0)$$
$$(0,1), (1,0) \mapsto (0,0), (0,0)$$

It maintains the invariant that the number of 1 tokens in the left-hand position minus the number of 1 tokens in the right-hand position is fixed. It converges when only $(1,0)$ and $(0,0)$ or only $(0,1)$ and $(0,0)$ agents remain. We assume that there are no $(1,1)$ agents as these can be converted to $(0,0)$ agents at the start of the protocol. We refer to agents in state $(1,0)$ or $(0,1)$ as nonzero agents.

As with duplication, the number of interactions to converge when $(1,0)$ and $(0,1)$ are nearly equally balanced can be as many as $\Theta(n^2)$, since we must wait in the end for the last few survivors to find each other. This is too slow to use cancellation to implement subtraction directly. Instead, we will use cancellation for inequality testing, using duplication to ensure that there is a large enough majority of one value or the other to ensure fast convergence. We will use the following fact.

Lemma 4. *Starting from any initial configuration, with probability at least $1 - n^{-c}$, after $4(c+1)n \ln n$ interactions a cancellation protocol has either converged or has at most $n/8$ of each type of nonzero agent.*

3.5 Probing

A **probing** protocol is used to detect if any agents satisfying a given predicate exist. It uses three states (in addition to any state tested by the predicate) and has transition rules

$$(x, y) \mapsto (x, \max(x, y))$$

when the responder does not satisfy the predicate and

$$(0, y) \mapsto (0, y)$$
$$(x, y) \mapsto (x, 2) \quad [x > 0]$$

when the responder does. Note that this is a one-way protocol.

To initiate a probe, a leader starts in state 1; this state spreads through an initial population of state 0 agents as in a one-way epidemic and triggers the epidemic spread of state 2 if it reaches an agent that satisfies the predicate.

Lemma 5. *For any $c > 0$, there is a constant d such that for sufficiently large n, with probability at least $1 - n^{-c}$ it is the case that after $dn \ln n$ interactions in the probing protocol either (a) no agent satisfies the predicate and every agent is in state 1, or (b) some agent satisfies the predicate and every agent is in state 2.*

4 Computation by Epidemic: The Microcode Level

In this section, we describe how to construct an abstract register machine on top of a population protocol. This machine has a constant number of registers each capable of holding integer values in the range 0 to n, and supports the usual arithmetic operations on these registers, including addition, subtraction, multiplication and division by constants, inequality tests, and so forth. Each of these operations takes at most a polylogarithmic number of basic instruction cycles, where an instruction cycle takes $\Theta(n \log n)$ interactions or $\Theta(\log n)$ time.

The simulation is probabilistic; there is an inverse polynomial probability of error for each operation, on which the exponent can be made arbitrarily large at the cost of increasing the constant factor in the running time.

The value of each register is distributed across the population in unary. For each register A, every member i of the population maintains one bit A_i and the current value of A is simply $\sum_i A_i$. Thus the finite state of each agent can be thought of as a finite set of finite-valued control variables, and one boolean variable for each of a finite set of registers. Recall that the identities of agents are invisible to the agents themselves, and are used to facilitate description of the model.

We assume there is a leader agent that organizes the computation; part of the leader's state stores the finite-state control for the register machine. We make a distinction between the "microcode layer" of the machine, which uses the basic mechanisms of Section 3, and the "machine code" layer, which provides familiar arithmetic operations.

At the microcode layer, we implement a basic instruction cycle in which the leader broadcasts an instruction to all agents using an epidemic. The agents then carry out this instruction until stopped by a second broadcast from the leader. This process repeats until the computation terminates.

To track the current instruction, each agent (including the leader) has a **current instruction register** in addition to its other state. These instructions are tagged with a **round number** in the range $0, 1, 2$, where round i instructions are overwritten by round $i + 1 \pmod 3$ instructions.

The instructions and their effects are given in Table 1. Most take registers as arguments. We also allow any occurrence of a register to be replaced by its negation, in which case the operation applies to those agents in which the appropriate bit is not set. For example, SET$(\neg A)$ resets A_i, PROBE$(\neg A)$ tests for agents in which A_i is not set, COPY$(\neg A, B)$ sets B_i to the negation of A_i, and so forth.

To interpret the table entries: when an agent changes its current instruction register to SET(A), it sets its boolean variable for register A to 1 and waits for the next instruction. Similarly, when it changes its current instruction register to COPY(A, B), then the agent sets its boolean variable for register B to the value of its boolean variable for register A. When its current instruction becomes DUP(A, B), then the agent begins running the duplication protocol (Section 3.3) on the ordered pair of its boolean variables for registers A and B. (In the case of $(1, 0)$, it immediately exchanges them to $(0, 1)$, and in the cases of $(1, 1)$ and $(0, 0)$, it participates in the duplication protocol when it interacts with other agents with current instruction DUP(A, B), until either its pair becomes inactive or a new instruction supersedes the current one.) CANCEL(A, B) and PROBE(A) are handled analogously, where the predicate probed is whether the agent's

Table 1. Instructions at the microcode level

Instruction	Effect on state of agent i
NOOP	No effect.
SET(A)	Set $A_i = 1$.
COPY(A, B)	Copy A_i to B_i
DUP(A, B)	Run duplication protocol on state (A_i, B_i).
CANCEL(A, B)	Run cancellation protocol on state (A_i, B_i).
PROBE(A)	Run probe protocol with predicate $A_i = 1$.

boolean variable for register A is 1. We omit describing the underlying transitions as the details are tedious.

When the leader updates its own current instruction register, the new value spreads to all other agents in $\Theta(n \log n)$ interactions with high probability (Lemma 1). The NOOP, SET, and COPY operations take effect immediately, so no additional interactions are required. The PROBE operation may require waiting for a second triggered epidemic, but the total interactions are still bounded by $O(n \log n)$ with high probability (by Lemma 5). Only the DUP and CANCEL operations may take longer to converge. Because subsequent operations overwrite each agent's current instruction register, issuing a new operation has the effect of cutting these operations off early. But if this new operation is issued $\Omega(n \log n)$ interactions later, the DUP operation converges with high probability unless it must recruit more than half the agents (Lemma 3), and the CANCEL operation either converges or leaves at most $n/4$ uncanceled values (Lemma 4). Note that for either operation, which outcome occurred can be detected with COPY and PROBE operations.

Thus, the leader waits for $\Omega(n \log n)$ interactions between issuing successive instructions, where the constant is chosen based on the desired error bound. But this can be done using a phase clock with appropriate parameter (Theorem 1): if it is large enough that both the probability that an operation completes too late and the probability that some phase clock triggers to early is $o(n^{-2c})$ per operation, then the total probability that any of n^c operations fails is $o(n^{-c})$.

5 Computation by Epidemic: Higher-Level Operations

The operations of the previous section are not very convenient for programming. In this section, we describe how to implement more traditional register operations.

These can be divided into two groups: those that require a constant number of microcode instructions, and those that are implemented using loops. The first group, shown in Table 2, includes assignment, addition, multiplication by a constant, and zero tests. The second group includes comparison (testing for $A < B$, $A = B$, or $A > B$), subtraction, and division by a constant (including obtaining the remainder). These operations are described in more detail below.

Comparison. For comparison, it is tempting just to apply CANCEL and see what tokens survive. But if the two registers A and B being compared are close in value,

Table 2. Simple high-level operations and their implementations. Register X is an auxiliary register.

Operation	Effect	Implementation	Notes
Constant 0	$A \leftarrow 0$	SET($\neg A$)	
Constant 1	$A \leftarrow 1$	SET($\neg A$) $A_{\text{leader}} \leftarrow 1$	
Assignment	$A \leftarrow B$	COPY(B, A)	

Addition	$A \leftarrow A + B$	COPY(B, X)
		DUP(X, A)
		PROBE(X)
			May fail with $X \neq 0$ if $A + B > n/2$.

Multiplication	$A \leftarrow kB$	Use repeated addition.	$k = O(1)$
Zero test	$A \neq 0$?	PROBE(A)	

```
$A' \leftarrow A$.
$B' \leftarrow B$.
$C \leftarrow 1$.
$r \leftarrow 0$.
**while** true **do**
    CANCEL($A', B'$).
    **if** $A' = 0$ and $B' = 0$ **then**
        return $A = B$.
    **else if** $A' = 0$ **then**
        return $A < B$.
    **else if** $B' = 0$ **then**
        return $A > B$.
    **end if**
    $r \leftarrow 1 - r$.
    **if** $r = 0$ **then**
        $C \leftarrow C + C$.
        **if** addition failed **then**
            return $A = B$.
        **end if**
    **end if**
    $A' \leftarrow A' + A'$.
    $B' \leftarrow B' + B'$.
**end while**
```

Fig. 1. Comparison algorithm

```
$A' \leftarrow A$.
$B' \leftarrow B$.
CANCEL($A', B'$).
**if** $B' = 0$ **then**
    $C \leftarrow A$.
    return.
**end if**
$C \leftarrow 0$.
**while** $A' \neq B' + C$ **do**
    $D \leftarrow 1$.
    **while** $A' \geq B' + C + D + D$ **do**
        $D \leftarrow D + D$.
    **end while**
    $C \leftarrow C + D$.
**end while**
```

Fig. 2. Subtraction algorithm

then CANCEL may take $\Theta(n^2)$ interactions to converge. Instead, we apply up to $2 \lg n$ rounds of cancellation, alternating with duplication steps that double the discrepancy between A and B. If $A > B$ or $B > A$, the difference soon becomes large enough that all of the minority tokens are eliminated. The case where $A = B$ is detected by failure to converge, using a counter variable C that doubles every other round.

The algorithm is given in Figure 1. It uses registers A', B', and C plus a bit r to detect even-numbered rounds.

Lemma 6. *Algorithm 1 returns the correct answer with high probability after executing at most $O(\log n)$ microcode operations.*

Subtraction. Subtraction is the inverse of addition, and addition is a monotone operation. It follows that we can implement subtraction using binary search. Our rather rococo algorithm for computing $C \leftarrow A - B$, given in Figure 2 repeatedly looks for the largest power of two that can be added to the candidate difference C without making the sum of the difference C and the subtrahend B greater than the minuend A. It obtains one more 1 bit of the difference for each iteration.

The algorithm assumes $A \geq B$. An initial cancellation step is used to handle particularly large inputs. This allows the algorithm to work even when A lies outside the safe range of the addition operation.

The algorithm uses several auxiliary registers to keep track of the power of two to add to C (this is the D register) and to perform various implicit sums and tests (as in computing $B' + C + D + D$).

Lemma 7. *When $A \geq B$, Algorithm 2 computes $C \leftarrow A - B$ with high probability in $O(\log^3 n)$ microcode operations.*

Division. Division of A by a constant k is analogous to subtraction; we set $A' \leftarrow A$ and $B \leftarrow 0$ and repeatedly seek the largest power of two D such that kD can be successfully computed (i.e., does not cause addition to overflow) and $kD \leq A'$. We then subtract kD from A' and add D to B.

The protocol terminates when $A' < k$, i.e. when no value of D works. At this point B holds the quotient $\lfloor A/k \rfloor$ and A' the remainder $A \bmod k$. Since each iteration adds one bit to the quotient, there are at most $O(\lg n)$ iterations of the outer loop, for a total cost of $O(\lg^4 n)$ microcode operations (since each outer loop iteration requires one subtraction operation).

One curious property of this protocol is that the leader does not learn the value of the remainder, even though it is small enough to fit in its limited memory. If it is important for the leader to learn the remainder, it can do so using k addition and comparison operations, by successively testing the remainder A' for equality with the values $0, 1, 1+1, 1+1+1, \ldots, k$. The cost of this test is dominated by the cost of the division algorithm.

Other operations. Multiplication and division by constants give us the ability to extract individual bits of a register value A. This is sufficient to implement basic operations like $A \leftarrow B \cdot C$, $A \leftarrow \lfloor B/C \rfloor$ in polylogarithmic time using standard bitwise algorithms.

Summary. Combining preceding results gives:

Theorem 2. *A probabilistic population can simulate steps of a virtual machine with a constant number of registers holding integer values in the range 0 to n, where each step consists of (a) assigning a constant 0 or 1 value to a register; (b) assigning the value of one register to another; (c) adding the value of one register to another, provided the total does not exceed $n/2$; (d) multiplying a register by a constant, provided the result* *does not exceed $n/2$; (e) testing if a register is equal to zero; (f) comparing the values of two registers; (g) subtracting the values of two registers; or (h) dividing the value of a register by a constant and computing the remainder. The probability that for any single operation the simulation fails or takes more than $O(n \log^4 n)$ interactions can be made $O(n^{-c})$ for any fixed c.*

6 Applications

Simulating RL. In [3], it was shown that a probabilistic population protocol with a leader could simulate a randomized LOGSPACE Turing machine with a constant number of read-only unary input tapes with polynomial slowdown. The basic technique was to use the standard reduction of Minsky [22] of a Turing machine to a counter machine, in which a Turing machine tape is first split into two stacks and then each stack is represented as a base-b number stored in unary. Because the construction in [3] could only increment or decrement counters, each movement of the Turing machine head required decrementing a counter to zero in order to implement division or multiplication. Using Theorem 2, we can perform division and multiplication in $O(n \log^4 n)$ interactions, which thus gives the number of interactions for a single Turing machine step. If we treat this quantity as $O(\log^4 n)$ time, we get a simulation with polylogarithmic slowdown.

Theorem 3. *For any fixed $c > 0$, there is a constant d such that a probabilistic population protocol on a complete graph with a leader that can simulate n^c steps of a randomized LOGSPACE Turing machine with a constant number of read-only unary input tapes using $d \log^4 n$ time per step with a probability of failure bounded by n^{-c}.*

Protocols for semilinear predicates. From [3] we have that it is sufficient to be able to compute congruence modulo k, $+$, and $<$ to compute any semilinear predicate. From Theorem 2 we have that all of these operations can be computed with a leader in $O(n \log^4 n)$ interactions with high probability. The final stage of broadcasting the result to all agents can also be performed in $O(n \log n)$ interactions with high probability using an epidemic.

However, there is some chance of never converging to the correct answer if the protocol fails. To eliminate this possibility, we construct an optimistic hybrid protocol in which the fast but potentially inaccurate $O(n \log^4 n)$-interaction protocol is supplemented by an $O(n^2)$ leaderless protocol, with the leader choosing (in case of disagreement) to switch its output from that of the fast protocol to that of the slow protocol when it is likely the slow protocol has finished. The resulting hybrid protocol converges to the correct answer in all executions while still converging in $O(n \log^4 n)$ interactions in expectation and with high probability.

Theorem 4. *For any semilinear predicate P, and for any $c > 0$, there is a probabilistic population protocol on a complete graph with a leader to compute P without error that converges in $O(n \log^4 n)$ interactions with probability at least $1 - n^{-c}$ and in expectation.*

7 Open Problems

For most of the paper, we have assumed that a unique leader agent is provided in the initial input. The most pressing open problem is whether this assumption can be eliminated without drastically raising the cost of our protocols.

One problem is the question of whether we can efficiently restart the phase clock after completing an initial leader election phase. A proof of possibility can be obtained by observing that the leader can shut off all other agents one at a time in $O(n^2 \log n)$ interactions, and then restart them in the same number of interactions; however, the leader may have to wait an additional large polynomial time to be confident that it has in fact reached all agents. We believe, based on preliminary simulation results, that a modified version of our phase clock can be restarted much more efficiently by a newly-elected leader. This would allow us to use our LOGSPACE simulator after an initial $O(n^2)$-interaction leader election stage. But more work is still needed.

Even better would be a phase clock that required no leader at all. This would allow every agent to independently simulate the single leader, eliminating both any initial leader election stage and the need to disseminate instructions. Whether such a leaderless phase clock is possible is not clear.

It would be interesting to explore refinements of the underlying assumption that pairs are drawn uniformly at random to interact, for example, to reflect the physical effects of spatial dispersion of the agents.

References

1. Dana Angluin, James Aspnes, Melody Chan, Michael J. Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In Viktor K. Prasanna, Sitharama Iyengar, Paul Spirakis, and Matt Welsh, editors, *Distributed Computing in Sensor Systems: First IEEE International Conference, DCOSS 2005, Marina del Rey, CA, USA, June/July, 2005, Proceedings*, volume 3560 of *Lecture Notes in Computer Science*, pages 63–74. Springer-Verlag, June 2005.
2. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Urn automata. Technical Report YALEU/DCS/TR-1280, Yale University Department of Computer Science, November 2003.
3. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC '04: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 290–299. ACM Press, 2004.
4. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, March 2006.
5. Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. To appear, PODC 2006, July 2006.
6. Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. On the power of anonymous one-way communication. In *Ninth International Conference on Principles of Distributed Systems*, pages 307–318, December 2005.
7. Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. In *Ninth International Conference on Principles of Distributed Systems*, pages 79–90, December 2005.

8. Dana Angluin, Michael J. Fischer, and Hong Jiang. Stabilizing consensus in mobile networks. To appear in Proc. International Conference on Distributed Computing in Sensor Systems (DCOSS06), June 2006.
9. Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 71–79, Delphi, Greece, 1991. Springer-Verlag.
10. Norman T. J. Bailey. *The Mathematical Theory of Infectious Diseases, Second Edition*. Charles Griffin & Co., London and High Wycombe, 1975.
11. Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.
12. D. J. Daley and D. G. Kendall. Stochastic rumours. *Journal of the Institute of Mathematics and its Applications*, 1:42–55, 1965.
13. Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2003.
14. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. To appear in Proc. International Conference on Distributed Computing in Sensor Systems (DCOSS06), June 2006.
15. Zoë Diamadi and Michael J. Fischer. A simple game for the study of trust in distributed systems. *Wuhan University Journal of Natural Sciences*, 6(1–2):72–82, March 2001. Also appears as Yale Technical Report TR–1207, January 2001.
16. Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.
17. Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A*, 104:1876–1880, 2000.
18. Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
19. Daniel T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A*, 188:404–425, 1992.
20. Ted Herman. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1048–1057, 2000.
21. A. P. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms*, 7:59–80, 1995.
22. M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.
23. Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes-Rendus du I Congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.

On Self-stabilizing Search Trees

Doina Bein[1], Ajoy K. Datta[1], and Lawrence L. Larmore[1]

University of Nevada, Las Vegas, USA
{siona, datta, larmore}@cs.unlv.edu

Abstract. We introduce a self-stabilizing data structure, which we call either a *min-max search tree* or a *max-min search tree* (both abbreviated M^{2}ST), depending on whether the root has the minimum or the maximum value in the tree. Our structure is a refinement of the standard min-max heap (or max-min heap), with additional property that every value in the left subtree of a node is less than or equal to every value in the right subtree of that node. The M^2ST has all the power of a binary search tree and all the power of a min-max heap, combined; with the additional feature that maintaining balance is easy. We give a self-stabilizing algorithm for reorganizing the values of an asynchronous network with a binary tree topology into an M^{2}ST in $O(n)$ rounds. We then give an algorithm for reorganizing an asynchronous network with a binary tree topology, which is already in M^2ST order, into binary search tree order in $O(h)$ rounds. This result answers an open problem posed in [3].

Keywords: Distributed algorithm, min-max heap, search tree, self-stabilization.

1 Introduction

When transient faults or arbitrary initialization cause a data structure to lose a desired property, a *self-stabilizing* [5,7] data structure is able to correct itself, so that the property is restored in finite time. We present a self-stabilizing search structure, which we call a *min-max search tree* on a network with a binary tree topology.

Related Work. Abstractly, a min-max heap is defined to be a data structure that allows insertion and deletion of the minimum and the maximum. Min-max heaps have been defined in [2] as double-ended priority queues. Various implementations of min-max heaps are proposed in [1,4,6,12,13], but none of them is distributed or self-stabilizing.

A heap construction that supports insert and delete operations using a variation of a standard binary heap with the capacity of K items, is given in [9]. In [11], Herman *et al.* make the heap ADT (abstract data type) and B-Tree ADT self-stabilizing with respect to their properties. Stabilizing 2-3 trees are investigated in [10]. Bein *et al.* [3] present the first snap-stabilizing distributed *binary search tree* (BST) algorithm. The stabilization time is $O(n)$ rounds, but every

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 76–89, 2006.

process i requires $O(\log s_i)$ bits where s_i is the size of the subtree rooted at i. A lower bound of $\Omega(n)$ on the time complexity for the BST problem is also given in [3]. They also ask, as a open problem, whether there exists a self-stabilizing algorithm to build a BST in $O(n)$ rounds, using $O(1)$ bits per process.

Contributions. We present a new type of search structure, which we call a *min-max search tree*, (abbreviated M^{2}ST), a binary tree with a value at each node, with min-max heap order, *i.e.*, where the value at each node at an even-numbered level (counting the root level to be 0) is the minimum of all values in the subtree rooted at that node, and the value at each node at an odd-numbered level is the maximum of all values in the subtree rooted at that node; but with the additional property that all values in the left subtree of a node are less than or equal to all values in the right subtree of that node. This structure has the features of a binary search tree combined with the features of a min-max heap, with the additional feature that it is easy to keep balanced.

We give a distributed algorithm that sorts a binary tree network into an M^{2}ST in $O(n)$ rounds using $O(1)$ additional bits per process. The algorithm is self-stabilizing. We then give a distributed algorithm which sorts a binary tree network into BST order in $O(n)$ rounds, using $O(1)$ bits per process. The BST algorithm first sorts the network into a M^2ST order, then sorts it into BST order in $O(h)$ additional rounds, where h is the height of the tree, and requires $O(1)$ additional bits per process.

Outline of the paper. In Sections 1 and 2 we introduce the basic concepts needed in the paper. The M^2ST data structure is introduced in Section 3, and we discuss implementation the usual search structure operators for an M^2ST. In Section 4 we give an asynchronous distributed algorithm which sorts a binary tree network into M^2ST order starting from an arbitrary initial configuration. In Section 5. we give an asynchronous distributed algorithm which sorts a binary tree network into BST order, starting from a configuration which is already in M^2ST order. We conclude in Section 6.

2 Preliminaries

Throughout this paper, we will let T be a *binary tree network*, defined to be a network of processes with a binary tree topology, such that each process can only communicate with its immediate neighbors, and each process knows which of its neighbors is its left child, right child, or parent. Thus, for example, the root process knows it is the root, since its parent is nil. We will also assume that each process has one *value*, and that a process can read its neighbors' values. We will say that "T is a heap" if the values of T are in heap order, and that "T is a binary search tree" if the values of T are in inorder, and so forth.

Let T be a binary tree network. We use the following notation, where T is a binary tree network. Let *root*(T) be the root process of T. If x is a process of T, then $V(x)$ is the value at x, T_x is the subtree rooted at x; $p(x)$, $r(x)$, and

$\ell(x)$ are the parent, the right, and left child of x, respectively. $T^R = T_{r(root)}$ and $T^L = T_{\ell(root)}$ are the right and left subtree of the root, respectively.

We assume the local shared memory model of communication: a process can read and write its own memory, but can only read the memory of its neighbors. The program of every process consists of a finite set of guarded actions of the form: $<$ *label* $>$::$<$ *guard* $>\rightarrow<$ *action* $>$ that involve the process' variables and the variables of its neighbors. If an action has its guard, a Boolean expression, evaluated to *true*, then it is called *enabled.* A process with at least one enabled guard is called *enabled.* In a computation step, a *distributed daemon* selects a nonempty subset of enabled processes. Each enabled process executes one of its enabled actions. The guard evaluation and the execution of the corresponding action are considered to be done in one atomic step.

The *state* of a process is defined by the values of its variables. A *system state* (*configuration*) is a choice of a state for each process. If c, c' are configurations, we write $c \mapsto c'$ to mean that c can change to c' in one step.

An *execution* e is an infinite sequence of configurations $e = c_1 \mapsto c_2 \mapsto \ldots$. Given $\mathcal{C}$, the set of all possible states, and a predicate $\mathcal{P}$ over $\mathcal{C}$, the set of all the states that satisfy $\mathcal{P}$ is denoted by $\mathcal{L}_\mathcal{P} \subseteq \mathcal{C}$, and is called the set of all *legitimate states with respect to* $\mathcal{P}$. We say that a system is *self-stabilizing* with respect to a predicate $\mathcal{P}$ is the following two conditions hold:

1. If $c \in \mathcal{L}_\mathcal{P}$ and $c \mapsto c'$, then $c' \in \mathcal{L}_\mathcal{P}$.
2. If $e = c_1 \mapsto c_2 \mapsto \ldots$ is a computation, then there is some integer j such that $c_i \in \mathcal{L}_\mathcal{P}$ for all $i \geq j$.

For an asynchronous system, in order to compute time complexity, we use the concept of a *round* introduced by Dolev *et al.* [8]: A round is a minimal sequence of computation steps such that each process that was enabled in the first configuration of the sequence executes at least once during the sequence. We will use the strongest distributed daemon, the *unfair* daemon. The unfair daemon is not required to ever select a given enabled process, unless it is the only enabled process.

3 Min-Max and Max-Min Search Trees

We first recursively define three classes of orderings on the nodes of a binary tree, T.

Definition 1.

- *An ordering of the nodes of T is* left-to-right *if either T is empty, or all nodes of T^L come before all nodes of T^R and the induced orderings on T^L and T^R are left-to-right.*
- *An ordering of the nodes of T is* min-max *if either T is empty, or the root node is first and the induced orderings on T^L and T^R are max-min. An ordering is* min-max-left-right *if it is both min-max and left-to-right.*

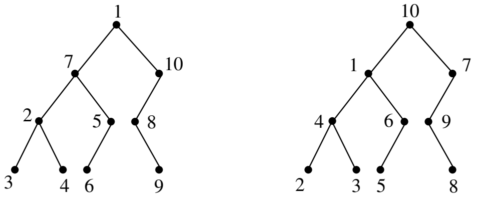
– *An ordering of the nodes of T is* max-min *if either T is empty, or the root node is last and the induced orderings on T^L and T^R are min-max. An ordering is* max-min-left-right *if it is both max-min and left-to-right.*

Property 1. The empty binary tree is vacuously both a min-max and a max-min search tree. A non-empty binary tree T with a value at each node is a min-max search tree (max-min search tree) if and only if the following conditions hold:

1. The minimum (maximum) value is at the root.
2. Every value in T^L is less than or equal to every value in T^R.
3. Both T^L and T^R are max-min (min-max) search trees.

Remark 1. Given a binary tree topology of size n and a set of n values, there is a unique min-max search tree on that topology which has those values. Similarly, there is a unique max-min search tree on that topology that has those values.

For example, given the tree topology shown in Figure 1, and the set of values $\{1, 2, \ldots, 10\}$, the unique min-max search tree on that topology is given in Figure 1(a), and the unique max-min search tree is given in Figure 1(b).



(a) Min-max search tree (b) Max-min search tree

Fig. 1. A min-max and max-min search tree

3.1 Operations on the M^2ST Data Structure

The min-max search tree (M^{2}ST) is an interesting concept in its own right. It allows for all the usual operations of a search structure, as well as functioning as min-max heap if required.

Another advantage of an M^2ST is that checking whether a given value x is within the range of the values stored in a binary tree T takes $O(h)$ rounds if the tree has the BST order, but only one round if the tree is an M^2ST.

We now provide the code for the usual operations on data structures: find, insert, and delete.

One additional advantage of an M^2ST is that it is easy to maintain balance while inserting an arbitrary sequence of values.

```
Min-max search tree
find( , )::
  if  =   then return
  else if  =  (root( )) then return
    else if     (root( )) then return
      else if     ( (root)) then return find( ,  )
        else return find( ,  )

insert( , )::
  if  =   then  =      ( )
  else if     (root( )) then    ( ,  (root( ))); insert( ,  )
    else if  =  (root( )) then insert( ,  )
      else if (root) =   ∨  ≤  ( (root)) then insert( ,  )
        else insert( ,  )

delete( , )::
  if  =   then return
  else if  =  (root( )) then
      if   =  ∧   =   then       ( )
      else  (root( )) =  (root( )++); delete( (root( )++),    ( )++)
    else if     (root( )) then return
      else if (root) ≠   ∧  ≤  ( (root)) then delete( ,  )
        else delete( ,  )

Max-min search tree
find( , )::
  if  =   then return
  else if  =  (root( )) then return
    else if     (root( )) then return
      else if     ( (root)) then return find( ,  )
        else return find( ,  )

insert( , )::
  if  =   then  =      ( )
  else if     (root( )) then    ( ,  (root( ))); insert( ,  )
    else if  =  (root( )) then insert( ,  )
      else if (root) =   ∨  ≤  ( (root)) then insert( ,  )
        else insert( ,  )

delete( , )::
  if  =   then return
  else if  =  (root( )) then
      if   =  ∧   =   then       ( )
      else  (root( )) =  (root( )−−); delete( (root( )−−),    ( )++)
    else if     (root( )) then return
      else if (root) ≠   ∧     ( (root)) then delete( ,  )
        else delete( ,  )
```

We now give the details of this balanced insert operation.

More generally, let *insertOK*(v) be a predicate which means that it is allowed to insert a value into the subtree rooted at v. There are many several ways to implement this predicate, for example:

1. *insertOK*(v) could mean that the height of T_v is no greater than the height of its sibling subtree.
2. *insertOK*(v) could mean that the number of values currently in T_v is no greater than the number of values currently in its sibling subtree.
3. Suppose that we are implementing a search structure using a fixed binary tree topology, perhaps hard-wired to a chip. Then *insertOK*(v) could mean that T_v has a vacancy, *i.e.*, a process where no current value is stored.

The above code can then be rewritten to use the predicate *insertOK*(v) instead of specifically asking about heights. In the third case, *i.e.*, where the binary tree topology is fixed, the structure will never experience false overflow; meaning that insertion can always take place if there is a null node anywhere in the tree.

```
Min-max search tree
balancedinsert( , )::
  if  =   then  =          ( )
    else
      if       (root( )) then       ( ,   (root( )))
      /* if    and    have the same height, then insert   appropriately */
      if      =       then
        if  ≤  ( (root)) then balancedinsert( ,     )
        else balancedinsert( ,     )
      /* if    has smaller height, then insert   into    if the  2   property is satisfied,
          else move the minimum value from    into    and then insert   into    */
      else if                 then
        if  ≤  ( (root)) then balancedinsert( ,     )
        else
          /* move the smallest value from    into    */
                =  ( (root)++)
          balancedinsert(    ,    )
          delete(    ,    )
          /* insert   into    */
          balancedinsert( ,    )
      /* if    has greater height, then insert   into    if the  2   property is satisfied
          else move the maximum value from    and then  ( (root)), into    and
      insert   into    */
      else if                 then
        if  ≥  ( (root)) then balancedinsert( ,     )
        else
          /* move  ( (root)) into    */
                =  ( (root))
          balancedinsert(    ,    )
          delete(    ,    )
          /* insert   into    */
          balancedinsert( ,    )
```

4 A Self-stabilizing Algorithm for Building a Min-Max Search Tree

In this section, we describe two self-stabilizing asynchronous distributed algorithms $\mathcal{A}_{\min}$ and $\mathcal{A}_{\max}$, which sort a binary tree network T into a min-max search tree and a max-min search tree, respectively. We can think of these as just one algorithm $\mathcal{A}(\textsc{parity})$, where $\mathcal{A}(0) = \mathcal{A}_{\min}$ and $\mathcal{A}(1) = \mathcal{A}_{\max}$.

$\mathcal{A}$ takes $O(n)$ rounds, where n is the number of processes of T, and requires $O(1)$ additional bits per process.

We say that v is at a Min-*level* if v is required to hold the minimum value of T_v after sorting, and that v is at a Max-*level* if v is required to hold the maximum value of T_v after sorting. Thus, the root of T is at a Min-level if T is to be sorted into a min-max search structure, and at a Max-level if T is to be sorted into a max-min search structure; and if v is not the root, v is at a Min-level if its parent is at a Max-level and vice-versa. Each process v has a *level bit*, *level*(v), which must, after stabilization, be 0 if that process is at a Min-level and 1 if that process is at a Max-level. We say a level bit is *correct* if it has the value that it must have after stabilization. We say that the level bit of v is *consistent* if either it is correct and v is the root, or it is different from the level bit of $p(v)$, the parent of v. Note that, although a level bit could be both consistent and incorrect, all level bits are consistent if and only if all level bits are correct.

The *successor process* of a process v in a min-max (or max-min) search tree is the process which is the successor of p in the min-max (or max-min) left-right order of the tree. The *predecessor process* of a process is defined similarly. Note that the successor process of p depends only on its position (with respect to topology), not on the values held in the processes. $\mathcal{A}$ works by constructing a *virtual chain* consisting of all the processes of T in min-max-left-right order, and then emulating an asynchronous distributed chain sorting algorithm on that virtual chain. This requires each process to "pretend" that it is adjacent to its predecessor and successor processes, although there may actually be as many as two intervening processes between them. For example, given the tree topology in Figure 1, the virtual chain obtained by considering T in min-max-left-right order is given in Figure 4(a), and the virtual chain obtained by considering T in max-min-left-right order is given Figure 4(b). To emulate the chain sorting algorithm, intervening processes must relay messages. We refer to the emulation of adjacency as a "virtual link" from a process to its successor or predecessor process. An emulated action along this link takes $O(1)$ rounds, as we explain below. In Subsection 5.1, we show, in detail, how the successor process of any process in a min-max or max-min search tree is computed.

The initial state is arbitrary. The root process knows it is the root, and can thus correct its level bit in one round. Correctness of all level bits descends from the root in a wave in $O(h)$ rounds. Using its level bit, each process x knows how to send a message to its successor process, which we call x^{++}, or to its predecessor process, which we call x^{--}. Figure 2 illustrates the definition of x^{++} in the case that x is at a Min-level, while Figure 3 illustrates the definition of x^{++} in the case that x is at a Max-level. (In both figures we represent a nil link as a short double-crossed line segment.) If a figure does not indicate either a child or a nil link in a particular place, then either possibility is allowed. For example, the right pointer from the middle node of Figure 3(a) could be nil, or could point to a node.

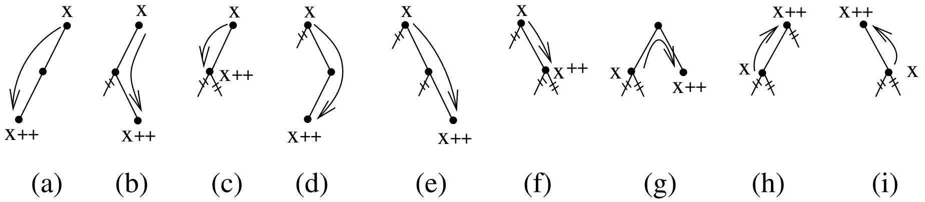


Fig. 2. Process x is a Min-level process

We do not actually define the needed message-passing and swapping protocols in this paper, but the following two properties guarantee they can be defined so that each needed operation can be executed in $O(1)$ rounds.

Property 2. For any process x in T, either x is the last process in the min-max-left-right order, or x^{++} has at most distance 3 from x.

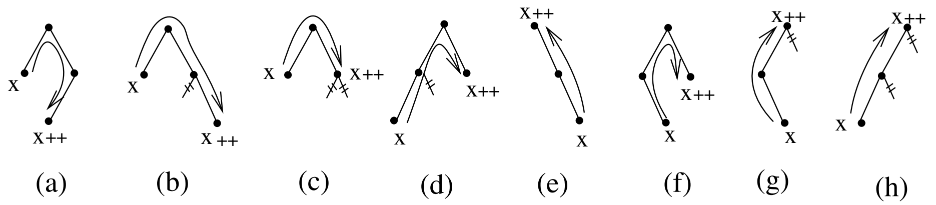


Fig. 3. Process x is a MAX-level process

Property 3. For any process y in T, there are at most two choices of x such that y lies on the interior of the shortest path from x to x^{++}. We call y a *relay process* of that virtual link.

Lemma 1. *The amount of memory necessary in every process to maintain the virtual chain is constant ($O(1)$ bits per process).*

Proof. By Properties 2 and 3, every process x must keep routing information for at most than six virtual pointers: the pointers to its predecessor x^{--} and its successor x^{++}, and at most four virtual pointers that pass through x. Thus, the additional memory needed by each process is finite.

4.1 Asynchronous M^2ST

Let T be a binary tree network.

We start by choosing $\mathcal{S}$ to be any asynchronous distributed sorting algorithm on an oriented chain which is self-stabilizing under the unfair daemon, and which stabilizes in $O(n)$ rounds from an arbitrary initial configuration, and which uses $O(1)$ bits per node in addition to the stored values. Our technique is to emulate $\mathcal{S}$ on the virtual chain of T, while increasing the time of the algorithm by only a constant factor. We do not give the details of this emulation, rather, we only prove that such an emulation exists.

We will need a predicate *OKtoexecute*(v), which returns *true* if the emulation of the next action of $\mathcal{S}$ on v is ready to commence, *false* otherwise. The action *execute*(v) commences the emulation of the next action of $\mathcal{S}$ on v, whatever that may be. The exact details of this predicate and this action depend on the details of the emulation of $\mathcal{S}$, which we do not give in this paper.

Predicate *levelOK* returns *true* if the level bit of v is consistent, otherwise *false.*

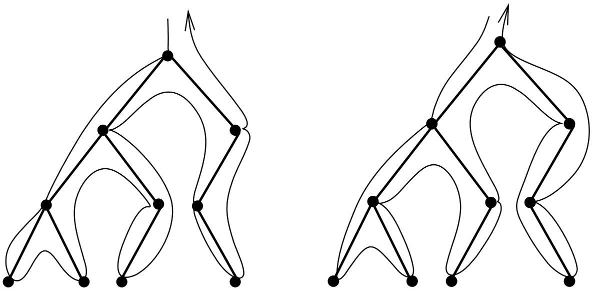
Action L_r sets the level bit of the root to be 0. Action L_{nr} sets the correct levels for all other processes. Action S sorts the value of the process and the value of successor.

Property 4. For any process x, the processes of T_x form an interval in the virtual chain such that:
(i) If x is a MIN-level process, then process x is the first process of the interval.
(ii) If x is a MAX-level process, then process x is the last process of the interval.

Algorithm 4.1. Algorithm $\mathcal{A}(0)$

```
Predicate $levelOK(v,l) \equiv ((p(v) = \bot \wedge level(v) = 0) \vee (p(v) \neq \bot \wedge level(v) \neq level(p(v))))$

Actions for any process $v$
    $p(v) = \bot \wedge \neg levelOK(v) \longrightarrow level(v) = 0$
    $p(v) \neq \bot \wedge \neg levelOK(v) \longrightarrow level(v) = \neg level(p(v))$
    $level\_OK(v) \wedge v^{++} \neq \bot \wedge levelOK(v^{++}) \wedge OKtoexecute(v) \longrightarrow execute(v)$
```



(a) Min-max-left-right order (b) Max-min-left-right order

Fig. 4. The Virtual Chain

Let T^{min} be the tree in which the root is a Min-level process, and the values in the processes are sorted in ascending order of the min-max-left-right chain. Let T^{max} be the tree in which the root is a Max-level process, and the values in the processes are sorted in ascending order of the max-min-left-right chain.

Property 5. Let T^{sort} be either T^{min} or T^{max}.
For any process $x \in T$, one of the following is *true*:

(i) if x is a Min-level process, then $V(x)$ is the minimum value in T_x^{sort} and T_x^{sort} has left-right order.

(ii) if x is a Max-level process, then $V(x)$ is the maximum value in T_x^{sort} and T_x^{sort} has left-right order.

Proof. For *(i)*, by Property 4, process x is the first process in the min-max-left-right chain. By applying Algorithm $\mathcal{A}(0)$ to T^{sort}, in at most $O(n)$ rounds the values in the chain are sorted in non-descending order, thus $V(x)$ will hold the minimum value in the chain, and subtree T_x.

For *(ii)*, by Property 4, process x is the last process in the min-max-left-right chain. By applying Algorithm $\mathcal{A}(1)$, in at most $O(n)$ rounds to T^{sort}, the values in the chain are sorted in non-descending order. thus $V(x)$ will hold the maximum value in the chain, and subtree T_x.

In summary, we have:

Theorem 1. *A binary tree network of n nodes and height h can be sorted into min-max-left-right (or max-min-left-right) order in $O(n)$ rounds and $O(1)$ bits per process in the asynchronous model, using an unfair daemon.*

Note that the minimum number of rounds needed for any M^{2}ST algorithm using the same computational model is $\Omega(n)$ in the worst case, since it might be necessary to move almost every value through the root.

5 Asynchronous BST Construction

We define algorithms $\mathcal{B}(0)$ and $\mathcal{B}(1)$ that run on an M^2ST, T. The result of either algorithm that the network is sorted into binary search tree order.

Algorithm $\mathcal{B}(0)$::

Step 0. Let T be in min-max-left-right order.
Note that the value of every process in T^R is greater than or equal to the value of every process not in T^R.
Therefore, we do not need to move values across the link from the root to T^R.
Let $\mathcal{B}(1)$ run independently on T^R.
If T^L is empty, we do nothing else. Otherwise, $V(\ell(root))$ is the value that belongs in *root*. We continue with Step 1.

Step 1. Swap the values of *root* and $\ell(root)$.
From now on, ignore the root.

Step 2. The last process in the *max-min-left-right chain* of T^L is $\ell(root)$, but it now holds the minimum item in T^L.
Create a circular linked list consisting of the *max-min-left-right chain* of T^L, together with one link from $\ell(root)$ to the first process in the chain.

Step 3. Push every item in the circular list one step forward.
The order to push descends the tree T^L in a wave: thus, a process of depth d in T^L will finish this step in $O(d)$ rounds.

Step 4. T^L is now in max-min-left-right order.
Let $\mathcal{B}(1)$ run independently on T^L.

For example, given the min-max search tree in Figure 1(a), by applying the steps of Algorithm $\mathcal{B}(0)$, the tree changes are presented in Figure 5.

```
Algorithm $\mathcal{B}(1)$::
Step 0.  Let $T$ be in max-min-left-right order.
         Note that the value of every process in $T^L$ is smaller than or equal to the
         value of every process not in $T^L$.
         Therefore, we do not need to move values across the link from the root to $T^L$.
         Let $\mathcal{B}(0)$ run independently on $T^L$.
         If $T^R$ is empty, we do nothing else. Otherwise, $V(r(root))$ is the value
         that belongs in root. We continue with Step 1.

Step 1.  Swap the values of  root  and $r(root)$.
         From now on, ignore the root.

Step 2.  The first process in the min-max-left-right chain of $T^R$ is $r(root)$, but it now
         holds the maximum item in $T^R$.
         Create a circular linked list consisting of the min-max-left-right chain of $T^R$,
         together with one link from $r(root)$ to the last process in the chain.

Step 3.  Push every item in the circular list one step backward.
         The order to push ascends the tree $T^R$ in a wave: thus, a process of
         depth $d$ in $T^R$ will finish this step in $O(d)$ rounds.

Step 4.  $T^R$ is now in min-max-left-right order.
         Let $\mathcal{B}(0)$ run independently on $T^R$.
```

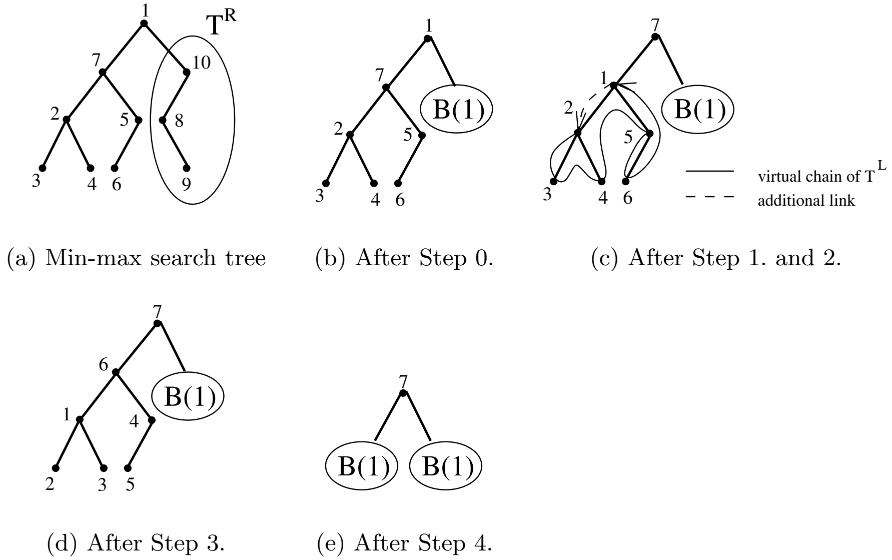
Given that the subtree T^L of the tree in Figure 5(d) is max-min-left-right order, the tree changes resulting from applying Algorithm $\mathcal{B}(1)$ are presented in Figure 6.

Lemma 2. *A binary tree T in min-max-left-right order or max-min-left-right order can be sorted into BST order in $O(h)$ rounds in the asynchronous model, using $O(1)$ bits per process.*

Proof. The remarks in Step 0. follow from the definition of an M^2ST tree. No actions are executed during this step. Step 1. requires one round. The remark in Step 2. follows from Property 5. Step 2. requires one round to add the extra link, since the virtual chain of T^L is already constructed and the two processes of the extra link are adjacent. Step 3. requires $O(h)$ rounds, since pushing the value of any process in the circular list occurs when the order to push, which descends the subtree in a wave, reaches that process. Step 4. requires no extra rounds.

Theorem 2. *A binary tree can be sorted into binary search tree (BST) order in $O(n)$ rounds in the asynchronous model, using only $O(1)$ states and $O(1)$ values in any process.*

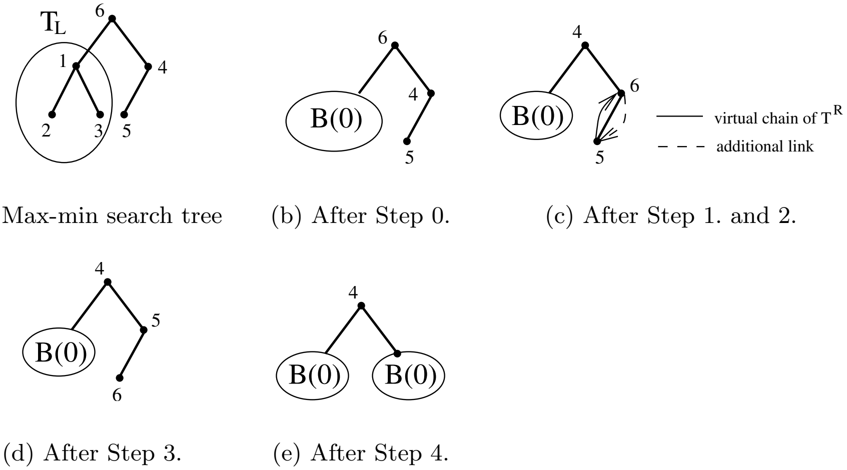
Theorem 2 follows from Lemma 2.



(a) Min-max search tree (b) After Step 0. (c) After Step 1. and 2.

(d) After Step 3. (e) After Step 4.

Fig. 5. Execution of Algorithm $\mathcal{B}(0)$



(a) Max-min search tree (b) After Step 0. (c) After Step 1. and 2.

(d) After Step 3. (e) After Step 4.

Fig. 6. Execution of Algorithm $\mathcal{B}(1)$

5.1 Finding the Successor of a Process in an M^{2}ST

Let x be the a process in a binary tree network T. Assume that the processes have already been partitioned into MIN-level and MAX-level processes. The successor process x^{++} is defined as follows.

MIN: Process x is a MIN-level process (see Figure 2).
a. Process $\ell(x)$ has a left child. Then $x^{++} = \ell(\ell(x))$ (Figure 2(a)).
b. Process $\ell(x)$ is an internal node that does not have a left child. Then $x^{++} = \ell(r(x))$ (Figure 2(b)).

c. Process $\ell(x)$ is a leaf node. Then $x^{++} = \ell(x)$ (Figure 2(c)).
d. Process x is an internal node without a left child, and process $r(x)$ has a left child. Then $x^{++} = r(\ell(x))$ (Figure 2(d)).
e. Process x is an internal node without a left child, and process $r(x)$ is an internal node without a left child. Then $x^{++} = r(r(x))$ (Figure 2(e)).
f. Process x is an internal node without a left child, and the process $r(x)$ is a leaf. Then $x^{++} = r(x)$ (Figure 2(f)).
g. Process x is a leaf node and is the left child of its parent $p(x)$, and process $p(x)$ has a right child. Then $x^{++} = r(p(x))$ (Figure 2(g)).
h. Process x is a leaf node and is the left child of its parent $p(x)$, and process $p(x)$ has no right child. Then $x^{++} = p(x)$ (Figure 2(h)).
i. Process x is a leaf node and is the right child of its parent $p(x)$. Then $x^{++} = p(x)$ (Figure 2(i)).
j. Process x is a leaf node and also the root. Then x^{++} is undefined.

Max: Process x is a Max-level process (see Figure 3).

a. Process x is the left child of its parent $p(x)$, and process $r(p(x))$ has a left child. Then $x^{++} = \ell(r(p(x)))$ (Figure 3(a)).
b. Process x is the left child of its parent $p(x)$ and the process $r(p(x))$ is an internal node without a left child. Then process $x^{++} = r(r(p(x)))$ (Figure 3(b)).
c. Process x is the left child of its parent $p(x)$, and the process $r(p(x))$ is a leaf node. Then process $x^{++} = r(p(x))$ (Figure 3(c)).
d. Process x is the left child of its parent $p(x)$, process $p(x)$ has no right child. and process $p(p(x))$ has a right child. Then $x^{++} = r(p(p(x)))$ (Figure 3(d)).
e. Process x is the right child of its parent $p(x)$, and process $p(x)$ is the right child of its parent $p(p(x))$. Then $x^{++} = p(p(x))$ (Figure 3(e)).
f. Process x is the right child of its parent $p(x)$, $p(x)$ is the left child of its parent $p(p(x))$, and process $p(p(x))$ has a right child. Then $x^{++} = r(p(p(x)))$ (Figure 3(f)).
g. Process x is the right child of its parent $p(x)$, $p(x)$ is the left child of its parent $p(p(x))$, and process $p(p(x))$ has no right child. Then $x^{++} = p(p(x))$ (Figure 3(g)).
h. Process x is the left child of its parent $p(x)$, process $p(x)$ has no right child, and process $p(p(x))$ has no right child. Then $x^{++} = p(p(x))$ (Figure 3(h)).
i. Process x is the root. Then x^{++} is undefined.
j. Process x is the left child of process $p(x)$, $p(x)$ is the root, and $p(x)$ has no right child. Then x^{++} is undefined.
k. Process x is the right child of process $p(x)$ and $p(x)$ is the root. Then x^{++} is undefined.

6 Conclusion

In this paper we define a data structure, M^2ST, which has the combined properties of a search structure and a min-max heap. We give a self-stabilizing algorithm for sorting a binary tree network into M^2ST order. The time to make an an arbitrary tree into an M^2ST is $O(n)$ rounds (Algorithm $\mathcal{B}$), and the algorithm needs $O(1)$ space.

Algorithm $\mathcal{A}$ reorganizes a min-max search tree on a binary tree network into a binary search tree in $O(h)$ rounds. Starting from an arbitrary state, by combining algorithms $\mathcal{A}(0)$, $\mathcal{B}(0)$, and $\mathcal{B}(1)$, we obtain a distributed algorithm to build a binary search tree on a binary tree network in an arbitrary state in $O(n)$ rounds, using only $O(1)$ bits per process.

References

1. A. Arvind and C.P. Rangan. Symmetric min-max heap: a simpler data structure for double-ended priority queue. *Information Processing Letters*, 69:197–199, 1999.
2. M. D. Atkinson, J.R. Sack, B. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.
3. D. Bein, A.K. Datta, and V. Villain. Snap-stabilizing optimal binary-search-tree. *Proceedings of the 7-th International Symposium on Self-Stabilizing Systems*, 2005.
4. S. Carlsson. The deap: - a double-ended heap to implement double-ended priority queues. *Information Processing Letters*, 26:33–36, 1987.
5. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
6. Y. Ding and M.A. Weiss. The relaxed min-max heap. *Acta Informatica*, 30:215–231, 1993.
7. S Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000.
8. S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
9. T. Herman and T. Masuzawa. Available stabilizing heaps. *Information Processing Letters*, 77:115–121, 2001.
10. T. Herman and T. Masuzawa. A stabilizing search tree with availability properties. *Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 398–405, 2001.
11. T. Herman and I. Pirwani. A composite stabilizing data structure. *5th International Workshop on Self-Stabilizing Systems (WSS 2001), Lecture Notes in Computer Science LNCS 2194, Springer Verlag*, pages 167–182, 2001.
12. C.M. Koong and H.W. Leong. Double-ended binomial queues. *Proceedings of ISAAC*, pages 128–137, 1993.
13. C. Makris, A. Tsakalidis, and K. Tsichlas. Reflected min-max heaps. *Information Processing Letters*, 86(4):209–214, 2003.

Efficient Dynamic Aggregation*

Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster

Technion – Israel Institute of Technology, Haifa 32000, Isreal
{birk@ee, idish@ee, liranl@tx, assaf@cs}.technion.ac.il

Abstract. We consider the problem of *dynamic aggregation* of inputs over a large fixed graph. A dynamic aggregation algorithm must continuously compute the result of a given aggregation function over a dynamically changing set of inputs. To be efficient, such an algorithm should refrain from sending messages when the inputs do not change, and should perform *local* communication whenever possible.

We present an instance-based lower bound on the efficiency of such algorithms, and provide two algorithms matching this bound. The first, MultI-LEAG, re-samples the inputs at intervals that are proportional to the graph size, achieving quiescence between samplings, and is extremely message efficient. The second, DynI-LEAG, more closely monitors the aggregate value by sampling it more frequently, at the cost of slightly higher message complexity.

1 Introduction

We consider the problem of continuous monitoring of an aggregation function over a set of dynamically changing inputs on a large fixed graph. We term this problem *dynamic aggregation*. For example, the inputs may reflect sensor readings of temperature or seismic activity, or load reported by computers in a computational grid. The aggregation function may compute the average temperature, or whether the percentage of sensors that detect an earthquake exceeds a certain threshold, or the maximum computer load. It is desirable to seek *local* solutions to this problem, whereby input values and changes thereof do not need to be communicated over the entire graph.

Since virtually every interesting aggregation function has some input instances on which it cannot be computed without global communication, a priori, it is not clear whether one can do better. Nevertheless, we have recently shown that when computing an aggregation function on a large graph for fixed (in time) inputs, it is often possible to reach the correct result without global communication [1]. Specifically, while *some* problem instances trivially require global communication, many instances can be computed locally, i.e., in a number of steps that is independent of the graph size. We introduced a classification of instances according to a measure called *Veracity Radius* (VR), which captures the degree to which a problem instance is amenable to local computation. The VR

* This work was supported in part by a grant from the Israel Ministry of Science.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 90–104, 2006.

is computed by examining the *r-neighborhood* of a node v, which is the set of all nodes within radius r from v. Roughly speaking, the VR identifies the minimum neighborhood radius r_0, such that for all neighborhoods with radius $r \geq r_0$ the aggregation function yields the same value as for the entire graph. (The formal definition of VR allows some slack in the environments over which the aggregate function is computed.) VR provides a tight lower bound on computation time. In addition, [1] presents an efficient aggregation algorithm, I-LEAG, which achieves the lower bound up to a constant factor.

The results of [1], however, are restricted to the computation of a *static* aggregation instance, and do not directly extend to dynamic aggregation. If I-LEAG is to be used in a dynamic setting, the entire computation must be periodically invoked anew, even if no inputs change. Specifically, all nodes must periodically send messages to their neighbors, which can lead to considerable waste of resources, especially when input changes are infrequent.

In this paper, we extend the results of [1] to deal with dynamic aggregation. We focus on algorithms that continuously compute the result of a given aggregation function at each node in the graph, and satisfy the following requirements: (1) the algorithm's output converges to the correct result in finite time once all input changes cease; and (2) once the algorithm has converged, no messages are sent as long as the input values persist.

In Sect. 3, we derive a lower bound on computation time for dynamic aggregation algorithms satisfying the above requirements. We show that if an algorithm has converged for some input I^{old}, and subsequently the inputs change to I^{new}, then the computation of I^{new} must take a number of steps that is proportional to the maximum between the VRs of I^{old} and I^{new}. The lower bound is proven for both the time until the correct result is observed at all nodes (*output stabilization time*) and the time until no messages are sent (*quiescence time*).

We provide two efficient dynamic aggregation algorithms that achieve this lower bound up to a constant factor. Our algorithms employ the basic principles of I-LEAG, but are more involved as they need to refrain from sending messages when there are no changes.

In Sect. 4, we consider a scenario wherein it suffices to update the output reflecting the aggregation result periodically, e.g., every few minutes. For this setting, we present MultI-LEAG, which operates in a multi-shot fashion: the inputs are sampled at regular intervals, and the correct (global) result relative to the last sample is computed before the next sample is taken. The sampling interval is proportional to the graph diameter. MultI-LEAG selectively caches values according to the previous input's VR to avoid sending messages when the inputs do not change. After every sample, MultI-LEAG reaches both output-stabilization and quiescence in time proportional to the lower bound, which never exceeds the sampling interval and may be considerably shorter. We call this *sample-compute-output* cycle an iteration. MultI-LEAG is very efficient, and does not send more messages than necessary.

In Sect. 5, we consider a scenario wherein the output must reflect the correct aggregation value promptly. That is, the input is sampled very frequently, e.g., at intervals on the order of a single-hop message latency between neighboring nodes, and not proportional to the graph's diameter as in MultI-LEAG. For this setting, we present DynI-LEAG, which invokes multiple MultI-LEAG iterations in parallel. Although each MultI-LEAG iteration is comprised of several phases with different durations, DynI-LEAG manages to carefully pipeline a combination of complete and partial MultI-LEAG iterations to achieve $O(\log^2(\mathit{diameter}))$ memory usage per node. Note that DynI-LEAG inspects multiple input samples during the time frame in which MultI-LEAG conducts a single sample. The corresponding lower bound on algorithms that operate in this mode reflects not only two inputs, I^{old} and I^{new}, as described above, but rather all inputs sampled within a certain time window.

There is a tradeoff between our two algorithms: whereas MultI-LEAG delivers correct results corresponding to relatively old snapshots, DynI-LEAG closely tracks the aggregate result at the expense of a somewhat higher message complexity. Nevertheless, the total number of messages sent in both algorithms depends only on the actual number of input changes and on the VR values of recent inputs but not on the system size.

Related work. Following the proliferation of large-scale distributed systems such as sensor networks [2,3], peer-to-peer systems [4], and computational grids [5], there is growing interest in methods for collecting and aggregating the massive amount of data that these systems produce, e.g., [6,7,8,9,10]. The semantics of validity for dynamic aggregation have been discussed in [11]. However, most of this work has not dealt with locality.

The initial work on using an "instance-based" approach to solve seemingly global problems in a local manner has focused on self-stabilization [12,13,14]. Instance-local solutions have also been proposed for distributed error confinement [15], location services [16] and Minimum Spanning Tree [17]. The first work that demonstrated instance-local aggregation algorithms by means of an empirical study is [18,19]. Only recently, instance-local aggregation has been formalized [1]. However, this work did not consider dynamic scenarios.

2 Preliminaries

Model and Problem Definition. Given a set D, we denote a multi-set over D by $\{d_1^{n_1}...d_m^{n}\}$, where $d_i \in D$ and $n_i \in \mathbb{N}$ indicates the multiplicity of d_i. We denote the set of multi-sets over D by $\mathbb{N}^D$. An *aggregation function* is a function F:$\mathbb{N}^D \to R$, where R is a discrete totally-ordered set, and F satisfies the following: (i) *convexity*: $\forall X, Y \in \mathbb{N}^D$: $F(X \cup Y) \in \Big[F(X), F(Y)\Big]$; and (ii) *onto* (in singletons): $\forall r \in R, \exists x \in D$: $F(x) = r$. Many interesting functions have these properties, e.g., min, max, majority, median, rounded average (with a discrete range) and consensus (e.g., by using OR/AND functions).

We model a distributed system as a fixed undirected graph $G = G(V, E)$. Computation proceeds in synchronous rounds in which each node can communicate with its immediate neighbors. A graph G and an aggregation function F

define the *aggregation problem* $P_{G,F}$ as follows: Every node v has an input value $I_v \in D$, which can change over time, and an output register $O_v \in R \cup \{\bot\}$. Initially, $O_v = \bot$ and v only knows its own input. We denote by $I(t)$ the input assignment (of all nodes) at time t. For a set of nodes $X \subseteq V$, we denote by I_X the multi-set induced by the projection of I on X, e.g., $I_V = I$. Assume that there exists a time t_0 such that $\forall t \geq t_0$: $I(t) = I(t_0)$. An algorithm *solves* $P_{G,F}$ if it has finite output-stabilization and quiescence times after t_0, and its final outputs are $\forall v \in V$: $O = F(I(t_0))$.

For a multi-shot algorithm A, given two consecutive sampled input assignments I^{old} and I^{new}, we denote by $OS_A(I^{\text{old}}, I^{\text{new}})$ and $Q_A(I^{\text{old}}, I^{\text{new}})$ the output-stabilization and quiescence times, respectively, following I^{new}. In the general case, we denote by $OS_A(\mathcal{I})$ and $Q_A(\mathcal{I})$ the output-stabilization and quiescence times for an infinite input sequence $\mathcal{I}$ in which the inputs do not change after some time t_0.

Finally, we note that every aggregation function can be represented as a tuple $F = \langle \widehat{R}, F_I, F_{agg}, F_O \rangle$, where: $\widehat{R}$ is some internal representation, and $F_I{:}D \rightarrow \widehat{R}$, $F_{agg}{:}\widehat{R}^n \rightarrow \widehat{R}$ and $F_O{:}\widehat{R} \rightarrow R$ are functions such that for every set of nodes $V = \{v_1, ..., v_n\}$ and an input assignment I:

$$F(I_V) = F_O\Big(F_{agg}\big(\{F_I(I_v) \mid v \in V\}\big)\Big).$$

In many cases, the internal representation $\widehat{R}$ can be extremely compact. For example, for computing OR, it can be a single bit, and for simple majority voting, the number of "yes" and "no" votes.

Graph Notions. Let $G = G(V, E)$ be a graph. Denote G's diameter and radius by $Diam(G)$ and $Rad(G)$, respectively. We use the following graph-theoretic notation:

Cluster. A subset $S \subseteq V$ of vertices whose induced subgraph $G(S)$ is connected.

Distance. For every two nodes $v_1, v_2 \in V$, the distance between v_1 and v_2 in G, $dist(v_1, v_2)$, is the length of the shortest path connecting them.

Neighborhood. The $r-$neighborhood ($r \in \mathbb{R}^+$) of a node v, $\Gamma_r(v)$, is the set of nodes $\{v' \mid dist(v, v') \leq r\}$. $\widehat{\Gamma}(v) = \Gamma_1(v) - \{v\}$ denotes the neighbors of a node v. For a cluster S: $\Gamma_r(S) = \bigcup_{v \in S} \Gamma_r(v)$ and $\widehat{\Gamma}(S) = \Gamma_1(S) - S$.

3 Lower Bound

In [1], we introduced an inherent metric for locality, the *Veracity Radius* (VR), which is defined as follows. A K-bounded *slack function*, is a non-decreasing continuous function $\alpha{:}\mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that $\alpha(r) \in [\frac{r}{K}, r]$, for some $K \geq 1$. Given a graph G and an aggregation function F, the VR (parameterized by a slack function α) of an input instance I is:

$$VR_\alpha(I) \triangleq min\{r \in \mathbb{R}^+ \mid \forall r' \geq r, v \in V, S \subseteq V \text{ s.t. } \Gamma_{\alpha(r')}(v) \subseteq S \subseteq \Gamma_{r'}(v){:}$$
$$F(I_S) = F(I)\}.$$

Simply speaking, VR identifies the minimum neighborhood radius r_0 such that for all neighborhood-like environments with radius $r \geq r_0$ (i.e., all subgraphs S that include an $\alpha(r)$-neighborhood and are included in an r-neighborhood), the aggregation function yields the same value as the entire graph. If $F(I_v) = F(I)$ for every $v \in V$, then $VR(I) = 0$ and I is called a *trivial* input assignment.

Given an aggregation problem $P_{G,F}$, we proved in [1] that for every $r \geq 0$, every slack function α and every deterministic algorithm A that solves P, there exists an assignment I with $VR_\alpha(I) \leq r$ for which $OS_A(I) \geq min\{\lfloor\alpha(r)\rfloor, Rad(G)\}$. A similar bound was also proven for quiescence. However, this *single-shot* lower bound is overly restrictive for dynamic systems because it ignores previous inputs. We now show that for dynamic aggregation, in which an algorithm is not allowed to send messages after it converges, both current and previous inputs are inherent to computation time. Due to lack of space, the proofs are detailed in the full paper [20].

For multi-shot algorithms, in which convergence is guaranteed following every input sample, it suffices to consider only the two latest input samples:

Theorem 1 (Multi-shot Lower Bound). *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{\text{old}}, r^{\text{new}} \geq 0$ such that $\alpha(r^{\text{old}}), \alpha(r^{\text{new}}) \leq Rad(G)$, and every deterministic multi-shot algorithm A that solves F, there exist two input samples $I^{\text{old}}, I^{\text{new}}$ such that $VR_\alpha(I^{\text{old}}) \leq r^{\text{old}}$, $VR_\alpha(I^{\text{new}}) \leq r^{\text{new}}$, and $OS_A(\{I^{\text{old}}, I^{\text{new}}\}) \geq max\{\lfloor\alpha(r^{\text{old}})/6\rfloor, \lfloor\alpha(r^{\text{new}})\rfloor\}$. The same holds for quiescence.*

For algorithms that do not necessarily converge between consecutive samples, the multi-shot lower bound implies that the effects of an input assignment may impact algorithm performance during multiple future samples; the duration of these effects is proportional to the input's VR:

Corollary 1 (Dynamic Lower Bound). *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{\text{old}}, r^{\text{new}} \geq 0$ such that $\alpha(r^{\text{old}}), \alpha(r^{\text{new}}) \leq Rad(G)$, every constant $C \geq 1$, and every deterministic algorithm A that solves F, there exist an input sequence $\mathcal{I}$ and time t_0 such that: (1) $\forall r > r^{\text{old}}$: for every $t \in [t_0 - C \cdot r, t_0)$, $VR(I(t)) < r$; (2) $VR(I(t_0)) \leq r^{\text{new}}$; and (3) $\forall t \geq t_0$: $I(t) = I(t_0)$; for which $OS_A(\mathcal{I}) \geq max\{\lfloor\alpha(r^{\text{old}})/6\rfloor, \lfloor\alpha(r^{\text{new}})\rfloor\}$. The same holds for quiescence.*

Finally, we note that for output-stabilization, these bounds are nearly tight: in [20], we show how *full information* (FI) protocols, in which every node broadcasts all input changes to all other nodes, achieve $O(max\{\lfloor\alpha(r^{\text{old}})\rfloor, \lfloor\alpha(r^{\text{new}})\rfloor\})$ output-stabilization time (for both multi-shot and ongoing operation), albeit at high memory usage and communication costs. Nevertheless, eventual quiescence is still guaranteed.

4 MultI-LEAG: An Efficient Multi-shot Aggregation Algorithm

We now introduce MultI-LEAG, an efficient aggregation algorithm that operates in a multi-shot fashion. MultI-LEAG is quiescent and maintains fixed outputs when the input does not change, while leveraging the veracity radius of the inputs to reach fast quiescence and output stabilization when changes do occur. This enables MultI-LEAG to achieve an extremely low communication complexity, which depends only on the number of changes and the VR of the previous and current input samples, rather than on graph size.

Let $G = G(V, E)$ be a graph, and let $\Lambda_\theta = \lceil \log_\theta(Diam(G)) \rceil$. In order to operate, MultI-LEAG requires a (θ, α)-local partition hierarchy of G, which was first defined in [1] and utilized by the I-LEAG algorithm:

Definition 1 ((θ, α)-Local Partition Hierarchy (LPH)). *Let $\theta \geq 2$ and let α be a slack function. A (θ, α)-local partition hierarchy of a graph G is a triplet $\langle \{\mathcal{S}_i\}, \{\mathcal{P}_i\}, \{\mathcal{T}_i\} \rangle, 0 \leq i \leq \Lambda_\theta$, where:*

- *$\{\mathcal{S}_i\}$ is a set of partitions, in which for every cluster $S' \in \mathcal{S}_{i-1}$ there exists a cluster $S \in \mathcal{S}_i$ such that $S' \subseteq S$. The topmost level, $\mathcal{S}_{\Lambda}$, contains a single cluster equal to V. Denote by $S_i(v)$ the cluster $S \in \mathcal{S}_i$ such that $v \in S$.*
- *$\{\mathcal{P}_i\}$ is a set of pivot sets. $\mathcal{P}_i$ includes a single pivot (sometimes called cluster head) for every cluster $S \in \mathcal{S}_i$. For every $p \in \mathcal{P}_i$, denote $Sub_{i-1}(p) = \{p' \in P_{i-1} \mid p' \in S_i(p)\}$.*
- *$\{\mathcal{T}_i\}$ is a set of forests. For every $p \in \mathcal{P}_i$, $\mathcal{T}_i$ contains a directed tree $T_i(p)$ whose root is p and whose leaves are either $Sub_{i-1}(p)$ or the nodes in $S_0(p)$ if $i = 0$. For every $i > 0$, denote by $\widetilde{T}_i(p)$ the* logical tree *formed by concatenating $T_i(p)$ and $\widetilde{T}_{i-1}(p')$ at every $p' \in Sub_{i-1}(p)$, where $\forall p' \in \mathcal{P}_0$: $\widetilde{T}_0(p') = T_0(p')$.*

In addition, the following conditions must hold for every $p \in \mathcal{P}_i$, $S_i(p) \in \mathcal{S}_i$, and $T_i(p) \in \mathcal{T}_i$: (1) $\Gamma_{\alpha(\theta\ )}(p) \subseteq S_i(p) \subseteq \Gamma_{\theta}\ (p)$; (2) $T_i(p) \subseteq S_i(p)$; (3) the height of $\widetilde{T}_i(p)$ is at most θ^i.

Apart from the second condition, this definition of an LPH is identical to [1], which provides general LPH construction algorithms. Although we can do without it, it greatly simplifies the presentation. Note that this condition also implies that clusters must be connected within themselves (i.e., clusters are not *weak* [1]).

An LPH can be computed once per graph, and used for any duration and any aggregation function. We next introduce two notions that link an aggregation problem and an LPH for it, which are closely related to VR:

Cluster in conflict. Let $P_{G,F}$ be an aggregation problem. Given an input assignment I and an LPH for G, for every level $i > 0$, a cluster $S \subseteq \mathcal{S}_i$ is *in conflict* if at least two of the level-$(i-1)$ clusters that constitute S have different aggregate results. Level-0 clusters are always considered in conflict.

Algorithm 1. (MultI-LEAG) for node $v \in V$

Parameters: $F{:}\mathbb{N}^D \to R$, (θ, α)-local hierarchy $\langle \{\mathcal{S}_i\}, \{\mathcal{P}_i\}, \{\mathcal{T}_i\} \rangle, 0 \le i \le \Lambda_\theta$ of $G(V, E)$
Input: $I_v \in D$
Output: $O_v \in R \cup \{\perp\}$ initially $\perp$
Definitions: $\mathcal{P}_{i-1} \triangleq V$, $\mathit{Phases} \triangleq \{-1, 0, ..., \Lambda_\theta\}$,
$\mathit{Tree}^+ \triangleq \bigcup_{i,p\in\mathcal{P}_i} T_i(p)$ ignoring edge directions (i.e., $\mathit{Tree}^+ \subseteq E$),
$\widehat{S}_i(v) \triangleq S_i(v) \;\cup\; \{w \in \widehat{\Gamma}(S_i(v)) \mid \exists u \in S_i(v){:}\ (u, w) \in \mathit{Tree}^+\}$
Variables:
$\forall u \in \widehat{\Gamma}(v){:}\ O_v^u \in R \cup \{\perp\}$ initially $\perp$,
VP_v, $VP_v^{\text{new}} \in \mathit{Phases}$ initially 0,
$\mathit{Conf}_v(i){:}\mathit{Phases} \to \{\mathsf{true}, \mathsf{false}\}$, initially true for $i = 0$ and false otherwise,
$\mathit{Agg}_v(i){:}\mathit{Phases} \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$\mathit{Agg}_v^{\text{sent}}(i){:}\mathit{Phases} \to \widehat{R} \cup \{\perp\}$ initially $\perp$,
$\mathit{Agg}_v^{\text{recv}}(i, p){:}\mathit{Phases} \times V \to \widehat{R} \cup \{\perp\}$ initially $\perp$

Synchronous phases:

```
loop    /* forever */
    $Agg_v(-1) \leftarrow F_I(I_v)$    /* read changes in input */
    $\forall i > 0$: $Conf_v(i) \leftarrow$ false
    $VP_v \leftarrow VP_v^{new}$
    for phase $i = 0$ to $\Lambda_\theta$ do
        do-phase(i)
```

Veracity Level (VL). Let $P_{G,F}$ be an aggregation problem. Given an input assignment I and an LPH for G, a node v's *Veracity Level* is defined as:

$$VL_v(I) \triangleq max\{i \in [0, \Lambda_\theta] \mid S_i(v) \text{ is in conflict}\}.$$

It directly follows from convexity that the aggregate result of any level-i cluster whose nodes' VL is i, equals the global outcome. We denote by $VL(I)$ the maximum VL over all nodes.

MultI-LEAG is presented in Algorithm 1.. It is provided with an LPH, and uses two procedures, do-phase and converge-cast, which are depicted in Algorithms 2. and 3., resp. Code in gray only applies to the DynI-LEAG algorithm presented in the next section, which also uses these procedures. Apart from its input I_v and output register O_v, every node v holds the following variables: O_v^u, the output of every neighbor $u \in \widehat{\Gamma}(v)$, VP_v and VP_v^{new}, v's veracity phase (used to compute v's VL as explained shortly) in the previous and current input samples, resp. Additionally, for every level i in which v is a pivot, v holds the following mappings: Conf_v, a boolean indicating if $S_i(v)$ is in conflict; Agg_v, the internal aggregate representation of the input values in $S_i(v)$; $\mathit{Agg}_v^{\text{sent}}$, the last value of Agg_v sent to v's pivot in the next level; and $\mathit{Agg}_v^{\text{recv}}$, the last internal representation received from every $p' \in \mathit{Sub}_{i-1}(v)$.

Algorithm 2. (do-phase procedure) for node $v \in V$

```
Function do-phase($i, t$)
set timer to $5\theta^i$
let $p \in \mathcal{P}_i$ s.t. $v \in S_i(p)$
if $i > VP_v^v(t)$ then    /* fall back to I-LEAG */
    if $v \in T_i(p) \ \wedge\ \exists u \in \widehat{\Gamma}(v)$ s.t. $u$ is $v$'s parent in $T_i(p)$ and $O_v^v(t) \neq O_v^u(t)$ then
        send $\langle$conflict,i,p,t$\rangle$ to $u$
else    /* $i \leq VP_v^v(t)$ */
    if $v \in Sub_{i-1}(p) \ \wedge\ Agg_v^{\text{sent}}(i-1) \neq Agg_v(i-1, t)$ then    /* send changes */
        $Agg_v^{\text{sent}}(i-1) \leftarrow Agg_v(i-1, t)$
        forward $\langle \mathsf{change}, i, v, Agg_v(i-1, t), p \rangle$ towards $p$ in $T_i(p)$
    if $v = p$ then
        wait until timer $< 4\theta^i$    /* wait for all changes to arrive */
        if $\exists p', p'' \in Sub_{i-1}(v)$ s.t. $F_O(Agg_v^{\text{recv}}(i, p')) \neq F_O(Agg_v^{\text{recv}}(i, p''))$ then
            $Conf_v(i, t) \leftarrow \mathsf{true}$
        $Agg_v(i, t) \leftarrow F_{agg}(\ \{Agg_v^{\text{recv}}(i, p') \mid p' \in Sub_{i-1}(v)\}\ )$
        if $i = VP_v^v(t)$ then    /* reached prev. VL: update output and VP */
            if $O_v^v(t) \neq F_O(Agg_v(i, t))$ then
                multicast $\langle \mathsf{output}, i, v, F_O(Agg_v(i, t)), t \rangle$ to $\widehat{S}_i(v)$
            if $i > 0 \ \wedge\ Conf_v(i, t) = \mathsf{false}$ then multicast $\langle \mathsf{update\text{-}vp}, i, v, 0, t \rangle$ to $T_i(v)$
wait until timer expires

Message handlers:

upon receiving the first $\langle$conflict,i,p,t$\rangle$ message:
    if $v = p$ then
        $Agg_v(i, t) \leftarrow$ converge-cast($i, t$)    /* see Algorithm 3. */
        $Conf_v(i, t) \leftarrow \mathsf{true}$
        multicast $\langle \mathsf{output}, i, v, F_O(Agg_v(i, t)), t \rangle$ to $\widehat{S}_i(p)$
        multicast $\langle \mathsf{update\text{-}vp}, i, v, i, t \rangle$ to $T_i(v)$
    else forward message to $v$'s parent in $T_i(p)$

upon receiving a $\langle \mathsf{change}, i, p', \widehat{R}, p \rangle$ message:
    if $v = p$ then $Agg_v^{\text{recv}}(i, p') \leftarrow \widehat{R}$
    else forward message to $v$'s parent in $T_i(p)$

upon receiving a $\langle \mathsf{output}, i, p, val, t \rangle$ message:
    wait until timer expires
    if $v \in S_i(p)$ then $O_v^v(t) \leftarrow val$
    $\forall u \in \widehat{\Gamma}(v)$: if $u \in S_i(p)$ then $O_v^u(t) \leftarrow val$

upon receiving a $\langle \mathsf{update\text{-}vp}, i, p, l, t \rangle$ message:
    if $i = 0$ then
        if $v \in S_i(p)$ then
            $\forall u \in \widehat{\Gamma}(v)$ s.t. $u \notin S_i(p) \ \wedge\ (u, v) \in Tree^+$: send $\langle \mathsf{update\text{-}vp}, 0, p, l, t \rangle$ to $u$
        wait until timer expires,$\forall u \in \Gamma(v)$ s.t. $u \in S_i(p)$: $VP_v^{\text{new},u}(t) \leftarrow l$
    else if $v \in \mathcal{P}_{i-1}$ then
        if $l = 0 \ \wedge\ Conf_v(i-1, t) = \mathsf{true}$ then $l \leftarrow (i-1)$
        multicast $\langle \mathsf{update\text{-}vp}, i-1, v, l, t \rangle$ to $T_{i-1}(v)$
```

Algorithm 3. (converge-cast RPC) for node $v \in V$

```
Function converge-cast($i, t$) → $\widehat{R}$
  if $i > VP_v^v(t) \ \wedge \ Conf_v(i, t) = $ false then
    for all $p' \in Sub_{i-1}(v)$ parallel do
      $tmp(p') \leftarrow p'$.converge-cast($i-1, t$)    /* $p'$ is reached via $T_i(v)$ */
    $Agg_v(i, t) \leftarrow F_{agg}(\ \{tmp(p') \mid p' \in Sub_{i-1}(v)\}\ )$
  return $Agg_v(i, t)$
```

MultI-LEAG operates in iterations (the outer loop). An iteration begins by sampling the input and ends with all nodes holding the correct aggregate result matching the sampled inputs. Within an iteration, MultI-LEAG executes Λ_θ synchronous phases that correspond to the levels of the partition hierarchy, calling do-phase each time. (A timer ensures that the next phase is not started before all nodes complete the current phase.) It is convenient to think of do-phase as a sequential operation that takes place concurrently in every cluster S of the current level. Informally, for every phase i and cluster $S \in \mathcal{S}_i$, do-phase operates in one of two modes. The first is to react according to S's conflict state: if S is in conflict, explicitly compute its aggregate result and assign it to the output of all nodes in S. (If not in conflict, do nothing.) The second is to merely propagate input *changes* in S, if any exist, to S's pivot.

The decision regarding which mode to use, from a node v's perspective, is as follows. Let j be v's VL in the *previous* input, I^{old}. Until phase j for the *current* input, I^{new}, is reached, we just propagate changes if there are any, and otherwise do nothing. At phase j, we additionally verify that all nodes in $S_j(v)$ hold the correct output according to I^{new}; if they do not, we multicast the correct output to them. Subsequently, we reactively handle conflicts as they occur. Note that for every phase i higher than v's current VL, $S_i(v)$ does not incur conflicts. Thus, Multi-LEAG achieves $O(max\{\,VR(I^{\text{old}}),\ VR(I^{\text{new}})\})$ output stabilization and quiescence times (Theorem 2). In any case, no messages are sent when there are no input changes.

Had we chosen to operate in conflict detection mode at all times, the resulting protocol would closely resemble I-LEAG [1], and would send messages for every non-trivial input (because at least one cluster would suffer a conflict) regardless of whether any inputs change, which is unacceptable.

We now describe MultI-LEAG's operation in more detail. For every node v, VP_v equals v's VL according to the *previous* input, and remains unchanged until the end of the iteration. VP_v^{new} is gradually updated to reflect the current VL, and is only used to set VP_v in the next iteration. Therefore, for facility of exposition, we currently ignore the $Conf_v$ mapping and the update-vp message handler, which are responsible for updating VP_v^{new}. For every phase i, $p \in \mathcal{P}_i$, and $S_i(p) \in \mathcal{S}_i$, we distinguish among the following cases:

$\forall v \in S_i(p)$: $i < VP_v$ **(change propagation).** Every $p' \in Sub_{i-1}$ sends changes in $Agg_{p'}(i-1)$ to p (lines 7-9). Every such update is saved in $Agg_p^{\text{recv}}(i, p')$ by the change message handler. After all updates are accepted (this is ensured by the **wait** statement in line 11), $Agg_p(i)$ is recalculated (line 14).

$\forall v \in S_i(p)$: $i = VP_v$ **(change propagation and output validation).** First, we update $Agg_p(i)$ as described above. Next, we ensure that the output of every $v \in S_i(p)$ equals $F(I_{S\ (p)})$. As previous phases (which follow the first case) have not altered $S_i(p)$'s outputs at all, every $v \in S_i(p)$ holds the same output, which equals the aggregate result according to the previous input. Therefore, it is sufficient to check only p's output. If $O_p \neq F_O(Agg_p(i))$ (line 16), then $S_i(p)$'s correct aggregate result is multicast to $\widehat{S}_i(v)$ and assigned by the output handler. Specifically, every $v \in S_i(p)$ updates O_v, and every neighbor u of v such that $u \in S_i(p)$ or $(u, v) \in Tree^+$ updates O_u^v. ($Tree^+$ denotes the union of all tree edges; see definition in Algorithm 1..) Otherwise, the outputs of all nodes in $S_i(p)$ remain unaltered.

$\forall v \in S_i(p)$: $i > VP_v$ **(conflict detection).** Assuming that all nodes within a level-$(i-1)$ cluster have the same output (see previous case), conflicts are detected without communication by comparing outputs of neighboring nodes along $T_i(p)$, which know each other's output. Detected conflicts are reported to p and handled by the conflict handler. In this case, p issues a converge-cast call (see Algorithm 3. and explanation below) to explicitly update $Agg_p(i)$. Finally, $S_i(p)$'s aggregate result, $F_O(Agg_p(i))$, is multicast to $\widehat{S}_i(v)$ as in the previous case.

Note that according to VL's definition, no other cases are possible.

To show how VP_v^{new} is gradually adjusted to reflect the current input, we begin by describing $Conf_v$, which records cluster conflict states. At the beginning of an iteration, $Conf_v$ maps trivially to false for every phase other than 0 in all nodes. In every phase i and pivot $p \in \mathcal{P}_i$, $Conf_p(i)$ is assigned true if $S_i(p)$ is in conflict. This is done either by examining updated aggregate results if $i \leq VP_p$ (line 12), or by receiving a conflict message if $i > VP_p$.

When a new iteration begins, VP_v^{new} is equal to VP_v. Subsequently, it is updated by update-vp messages, which are initiated by pivot nodes and flooded along their logical trees. Specifically, at phase i, a pivot $p \in \mathcal{P}_i$ changes VP_v^{new} for every node $v \in S_i(p)$ in two cases. If $i > VP_p$ and $S_i(p)$ is in conflict (i.e., a conflict message is received by p at level i), p increases VP_v^{new} to i. Alternatively, if $i > 0$, $i = VP_p$ and $S_i(p)$ is *not* in conflict (line 18), p decreases VP_v^{new} to the highest level for which v's cluster wan in conflict so far. This is done by sending the first update-vp messages with a VP value (the last parameter) of 0. When a descendent pivot p' of p at some level $j < i$ receives a 0 VP value and its cluster is in conflict, it replaces this value with j for the rest of the subtree.

Thus, for any node v, VP_v^{new} can be lowered at most once when phase VP_v is reached (by v's pivot in level VP_v), and possibly increased one or more times in subsequent phases. At the end of the iteration, VP_v^{new} equals the input's VL, and is assigned to VP_v.

The converge-cast procedure is described in Algorithm 3. using remote procedure call (RPC) semantics. At every phase j and pivot $p \in \mathcal{P}_j$, invoking

$p.\mathsf{converge\text{-}cast}(j)$ aggregates the inputs of $S_j(p)$ recursively based on p's logical tree, $\widetilde{T}_j(p)$. Note that for every level $i < j$ and pivot $p' \in \mathcal{P}_i$, if $i \leq VP_{p'}$ then $Agg_{p'}(i)$ is already up to date because all input changes in $S_j(p')$ have already been accounted for during phase i. In addition, if $i > VP_{p'}$ but $Conf_{p'}(i) = \mathsf{true}$, then $Agg_{p'}(i)$ was updated by a prior converge-cast operation during conflict handling in phase i. Thus, $Agg_{p'}(i)$ needs to be recalculated only if $i \leq VP_{p'}$ and $Conf_v(i) = \mathsf{false}$.

MultI-LEAG's correctness and complexity are proved in [20]. Specifically, we show that MultI-LEAG achieves the multi-shot lower bound (Theorem 1) up to a constant factor:

Theorem 2. *Let $P_{G,F}$ be an aggregation problem. Given a (θ, α)-LPH of G, for every two consecutive iterations with non-trivial input assignments, I^{old} and I^{new}, MultI-LEAG's output stabilization and quiescence times for I^{new} are at most: $\left(\frac{5\theta^2}{\theta-1}\right)r$, where $r = max\{\,VR_\alpha(I^{\text{old}}),\, VR_\alpha(I^{\text{new}})\}$.*

5 DynI-LEAG: An Efficient Dynamic Aggregation Algorithm

While MultI-LEAG is efficient in terms of communication and converges rapidly after sampling the inputs, its sampling interval is proportional to the graph diameter. Therefore, it is not suitable for applications in which fast output stabilization is desirable at all times. In this section, we present DynI-LEAG, an efficient aggregation algorithm with fast output stabilization.

DynI-LEAG achieves this by concurrently invoking multiple MultI-LEAG iterations, one per sample, and pipelining their phases. This is challenging, however, because phases have exponentially increasing durations. DynI-LEAG's samples occur frequently, at intervals reflecting the operation time of the first phase. Thus, invoking a full iteration upon each sample would create a number of concurrent iterations that is linear in the graph's diameter, which would lead to considerable resource (messages and memory) consumption. We overcome this challenge by invoking *partial* MultI-LEAG iterations, i.e., iterations that do not execute all phases, to ensure that at every level of the LPH only a single corresponding MultI-LEAG phase is executed at any given moment. This results in a "ruler-like" schedule that executes only $O(\log(Diam(G)))$ concurrent iterations, which we call *Ruler Pipelining.* Figure 1 illustrates ruler pipelining for an LPH with $\theta = 2$. As a consequence, DynI-LEAG requires only $O(\log^2(Diam(G)))$ memory per node (each MultI-LEAG iteration has practically the same memory utilization as I-LEAG, which requires $O(\log(Diam(G)))$ memory for reasonable LPHs [1]), while the interval between two consecutive MultI-LEAG phases at the same level is only θ times that of an algorithm that requires $\Omega(Diam(G))$ memory.

A MultI-LEAG iteration ensures that its calculated output and VP values are correct only after it completes. Since this takes $O(Diam(G))$ time, yet another challenge is to select the proper output and VP (for new iterations) from among

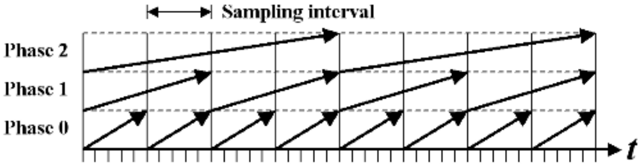


Fig. 1. Ruler Pipelining for a 3-level LPH with $\theta = 2$

multiple ongoing iterations, while achieving output-stabilization and quiescence times proportional to the lower bound rather than the diameter.

DynI-LEAG is depicted in Algorithm 4., and uses the do-phase and convergecast procedures (code in gray applies). To execute concurrent MultI-LEAG iterations, DynI-LEAG holds for every MultI-LEAG variable, except Agg_v^{sent} and Agg_v^{recv}, a mapping that associates each value the variable holds with a time stamp. This is also done for MultI-LEAG's output register, O_v, which is renamed to O_v^v to distinguish between the outputs of different iterations and the actual DynI-LEAG output. Note that the VP_v and VP_v^{new} variables are expanded to include a qualifier $u \in \widehat{\Gamma}(v)$, which enables nodes to hold the corresponding values of their neighbors. ($u = v$ designates v's values.) In addition, DynI-LEAG introduces one new variable, $t_v(i)$, which designates the starting time of the last level-i phase. Agg_v^{sent} and Agg_v^{recv} are not associated with time stamps since they can be perfectly pipelined, i.e., for every level i, $Agg_v^{\mathrm{sent}}(i-1)$ and $Agg_v^{\mathrm{recv}}(i)$ are only accessed by phase i. This enables DynI-LEAG to use partial iterations at no extra cost: each input change is communicated at most once to higher levels.

DynI-LEAG runs Λ_θ threads at each node, corresponding to the LPH levels, each of which repeatedly calls the *do-phase* procedure (line 14) for the matching level. An individual MultI-LEAG iteration is identified by its starting time, which is also passed during *do-phase* invocations. For every level-i phase, $t_v(i)$ equals the current time when it starts and is incremented by the phase duration, $5\theta^i$, when it completes (line 15). The starting time of the corresponding iteration is found by subtracting from $t_v(i)$ the duration of previous phases, $\Delta(i-1)$. Ruler pipelining is obtained as a direct outcome of this timing: the results of each completed level-i phase are either used in the level-$(i+1)$ phase that starts at the same time or ignored in the case of a partial iteration that ends at phase i. The barrier in line 16 eliminates data races between phases.

The crux of the algorithm is concentrated at the beginning of a new iteration (i.e., it is executed only by the thread handling phase 0), and comprises four operations: (1) sampling the input; (2) choosing the VP and initial output values for the new iteration; (3) estimating the output; and (4) performing some bookkeeping. The second operation is done both for a node itself and its neighbor information to ensure that neighboring nodes know each other's output upon starting the iteration.

Algorithm 4. (DynI-LEAG) for node $v \in V$

Parameters: $F{:}\mathbb{N}^D \to R$, (θ, α)-local hierarchy $\langle \{\mathcal{S}_i\}, \{\mathcal{P}_i\}, \{\mathcal{T}_i\} \rangle, 0 \leq i \leq \Lambda_\theta$ of $G(V, E)$

Input: $I_v \in D$

Output: $O_v \in R$ initially $\emptyset$

Definitions: $\mathcal{P}_{i-1} \triangleq V$, $\mathit{Phases} \triangleq \{-1, 0, ..., \Lambda_\theta\}$,
$\mathit{Tree}^+ \triangleq \bigcup_{i, p \in \mathcal{P}} T_i(p)$ ignoring edge directions (i.e., $\mathit{Tree}^+ \subseteq E$),
$\widehat{S}_i(p) \triangleq S_i(p) \cup \{v \in \widehat{\Gamma}(S_i(p)) \mid \exists u \in S_i(p){:}\ (u, v) \in \mathit{Tree}^+\}$,
$\Delta(i) \triangleq \sum_{j=0}^{i} 5\theta^j$,
$\mathit{LastIter}(i, t) \triangleq t - (t \bmod 5\theta^i) - \Delta(i)$

Variables:
$t_v(i){:}\mathit{Phases} \to \mathbb{Z}$ initially 0,
$\forall u \in \Gamma(v){:}\ O_v^u(t){:}\mathbb{Z} \to R \cup \{\bot\}$ initially $\bot$,
$\mathit{Conf}_v(i, t){:}\mathit{Phases} \times \mathbb{Z} \to \{\mathsf{true}, \mathsf{false}\}$ initially true for $i = 0$ and false otherwise,
$\forall u \in \Gamma(v){:}\ \mathit{VP}_v^u(t), \mathit{VP}_v^{\mathrm{new},u}(t){:}\mathbb{Z} \to \mathit{Phases}$ initially 0,
$\mathit{Agg}_v(i, t){:}\mathit{Phases} \times \mathbb{Z} \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
$\mathit{Agg}_v^{\mathrm{sent}}(i){:}\mathit{Phases} \to \widehat{R} \cup \{\bot\}$ initially $\bot$,
$\mathit{Agg}_v^{\mathrm{recv}}(i, p){:}\mathit{Phases} \times V \to \widehat{R} \cup \{\bot\}$ initially $\bot$

```
for all $i \in [0, \Lambda_\theta]$ parallel do
  loop    /* forever */
    if $i = 0$ then
      $\mathit{Agg}_v(-1, t_v(i)) \leftarrow F_I(I_v)$    /* read input */
      for all $u \in \Gamma_v$ do
        $\mathit{Candidates} \leftarrow \{k \in [0, \Lambda_\theta] \mid \mathit{VP}_v^u(\mathit{LastIter}(k, t_v(0))) \leq k \ \wedge$
            $\mathit{VP}_v^{\mathrm{new},u}(\mathit{LastIter}(k, t_v(0))) = k\}$
        $\mathit{VP}_v^u(t_v(0)), \mathit{VP}_v^{\mathrm{new},u}(t_v(0)) \leftarrow \mathit{max}(\mathit{Candidates} \cup \{0\})$
        $O_v^u(t_v(0)) \leftarrow O_v^u(t')$ where $t' = \mathit{LastIter}(\mathit{VP}_v^u(t_v(0)), t_v(0))$
      do-bookkeeping$(t_v(0))$
      $O_v \leftarrow O_v^v(t_v(0))$    /* adjust output */
    if $t_v(i) \geq \Delta(i-1)$ then
      do-phase$(i, t_v(i) - \Delta(i-1))$
    else
      wait for $5\theta^i$ time steps
    $t_v(i) \leftarrow t_v(i) + 5\theta^i$
    barrier$(t_v(i))$    /* synchronize all threads and message handlers that
                            complete a phase at time $t_v(i)$ */
```

Function do-bookkeeping(t)
$T \leftarrow \{\ t' \mid \exists j \in [0, \Lambda_\theta]$ s.t. $t - (t \bmod 5\theta^j) - t' = \Delta(j)$ or $\Delta(j-1)\ \}$
$\forall j \in \mathit{Phases}, u \in \Gamma(v), t' \notin T{:}O_v^u(t') \leftarrow \bot$, $\mathit{Conf}_v(j, t') \leftarrow \mathsf{false}$,
$\mathit{VP}_v^u(t'), \mathit{VP}_v^{\mathrm{new},u}(t') \leftarrow 0$, $\mathit{Agg}_v(j, t') \leftarrow \bot$

To choose a VP value, we initially prepare a list of candidate levels. Level k is considered a candidate if $S_k(v)$ is known to be in conflict according to the most recent information. More formally, we look at the last iteration that completed phase k, i.e., the iteration that started at $\mathit{LastIter}(k, t_v(0))$, where $t_v(0)$, at this point, is the current time. During an iteration, nodes can learn if their cluster at a certain level is in conflict by the reception (or absence) of update-vp messages during the corresponding phase. Specifically, upon completing phase k, if $VR_v^{\text{new},v} = k$, then $S_k(v)$ is in conflict. However, as update-vp messages are only sent after an iteration completes its VP phase, this information is not available beforehand. Consequently, we only accept k as a candidate if both $VP_v^u(\mathit{LastIter}(k, t_v(0))) \leq k$ and $VP_v^{\text{new},u}(\mathit{LastIter}(k, t_v(0))) = k$ hold. Next, we choose the highest candidate, where 0 is always considered a candidate. Both the initial output value and DynI-LEAG's output estimate, O_v, are simply taken as the current output of the iteration corresponding to the chosen candidate. Thus, after the inputs stabilize, the choices of VP converge to VL and the outputs converge to the global aggregate result, thereby guaranteeing both quiescence and output stabilization (Theorem 3).

Finally, the do-bookkeeping procedure ensures that every mapping that is never referenced again, i.e., the time its iteration has started corresponds to neither the current nor last phase of any level, is reset to its default value. Thus, every node has to maintain state for only $2\Lambda_\theta$ MultI-LEAG iterations.

DynI-LEAG's correctness and complexity are proved in [20]. Specifically, we show that DynI-LEAG achieves the dynamic lower bound (Corollary 1) up to a constant factor:

Theorem 3. *Let $P_{G,F}$ be an aggregation problem. For every slack function α, every $r^{\text{old}}, r^{\text{new}} \geq 0$, and every input sequence $\mathcal{I}$ such that all input changes cease at time t_0 and:*

1. *$\forall r > r^{\text{old}}$: for every $t \geq t_0 - 30\theta \cdot r$, $VR_\alpha(t) < r$*
2. *$VR_\alpha(t_0) = r^{\text{new}}$*

DynI-LEAG reaches both quiescence and output-stabilization by time $40\theta \cdot max\{r^{\text{old}}, r^{\text{new}}\}$.

6 Conclusions

We provided two efficient algorithms, MultI-LEAG and DynI-LEAG, for dynamic aggregation in large graphs with fixed topologies. When the inputs are stable, the algorithms are quiescent and hence do not waste any resources from the communication network. When changes do occur, the performance of these algorithms is proportional to the *Veracity Radius* of the inputs at hand, which enables them to achieve optimal *instance-local* operation and resource utilization.

Consequently, these algorithms are extremely attractive for data aggregation tasks in dynamic, resource-constrained environments in which topological changes are infrequent compared to the sampling rate, be it for periodically obtaining the result according to the most recent sample in a very efficient manner (MultI-LEAG) or for closely tracking the monitored environment to capture global trends as fast as possible (DynI-LEAG).

References

1. Birk, Y., Keidar, I., Liss, L., Schuster, A., Wolff, R.: Veracity radius - capturing the locality of distributed computations. To appear in PODC (2006)
2. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. In: Proc. of the 5th Annual Symposium on Operating Systems Design and Implementation (OSDI). (2002)
3. Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D.: Wireless sensor networks for habitat monitoring. In: Proc. of the ACM Workshop on Sensor Networks and Applications. (2002)
4. van Renesse, R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (2003)
5. The Condor Project, `http://www.cs.wisc.edu/condor/`.
6. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In Proceedings of the Sixth Annual Intl. Conf. on Mobile Computing and Networking (2000)
7. Kempe, D., Dobra, A., Gehrke, J.: Computing aggregate information using gossip. Proceedings of Fundamentals of Computer Science (2003)
8. Bawa, M., Garcia-Molina, H., Gionis, A., Motwani, R.: Estimating aggregates on a peer-to-peer network. Technical report, Stanford University, Database group (2003) Available from: `http://www-db.stanford.edu/~bawa/publications.html`.
9. Considine, J., Li, F., Kollios, G., Byers, J.: Approximate aggregation techniques for sensor databases. In: Proc. of ICDE. (2004)
10. Zhao, J., Govindan, R., Estrin, D.: Computing aggregates for monitoring wireless sensor networks. In: Proc. of SNPA. (2003)
11. Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The price of validity in dynamic networks. In: Proc. of ACM SIGMOD. (2004)
12. Kutten, S., Peleg, D.: Fault-local distributed mending. Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (1995)
13. Kutten, S., Peleg, D.: Tight fault-locality. In: Proc. of the 36th IEEE Symposium on Foundations of Computer Science. (1995)
14. Kutten, S., Patt-Shamir, B.: Time-adaptive self-stabilization. Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (1997) 149–158
15. Azar, Y., Kutten, S., Patt-Shamir, B.: Distributed error confinement. In: Proc. of the 22nd Annual Symp. on Principles of Distributed Computing. (2003)
16. Li, J., Jannotti, J., Couto, D.D., Karger, D., Morris, R.: A scalable location service for geographic ad hoc routing. In: Proc. of the 6th ACM Intl. Conf. on Mobile Computing and Networking. (2000)
17. Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA). (2004) 359–368
18. Liss, L., Birk, Y., Wolff, R., Schuster, A.: A local algorithm for ad hoc majority voting via charge fusion. In: Proceedings of the Annual Conference on Distributed Computing (DISC). (2004)
19. Wolff, R., Schuster, A.: Association rule mining in peer-to-peer systems. In: Proc. of the IEEE Conference on Data Mining (ICDM). (2003)
20. Birk, Y., Keidar, I., Liss, L., Schuster, A.: Efficient dynamic aggregation. CCIT Technical Report 589, EE Department, Technion (2006)

Groupings and Pairings in Anonymous Networks

Jérémie Chalopin[1], Shantanu Das[2], and Nicola Santoro[3]

[1] LaBRI Université Bordeaux 1, Talence, France
chalopin@labri.fr
[2] School of Information Technology and Engineering, University of Ottawa, Canada
shantdas@site.uottawa.ca
[3] School of Computer Science, Carleton University, Canada
santoro@scs.carleton.ca

Abstract. We consider a network of processors in the absence of unique identities, and study the *k-Grouping* problem of partitioning the processors into groups of size k and assigning a distinct identity to each group. The case $k = 1$ corresponds to the well known problems of *leader election* and *enumeration* for which the conditions for solvability are already known. The grouping problem for $k \geq 2$ requires to break the symmetry between the processors *partially*, as opposed to problems like leader election or enumeration where the symmetry must be broken completely (i.e. a node has to be distinguishable from all other nodes). We determine what properties are necessary for solving these problems, characterize the classes of networks where it is possible to solve these problems, and provide a solution protocol for solving them.

For the case $k = 2$ we also consider a stronger version of the problem, called *Pairing* where each processor must also determine which other processor is in its group. Our results show that the solvable class of networks in this case varies greatly, depending on the type of prior knowledge about the network that is available to the processors. In each case, we characterize the classes of networks where *Pairing* is solvable and determine the necessary and sufficient conditions for solving the problem.

1 Introduction

Consider a distributed system consisting of a network of n processors and suppose we want to partition the n nodes of the network into uniquely identified groups, each consisting of k nodes, where k divides n. This problem, called *k-Grouping*, is of simple resolution if the nodes have unique identifiers. However, in absence of distinct nodes identities (i.e., in an *anonymous* network), the solution of the *k-Grouping* problem becomes difficult, if at all possible. The goal of this paper is to understand under what conditions this problem is solvable in such a setting.

Notice that when $k = 1$, the grouping problem is equivalent to the well known *Node-Labelling* and *Enumeration* problems, where each node has to be assigned a distinct label (ranging from 1 to n, in case of Enumeration). The 1-*Grouping* problem is also computationally equivalent to the *Leader Election*

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 105–119, 2006.

problem, where one of the nodes has to become distinguished from all others. Although a natural extension to these problems, the k-Grouping problem for $k > 1$ has never been studied before, to the best of our knowledge.

For the leader election problem, it is known that the solvability of the problem depends on the (presence or absence) of symmetry between the nodes in the network. However, even if election is not solvable in a given network, it may be still possible to solve the grouping problem in that same network. In fact, the k-grouping problem for $k \geq 2$ requires to break the symmetry between the nodes only *partially*, as opposed to problems like leader election or enumeration where the symmetry must be broken completely (i.e. a node has to be distinguishable from all other nodes). Hence our investigation focuses on the computational aspects of *partial symmetry-breaking*; more precisely, our interest is in determining what conditions are necessary for solving these problems and in characterizing the solvable instances. A case of particular interest is when $k = 2$, called the *Matching* problem in which the nodes of the network are to be grouped in pairs[1].

It is interesting to note that the solvability of these problems depends not only on the symmetry of the network but also on what information is initially available to the nodes of the network, for instance, whether they know the topology or the size of the network or whether they have a map of the network.

We are also interested in a stronger version of the grouping problem, which we call *k-Relating*, where each node must also determine which other nodes have been grouped with it. Specifically, each node should be able to compute a path between itself and any other processor in its group. In the case $k = 2$, this problem is called *Pairing.* and each node must know a path to the other node it is paired with.

Related Results: The study of computability in an anonymous network of processors, has been a subject of intense research starting from the pioneering work of Angluin [1] who studied the problem of establishing a “center” in the network. This work was extended by Johnson and Schneider [10] and later by Yamashita and Kameda who gave a complete characterization of graphs where the leader election problem is solvable [16] and of graphs where any arbitrary function can be computed [17]. Boldi *et al.* [2] characterized labelled networks based on the election problem, whereas Boldi and Vigna [3] have studied the problem of general computability in directed graphs using the concepts of graph fibrations [4] and coverings, (which we also use in the present paper). Others have studied the computability issues in specific topologies or restricted to special functions (see [11] for a survey of such results). Sakamoto [15] studied the effects of initial conditions of the processors on computability in anonymous networks, while Flocchini *et al.* [8] investigated the impact of *sense of direction* on computability in anonymous networks.

[1] This problem is un-related to the distributed client-server match-making problem studied in the literature [14], where nodes are already divided into clients and servers and the network is not anonymous.

Mazurkiewicz [13] gave an algorithm (in the *local-computation* model) for the distributed enumeration problem, i.e. for numbering the nodes of an undirected graph G with integers from 1 to $|V(G)|$. They showed that it is possible to do this only when the graph G is "unambiguous". Godard et al. [9] translated this property in terms of coverings of simple graphs. Chalopin and Métivier [6] later adapted the Mazurkiewicz algorithm to the message passing model and showed that the enumeration problem is solvable in a symmetric directed graph G, if and only if G is *symmetric-covering-prime.*

Our Results: We first consider the *k-Grouping* problem and provide a complete characterization of its solvability. First of all, we show that the knowledge of the *exact* size of the network is necessary for solving the problem. Then we determine the necessary and sufficient condition for solving the *k-Grouping* problem, when such knowledge is available. For the case $k = 1$, this characterization corresponds precisely (as it should) to that given in [3,16] for the leader election problem. We then present an algorithm (Algorithm 1) that solves the *k-Grouping* problem using a simple extension to the Mazurkiewicz algorithm. As part of our solution, we introduce a deterministic procedure with explicit termination, that computes the minimum base of any given network in the message-passing system. Our solution is able to detect if *k-Grouping* is solvable for any given k in any given network and reports failure when the problem is not solvable in that network.

Building on the above results, in section 4.1, we investigate the *Pairing* problem under three different types of prior information that may be available to the processors in the network, and we provide an almost complete characterization of its solvability. The types of prior knowledge we consider are: (i) a complete map of the network[2]; (ii) just the number of nodes; (iii) only an upper bound on the number of nodes. We determine sufficient conditions for solving the *Pairing* problem under all three different types of knowledge.

Finally, in section 4.2, we determine necessary conditions for solving the *Pairing* problem in each of the different cases. We show that when a complete map is available or, when only an upper-bound on n is known, the sufficient conditions we have established for these two cases are necessary too; that is, our characterization is complete in case (i) and (iii). In case (ii), when the nodes have prior knowledge of the exact size of the network, there is a still gap between the necessary and sufficient conditions.

2 The Model and the Definitions

2.1 Directed Graphs

A directed graph(digraph) $D = (V(D), A(D), s_D, t_D)$ possibly having muliple arcs and self-loops, is defined by a set $V(D)$ of vertices, a set $A(D)$ of arcs and by two maps s_D and t_D that assign to each arc two elements of $V(D)$: a source and a target (in general, the subscripts will be omitted). A digraph D is strongly connected if for all vertices $u, v \in V(D)$, there exists

[2] The map is unanchored i.e. a node may not know its location in the map.

a path between u and v. A *symmetric* digraph D is a digraph endowed with a symmetry, that is, an involution $Sym : A(D) \to A(D)$ such that for every $a \in A(D), s(a) = t(Sym(a))$. In a symmetric digraph, the mirror of a path $P = (a_0, a_1, \dots, a_p)$ is the path $(Sym(a_p), Sym(a_{p-1}), \dots, Sym(a_0))$. In this paper, we will only consider strongly connected symmetric digraphs.

A digraph homomorphism γ between the digraph D and the digraph D' is a mapping $\gamma\colon V(D) \cup A(D) \to V(D') \cup A(D')$ such that if u, v are vertices of D and a is an arc such that $u = s(a)$ and $v = t(a)$ then $\gamma(u) = s(\gamma(a))$ and $\gamma(v) = t(\gamma(a))$. We consider digraphs where the vertices and the arcs are labelled with labels from a recursive label set L and such digraphs will be denoted by (D, λ), where $\lambda\colon V(D) \cup A(D) \to L$ is the labelling function. A homomorphism from (D, λ) to (D', λ') is a digraph homomorphism from D to D' which preserves the labelling, i.e., such that $\lambda'(\gamma(x)) = \lambda(x)$ for every $x \in V(D) \cup A(D)$.

2.2 The Message-Passing Network Model

We represent a point-to-point message passing network by a connected symmetric digraph $\mathbf{G}$ without self-loops and multiple arcs. The vertices represent processors and if there is a (bidirectional) communication link between processors corresponding to some vertices u and v, there is an arc a_{uv} from u to v, an arc a_{vu} from v to u and $Sym(a_{uv}) = a_{vu}$. The initial state of the processors is encoded by a vertex labelling function $\lambda^V : V(\mathbf{G}) \to \Sigma$, where Σ is a set with a total order $<_\Sigma$. In particular, if all vertices have the same label i.e. $\lambda^V(v) = \lambda^V(v')$, $\forall v, v' \in V(G)$, then the network is anonymous.

We assume the presence of a local orientation λ^A on the network: for each vertex u (of degree d), there exists an injective mapping λ_u^A that associates a unique number $\lambda_u^A(v) \in [1, d]$ to each neighbor v of u. This local orientation defines a labelling on the arcs of $\mathbf{G}$ as follows. For any pair of neighboring nodes $\{u, v\}$ in $\mathbf{G}$, $\lambda^A(a_{uv}) = (\lambda_u^A(v), \lambda_v^A(u))$ and $\lambda^A(a_{vu}) = (\lambda_v^A(u), \lambda_u^A(v))$. From this construction, one can notice that for any arc $a \in (\mathbf{G}, \lambda)$, if $\lambda^A(a) = (p, q)$, then $\lambda^A(Sym(a)) = (q, p)$.

The labelled digraph $(\mathbf{G}, \lambda)$ would be called a *network*, if and only if it satisfies each of the following: (i) There does not exist any arc $a \in A(\mathbf{G})$ such that $s(a) = t(a)$ (i.e. no self loops), (ii) There does not exist two distinct arcs $a, a' \in A(\mathbf{G})$ such that $s(a) = s(a')$ and $t(a) = t(a')$ (i.e. no parallel arcs), and (iii) $\lambda = (\lambda^V, \lambda^A)$, where $\lambda^V : V(\mathbf{G}) \to \Sigma$ and λ^A is a local orientation on $\mathbf{G}$, as defined above.

The vertices of the network $(\mathbf{G}, \lambda)$ would be called nodes or, processors. Each processor v in the network represents an entity that is capable of performing computation steps, sending messages on any outgoing arcs, and receiving any message that was sent on any of the incoming arcs. Notice that the entity can distinguish among the arcs due to the presence of local orientation. The following procedure calls are available to the entity at a node v : *Send*$< M, p >$ and *Receive*$< M, p >$, to send (respectively receive) the message M on the communication link labelled by p. Every entity executes the same algorithm provided to it which consists of a sequence of computation steps interspersed with procedure calls of the two types mentioned above. Each of the steps of execution may take an unpredictable (but finite) amount of time (i.e. we consider fully asynchronous systems).

For any path $P = (a_1, a_2, \ldots, a_j)$ in the network $(\mathbf{G}, \lambda)$, the sequence of arcs labels corresponding to it is denoted by $\Lambda(P) = (\lambda^A(a_1), \lambda^A(a_2), \ldots, \lambda^A(a_j))$. For any sequence of edge-labels α, we define the function T_α for a network $(\mathbf{G}, \lambda)$ as follows. A node $u = T_\alpha(v)$ if and only if there is path P from v to u in G whose label-sequence $\Lambda(P)$ is α. Notice that if λ is a local orientation then there can be at most one node of this kind and then $T_\alpha(v)$ is a mapping.

Each processor, at the beginning of computation would have the same knowledge about the network. As in [16] we will focus on three different kinds of initial knowledge that may be available to the processors:

[UB] Knowledge of an upper bound on n, the size of G,
[ES] Knowledge of the exact value of n, the size of G
[MP] Knowledge of a map (i.e. an isomorphic copy) of the labelled graph $(\mathbf{G}, \lambda)$.

2.3 Problems and Definitions

Informally speaking, the problem of *k-GROUPING* is to partition the nodes of the network into groups of k nodes, where nodes in the same group are identified by a common label assigned to them.

k-GROUPING: Given the network represented by $(\mathbf{G}, \lambda)$, compute at each node v the value $LABEL(v)$ where $LABEL : V(G) \to \mathbb{N}$ satisfies the condition that for each $v \in V(G)$, $|\{u \in V(G) : LABEL(u) = LABEL(v)\}| = k$.

In the particular case, where $k = 1$, this problem corresponds to the well-studied problems of naming/enumeration and election. For the case $k = 2$, we call it the *MATCHING* problem where the nodes of the network are matched-up in pairs such that nodes in a pair share the same label. Notice that the nodes matched to each-other may not be adjacent and in general, a node may not know which other node it has been matched with. A more difficult version of MATCHING (or, 2-GROUPING) is the *PAIRING* problem which involves forming pairs among the nodes of the graph, such that each node v knows a path leading to the other node it is paired with, denoted by $Pair(v)$. This is defined formally as:

PAIRING: Given a network represented by $(\mathbf{G}, \lambda)$, compute at each node v the sequence of edge-labels representing a path from node v to the node $pair(v)$, where the function $pair : V(G) \to V(G)$ is such that (i) $pair(v) = u \Leftrightarrow pair(u) = v$, (ii) $pair(u) = pair(v) \Leftrightarrow u = v$, and (iii) $pair(v) \neq v$ for any $v \in V(G)$.

The generalized version of the *Pairing* problem, called *k-relating*, $k \geq 1$, is not considered in the present paper.

Definition 1. *For each of the above problems, we say that the problem is* solvable *on a given instance* $(\mathbf{G}, \lambda)$*, under the knowledge* MP*(respectively* ES *or,* UB*) if there exists a deterministic (distributed) algorithm* A *such that every execution of* *algorithm A on $(\mathbf{G}, \lambda)$, succeeds in solving the problem (i.e. produces the required output), when provided with the appropriate input according to* MP*(respectively* ES *or,* UB*).*

We are interested in generic solution protocols for the problems, i.e. algorithms which, when executed on any given instance $(\mathbf{G}, \lambda)$, always terminates within a finite time, either successfully solving the problem, or reporting failure to do so.

Definition 2. *We say that an algorithm A is* effective *for a given problem, under the knowledge* MP*(respectively* ES *or,* UB*) if for every instance $(\mathbf{G}, \lambda)$ of the problem, the algorithm A succeeds in solving the problem if and only if the problem is solvable on $(\mathbf{G}, \lambda)$ under the knowledge* MP*(respectively* ES *or,* UB*)*

2.4 Fibrations and Coverings

The notions of fibrations and coverings were defined by Boldi and Vigna in [4]. We present the main definitions and properties here, that are going to be used in this work.

A *fibration* between the digraphs D and D' is a homomorphism φ from D to D' such that for each arc a' of $A(D')$ and for each vertex v of $V(D)$ such that $\varphi(v) = v' = t(a')$ there exists a unique arc a in $A(D)$ such that $t(a) = v$ and $\varphi(a) = a'$. The fibre over a vertex v (resp. an arc a) of D' is the set $\varphi^{-1}(v)$ of vertices of D (resp. the set $\varphi^{-1}(a)$ of arcs of D).

An *opfibration* between the digraphs D and D' is a homomorphism φ from D to D' such that for each arc a' of $A(D')$ and for each vertex v of $V(D)$ such that $\varphi(v) = v' = s(a')$ there exists a unique arc a in $A(D)$ such that $s(a) = v$ and $\varphi(a) = a'$.

A *covering projection* is a fibration that is also an opfibration. If a covering projection $\varphi : D \to D'$ exists, D is said to be a *covering* of D' via φ and D' is called the base of φ. A symmetric digraph D is a *symmetric covering* of a symmetric digraph D' via a homomorphism φ if D is a covering of D' via φ such that $\forall a \in A(D), \varphi(Sym(a)) = Sym(\varphi(a))$. A digraph D is *symmetric-covering-minimal* if there does not exist any graph D' not isomorphic to D such that D is a symmetric covering of D'.

Property 1 ([4]). Given two non-empty strongly connected digraphs D, D', each covering projection φ from D to D' is surjective; moreover, all the fibres have the same cardinality. This cardinality is called the number of sheets of the covering.

The notions of fibrations and of coverings extend to labelled digraphs in an obvious way: the homomorphisms must preserve the labelling. Given a labelled symmetric digraph (G, λ), the minimum base of (G, λ) is defined to be the labelled digraph (H, λ_H) such that (i) (G, λ) is a symmetric covering of (H, λ_H) and (ii) (H, λ_H) is symmetric covering minimal.

The above definition is equivalent to that given in [12,4] where the minimum base is defined using the degree refinement technique that is related to techniques used for minimizing deterministic automata.

Given a labelled digraph (G, λ_G) and its minimum base (H, λ_H), the quantity $q = |V(H)|/|V(G)|$ is called the *symmetricity* (see [16]) of the labelled digraph (G, λ_G). This quantity is same as the number of sheets of the covering projection φ from (G, λ_G) to (H, λ_H).

The following property says that if (G, λ_G) is a covering of (H, λ_H), then from any execution of an algorithm on (H, λ_H), one can build an execution of the algorithm on (G, λ_G). This is the counterpart of the lifting lemma that Angluin gives for coverings of simple graphs [1] and the proof can be found in [4,6].

Property 2. If (G, λ_G) is a covering of (H, λ_H) via φ, then any execution of an algorithm $\mathcal{A}$ on (H, λ_H) can be lifted up to an execution on (G, λ_G), such that at the end of the execution, for any $v \in V(G)$, v would be in the same state as $\varphi(v)$.

In particular, if we consider a synchronous execution of an algorithm $\mathcal{A}$ on (G, λ), then this execution is obtained by lifting up the synchronous execution of $\mathcal{A}$ on the minimum base (H, λ). As a result of the above property we have the following additional property, which is useful for proving impossibility results.

Property 3. Consider two labelled digraphs (G_1, λ_1) and (G_2, λ_2) that both cover the same labelled digraph (H, λ_H) via φ_1 and φ_2 respectively. For any algorithm $\mathcal{A}$, there exist executions of $\mathcal{A}$ on (G_1, λ_1) and (G_2, λ_2) such that at the end of these executions, any vertex $v_1 \in V(G_1)$ would be in the same state as any vertex $v_2 \in \varphi_2^{-1}(\varphi_1(v)) \subset V(G_2)$ provided that the vertices are given the same input initially.

3 Solving the k-Grouping Problem

3.1 Conditions for Solvability

Throughout the rest of this paper, we shall assume that the values of k and n are such that k divides n, which is a necessary condition for solving the problems that we consider.

Lemma 1. *For solving the* k-Grouping *problem for a given k in a network* $(\mathbf{G}, \lambda)$*, (i) knowledge of the exact size of the network, is necessary (i.e. the knowledge [*UB*] is not sufficient) and (ii) q must divide k, where q is the symmetricity of* $(\mathbf{G}, \lambda)$.

Proof Omitted.

3.2 Solution Protocol

We give below an algorithm *Grouping(n, k)* for solving the k-Grouping problem in a network of size n. The algorithm computes the minimum base (H, λ_H) of the network $(\mathbf{G}, \lambda)$, using the sub-procedure *Enumerate* which is based on Chalopin and Métivier's version of the Mazurkiewicz enumeration algorithm. However, we modify the algorithm to obtain a pseudo-synchronous algorithm that labels the vertices of the network with integers from 1 to $|V(H)|$, such that all nodes that map to the same vertex v in H share the same label. This enables us to compute the minimum base (H, λ_H) based on the labelling. (Note that computing the minimum-base of a digraph is a fundamental problem which is related to state-minimization of automata and also to graph-partitioning and isomorphism detection [5,7]. However the known solutions are not directly applicable in the present model.)

Algorithm 1. Grouping(n,k)

```
$(H, num) :=$ Enumerate$(n)$ ;
$q := n/|V(H)|$ ;
$x := n/k$ ;
if $q$ divides $k$ and $k$ divides $n$ then
    return (num modulo x) + 1 ;
else
    Terminate with failure ;
```

Procedure `Enumerate(`$\hat{n}$`)` at node v

```
$n(v) := 1$ ;
$N(v) := \emptyset$ ;
$M(v) := \{(1, \lambda(v), \emptyset\}$ ;
for $\hat{n}^4$ iterations do
    for $p$ := 1 to $d_G(v)$ do send $< (n(v), M(v)), p >$ via port $p$ ;
    for $p$ := 1 to $d_G(v)$ do
        receive $< (x, M), q >$ via port $p$ ;
        $N(v) := N(v) \setminus \{(\_, p, \_)\} \cup \{(x, p, q)\}$ ;
        $M(v) := M(v) \cup M \cup \{(n(v), \lambda(v), N(v))\}$ ;
    if $\exists (n(v), l, N) \in M(v) \mid (\lambda(v), N(v)) \prec (l, N)$ then
        $n(v) := 1 + \max\{x \mid \exists (x, l, N) \in M(v)\}$ ;
        $M(v) := M(v) \cup \{(n(v), \lambda(v), N(v))\}$ ;
$Map :=$ Construct-Graph($M(v)$) ;
return $(Map, n(v))$ ;
```

During the procedure *Enumerate*, the state of each processor $v_i \in V(G)$ is represented by $(\lambda^V(v_i),\ c(v_i))$, where $c(v_i) = (n(v_i), N(v_i), M(v_i))$ represents the following information obtained during the computation:

- $n(v_i) \in \mathbb{N}$ is the *number* assigned to v_i by the algorithm.
- $N(v_i)$ is the *local view* of v_i, i.e., the information the vertex v_i has about its neighbours. This contains elements of the form (n_j, p_j, q_j) where n_j is the number assigned to a neighbor v_j and the arc from v_i to v_j is labelled (p_j, q_j).

- $M(v_i)$ is the *mailbox* of v_i containing all of the information received by v_0 at previous computation steps. Formally, it is a set of elements of the form (n_j, l_j, N_j) where n_j, l_j, and N_j are respectively the number, the initial label and the local view of some node at some previous step of the algorithm.

As in the original algorithm of Mazurkiewicz [13], we need a total order on the local views. Given two local views N_1 and N_2, we shall say that $N_1 \prec N_2$ if the maximum element for the lexicographic order of the symmetric difference $N_1 \bigtriangleup N_2 = N_1 \cup N_2 \setminus N_1 \cap N_2$ belongs to N_2. We will also say that $(l_1, N_1) \prec (l_1, N_1)$ if $l_1 <_\Sigma l_2$ or if $l_1 = l_2$ and $N_1 \prec N_2$.

Procedure `Construct-Graph(`M`)`

```
$n_{\max} := \max\{x \mid \exists(x, l, N) \in M\}$ ;
$V(H) := \{v_i \mid 1 \leq i \leq n_{\max}\}$ ;
$A(H) := \emptyset$ ;
for $i := 1$ to $n_{\max}$ do
    $(\lambda_H(v_i), N_i) := \max_\prec\{(l, N) \mid (i, l, N) \in M\}$ ;
    foreach $(j, p, q) \in N_i$ do
        $A(H) := A(H) \cup \{a_{ijpq}\}$ ;
        $s(a_{ijpq}) = v_i$ ;
        $t(a_{ijpq}) = v_j$ ;
        $\lambda_H(a_{ijpq}) = (p, q)$ ;
return $(H, \lambda_H)$ ;
```

Lemma 2. *During the execution of algorithm* Enumerate($\hat{n}$) *on a network* $(\mathbf{G}, \lambda_G)$ *of size* $\leq \hat{n}$*, the map constructed by Procedure* Construct-graph *represents the minimum base* (H, λ_H) *of* $(\mathbf{G}, \lambda_G)$*.*

Once the minimum base of $(\mathbf{G}, \lambda_G)$ has been constructed, it is quite straightforward to solve k-Grouping as shown in Algorithm 1. Notice that the algorithm always succeeds if q divides k which is the necessary condition for solving k-Grouping. Hence we have the following results:

Theorem 1. *Under the knowledge* [ES]*, k-*Grouping *is solvable (for any k that divides n) in the network* $(\mathbf{G}, \lambda)$ *if and only if q divides k, where q is the symmetricity of* $(\mathbf{G}, \lambda)$*.*

Corollary 1. *When the size of the network is known,* Matching *is solvable in* $(\mathbf{G}, \lambda)$ *if and only if the symmetricity of* $(\mathbf{G}, \lambda)$ *is either* 1 *or* 2*.*

4 Solving the Pairing Problem

4.1 Sufficiency Conditions and Solutions

Lemma 3. *If k-Grouping is solvable in* $(\mathbf{G}, \lambda)$ *for* $k = |G|/2$*, then Pairing is also solvable in* $(\mathbf{G}, \lambda)$*.*

Combining the results of Lemma 1 and Lemma 3, we know that Pairing is solvable in $(\mathbf{G}, \lambda)$ if the symmetricity q divides $n/2$. However, since q always divides n, the above condition is equivalent to the condition that 2 divides n/q (i.e. the size of the minimum base). This gives us the following corollary:

Corollary 2. *Pairing is solvable in* $(\mathbf{G}, \lambda)$ *if it is solvable in* (H, λ_H)*, the minimum base of the network (or equivalently, if* H *has even size).*

In the minimum base (H, λ_H), each vertex is uniquely labelled. Thus, Pairing is solvable in (H, λ_H) if and only if H has even number of vertices. From a solution for Pairing in (H, λ_H), we can easily construct a corresponding solution for $(\mathbf{G}, \lambda_G)$. (We only need to ensure that if a node u is paired to v, using the label sequence α, then v should be paired to u using the inverse sequence of α.) In case H has an odd number of vertices, then some node in $\mathbf{G}$ should be paired with another node having the same label (which is possible if there is a symmetric arc joining them).

Theorem 2. *Under the knowledge* [UB]*,* Pairing *is solvable for* $(\mathbf{G}, \lambda_G)$ *having minimum base* (H, λ_H) *and symmetricity* q*, if any one of the following holds:*

(i) (H, λ_H) *has an even number of vertices (i.e.* n/q *is even) or,*
(ii) (H, λ_H) *contains a symmetric self-loop (i.e. an arc* a*, s.t.* $Sym(a) = a$*).*

Let us now consider the case when the exact value of the network size is known.

Theorem 3. *Under the knowledge* [ES]*,* Pairing *is solvable for the network* $(\mathbf{G}, \lambda_G)$ *having minimum base* (H, λ_H) *and symmetricity* q*, if one of the following holds:*

(i) (H, λ_H) *has an even number of vertices (i.e.* n/q *is even),*
(ii) there exists a symmetric self-loop in (H, λ_H) *(i.e., a self-loop whose label has the form* (p, p)*),*
(iii) the minimum base has $2|V(H)|$ *arcs , i.e.,* $|A(H)| = 2n/q$*,*
(iv) $q = 4$ *and there exists a self-loop in* (H, λ_H)*,*
(v) $q = 2$ *and there exists two distinct arcs* $a, a' \in A(H)$ *such that* $s(a) = s(a')$ *and* $t(a) = t(a')$*.*

Proof. Suppose that the size of (H, λ_H) is odd and that it does not contain any symmetric self-loop (otherwise, from Lemma 2, we already know that it is possible to solve Pairing). Since (H, λ_H) does not contain any symmetric self-loop, for each arc $a \in A(H)$, there exists $a' \neq a$ such that $Sym(a) = a'$.

Suppose that (H, λ_H) has $2|V(H)|$ arcs. Then there exists exactly two simple cycles C, C' in (H, λ_H) where C' is the mirror of C. The preimage of a cycle in (H, λ_H) is a set of disjoint cycles in $(\mathbf{G}, \lambda_G)$. If the preimage of C contains at least two cycles, then $(\mathbf{G}, \lambda_G)$ would be disconnected. Consequently, the preimage of C must be a single cycle of length $|C|.q$ in $(\mathbf{G}, \lambda_G)$. Moreover there does not exist any other cycle in $(\mathbf{G}, \lambda_G)$ different from the preimage of C or C'. Since $|V(H)|$ is odd, and $|V(G)| = q|V(H)|$ is even, we know that q is even. Let us fix a vertex $v = s(a_0)$ belonging to the cycle C. Then for each vertex $x \in \varphi^{-1}(v)$, we use the label sequence $\alpha(x) = \Lambda(C)^{q/2}$ to pair it with another vertex $y \in \varphi^{-1}(v)$. Now, the remaining vertices in $\mathbf{G}$ can be easily paired-up.

Suppose there exists a (non-symmetric) self-loop in (H, λ_H) and $q = 4$. Let a be such a self-loop and let $v = s(a) = t(a)$. As explained above, the preimage of a is a set of cycles and the sum of the lengths of these cycles must be 4. Since $(\mathbf{G}, \lambda)$ is a network that contains neither self-loop, nor multiple arcs, the preimage of a cannot contain cycles of length 1 or 2, and then, the preimage of a is a set of cycles of length 4. Consequently, we can associate to each vertex $x \in \varphi^{-1}(v)$ the label $\alpha(x) = \Lambda(aa)$. If $T_{\alpha(x)}(x) = x$, then it means that there exists a cycle of length 2 in $(\mathbf{G}, \lambda_G)$ which is impossible. Let us now consider $y = T_{\alpha(x)}(x)$. Since $\varphi(y) = v$, $\alpha(y) = \alpha(x)$ and consequently $T_{\alpha(y)}(y) = T_{\alpha(x)}(T_{\alpha(x)}(x)) = T_{\alpha(x)^2}(x) = T_{\Lambda(aaaa)}(x) = x$, since the preimage of a consists of cycles of length 4. Consequently all the vertices in $\varphi^{-1}(v)$ will be paired in $(\mathbf{G}, \lambda_G)$. For all the other vertices we proceed as before.

Suppose that $q = 2$ and that there exists two arcs $a, a' \in A(H)$ such that $s(a) = s(a')$ and $t(a) = t(a')$. Let $v = s(a)$ and consider the cycle $(a, Sym(a'))$ of length 2. The preimage of this cycle in $\mathbf{G}, \lambda_G$ is a set of cycles and the sum of the lengths of these cycles must be 4. As before, it implies that the preimage of this cycle consists of a set of cycles of length 4. Then, one can associate to each vertex $x \in \varphi^{-1}(v)$ the label $\alpha(x) = \Lambda(aSym(a'))$. Consider a vertex $x \in \varphi^{-1}(v)$, if $T_{\alpha(x)}(x) = x$, then it means that there exists a cycle of length 2 in $(\mathbf{G}, \lambda_G)$ which is impossible. Let us now consider $y = T_{\alpha(x)}(x)$. Since $\varphi(y) = v$, $\alpha(y) = \alpha(x)$ and consequently $T_{\alpha(y)}(y) = T_{\alpha(x)}(T_{\alpha(x)}(x)) = T_{\alpha(x)^2}(x) = T_{\Lambda(aSym(a')aSym(a'))}(x) = x$ and consequently all the vertices in $\varphi^{-1}(v)$ will be paired in $(\mathbf{G}, \lambda_G)$. For all the other vertices we proceed as before. □

Theorem 4. *Under the knowledge* [MP], Pairing *can be solved for* $(\mathbf{G}, \lambda_G)$ *whose minimum base is* (H, λ_H) *if one of the following holds:*

(i) (H, λ_H) *has an even number of vertices (i.e.* n/q *is even),*

(ii) there exists a vertex $v \in V(H)$ *and a closed path* $P(v, v)$ *in* (H, λ_H) *such that for any vertex* $u \in \varphi^{-1}(v)$, $T_\alpha(u) \neq u$ *and* $T_\alpha(T_\alpha(u)) = T_{\alpha^2}(u) = u$ *where* $\alpha = \Lambda(P)$ *is the sequence of labels corresponding to the path* $P(v, v)$.

The above result clearly indicates how to solve the Pairing problem, when provided with a map of the graph. Observe that the condition (ii) in Theorem 2 and the conditions $(ii), (iii), (iv), (v)$ in Theorem 3 are particular cases of the condition (ii) in Theorem 4.

4.2 Necessary Conditions

We now show that most of the sufficient conditions presented in Section 4.1 are in fact, also necessary. First we state a general result about solving Pairing in networks whose minimum base has odd number of vertices.

Lemma 4. *If* $(\mathbf{G}, \lambda_G)$ *is a network whose minimum base* (H, λ_H) *has an odd number of vertices, then for any solution to the Pairing problem in* $(\mathbf{G}, \lambda_G)$*, by* *some algorithm $\mathcal{A}$, there exists $v \in V(H)$ such that each node $u \in \varphi^{-1}(v)$ is paired with another node $u' \in \varphi^{-1}(v)$.*

Theorem 5. *Under the knowledge* [MP]*,* Pairing *cannot be solved for any network $(\mathbf{G}, \lambda_G)$ whose minimum base (H, λ_H) has the following properties:*

(i) (H, λ_H) has an odd number of vertices, and

(ii) there does not exist a vertex $v \in V(H)$ and a closed path $P(v, v)$ in (H, λ_H) such that for every vertex $u \in \varphi^{-1}(v)$, $T_\alpha(u) \neq u$ and $T_\alpha(T_\alpha(u)) = T_{\alpha^2}(u) = u$ where $\alpha = \Lambda(P)$ is the sequence of labels of the arcs in $P(v, v)$.

Notice that there are indeed networks (of even size) which satisfy the conditions of Theorem 5, and thus, Pairing is unsolvable in such networks even when a map of the network is available. One such example is shown in Figure 1.

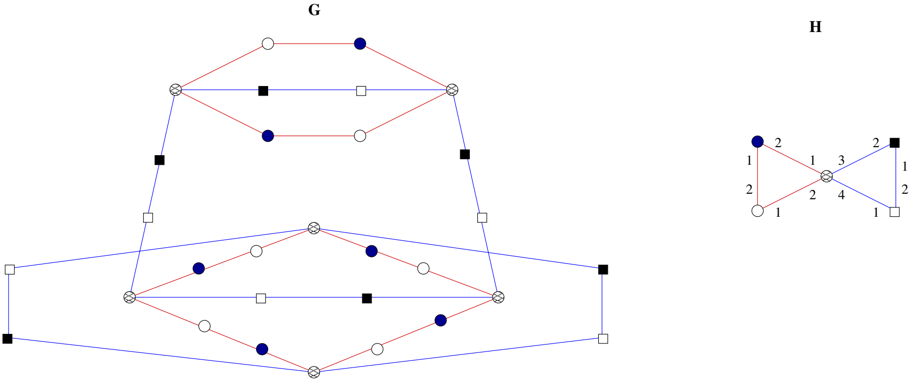


Fig. 1. A simple network, **G** and (its minimum base H) where Pairing is not solvable. Here each edge between two nodes represents a pair of arcs, one in each direction (For clarity, the edge labels have been removed from **G**).

Theorem 6. *Under the knowledge* [ES]*,* Pairing *is not solvable for any network $(\mathbf{G}, \lambda_G)$ having minimum base (H, λ_H) and symmetricity q, if all of the following hold:*

(i) (H, λ_H) has an odd number of vertices,

(ii) the minimum base has strictly more than $2|V(H)|$ arcs , i.e., $|A(H)| > 2n/q$,

(iii) there does not exist any self-loop in (H, λ_H), and

(iv) there does not exist two distinct arcs $a, a' \in A(H)$ such that $s(a) = s(a')$ and $t(a) = t(a')$.

The above result shows that there is a gap between the necessary and sufficient conditions for the case when the exact network size is known. In fact, if the minimum base of the network contains asymmetric self-loop and parallel arcs, then we do not know the exact conditions necessary for solving the Pairing problem. In these cases, it may be possible to characterize the networks in terms of the number of self-loops or parallel arcs in their minimum base, and thus minimize this gap between the necessary and sufficient conditions.

Theorem 7. *Under the knowledge* [UB]*,* Pairing *is not solvable in any network* $(\mathbf{G}, \lambda_G)$ *having minimum base* (H, λ_H)*, if the following holds:*

(i) (H, λ_H) *has an odd number of vertices, and*
(ii) there does not exist any symmetric self-loop in (H, λ_H) *(i.e. an arc a such that Sym(a)=a).*

Proof. Let (H, λ_H) be any symmetric digraph with odd number of vertices and having no symmetric self-loops. If there is an algorithm $\mathcal{A}$ that solves *Pairing* in $(\mathbf{G}, \lambda_G)$ under the knowledge [UB] then this algorithm should work for every network $(\mathbf{G}', \lambda'_G)$ which covers (H, λ_H) (the algorithm cannot differentiate between two networks with a common minimum base, when only an upper bound on the network size is known). We shall now show that the algorithm $\mathcal{A}$ would fail for at least one network $(\mathbf{G}, \lambda_G)$ that covers (H, λ_H). First note that $|A(H)| \geq 2 \cdot |V(H)|$ because otherwise either G is disconnected or G is an odd-sized tree (where Pairing is not possible anyway).

If $|A(H)| \geq 2\ |V(H)|$, then there exists an arc $a \in A(H)$ such that $H' = H \setminus \{a, Sym(a)\}$ is strongly connected. (a could be either a self-loop or one of the pair of parallel arcs or, one of the arcs in a cycle). Let $u = s(a)$, $v = t(a)$, and $a' = Sym(a) \neq a$. Consider the connected digraph H' that is obtained from H by removing the arcs a and a'. Suppose $(\mathbf{G}', \lambda'_G)$ be a network of odd size, whose minimum base is (H', λ_H). Notice that it is always possible to construct such a G', if H' has no symmetric self-loops.

We now construct two networks $(\mathbf{G}_1, \lambda_{G1})$ and $(\mathbf{G}_2, \lambda_{G2})$ defined as follows. To construct $(\mathbf{G}_1, \lambda_{G1})$, we take 4 distinct copies $(\mathbf{G}'_0, \lambda'_0), \ldots, (\mathbf{G}'_3, \lambda'_3)$ of $(\mathbf{G}', \lambda'_G)$. We will denote by $u_{i1}, u_{i2}, ...$ (resp. $v_{i1}, v_{i2}, ...$) the vertices that corresponds to u (resp. v) in $(\mathbf{G}'_i, \lambda_i)$. We then add the arc a_{ij} with the same label as a (and the symmetric arc a'_{ij} with the same label as a') between u_{ij} and v_{rj}, $r = i + 1 \mod 4$ for all i, j. To construct $(\mathbf{G}_2, \lambda_{G2})$, we do the same but we consider 8 distinct copies of $(\mathbf{G}', \lambda'_G)$. Clearly, the two graphs we have constructed are symmetric coverings of (H, λ_H). Thus, due to Lemma 4, there exists a vertex $v \in V(H)$, such that all nodes in $\varphi^{-1}(v)$ are paired among themselves, both in network $(\mathbf{G}_1, \lambda_{G1})$ and network $(\mathbf{G}_2, \lambda_{G2})$.

Due to property 2, there exists an execution of $\mathcal{A}$ on $(\mathbf{G}_1, \lambda_{G1})$ (respectively $(\mathbf{G}_2, \lambda_{G2})$) where the each node in the pre-image of v computes the same label sequence α as computed by v in an execution of $\mathcal{A}$ on (H, λ_H). Consider the path $P(v, v)$ in (H, λ_H), which corresponds to the sequence α. Let $|P|_a$ (resp. $|P|_{a'}$) be the number of times the arc a (resp, a') appears in P and let $n_a = |P|_a - |P|_{a'}$.

CLAIM(1): n_a is of the form $4r_1 + 2$ for some integer r_1.
CLAIM(2): n_a is of the form $8r_2 + 4$ for some integer r_2.

To see why the first claim is true, consider the subgraph G'_i of G_1. There are an odd number of vertices in G'_i, which belong to the preimage of v. Thus, at least one of these nodes must be paired with a node in some other subgraph $G'_j, j \neq i$ of G_1. (Notice that whenever we traverse an arc belonging to the pre-image of a, we move from one subgraph G'_i to the next $G'_{i+1 \mod 4}$.) Thus, in this case, $j = i + n_a \mod 4$ and also $i = j + n_a \mod 4$. So, n_a must be of the form $4r_1 + 2$ for some integer r_1. This proves the first claim. For the second claim, we consider the graph $(\mathbf{G}_2, \lambda_{G2})$ where, using a similar argument we can show that n_a must be of the form $8r_2 + 4$ for some integer r_2.

Note that the two claims above cannot be true simultaneously. This implies that the algorithm $\mathcal{A}$ must fail for one of the networks $(\mathbf{G}_1, \lambda_{G1})$ or $(\mathbf{G}_2, \lambda_{G2})$. □

Due to the earlier results and Theorem 7, we have a complete characterization of those networks where *Pairing* is solvable when provided with an upper bound on the network size.

5 Conclusions and Open Problems

In this paper, we studied the problem of *k-Grouping* which is a generalization of the node-enumeration or the election problem, in anonymous networks. In particular we also studied the problem of *Matching* or *Pairing* the nodes of the network. For the *Pairing* problem, the solvability depends on the amount of prior knowledge available. When an upper bound on the network size is known, it is possible to compute the minimum base for the network. We characterized the solvable instances of the *Pairing* problem in terms of the minimum base of the network. When the exact network size is known, the network can be represented by its minimum base and its symmetricity. In this case, the characterization presented in this paper is not complete and there is a gap between the necessary and sufficient conditions, which needs to be investigated. Another possible extension of this work would be to study the generalization of the *Pairing*(or 2-*Relating*) problem to other values of k (say for $k = 3, 4, 5, \dots$) or for arbitrary values of k. It would also be interesting to consider the problem of approximate k-groupings in the case when k does not divide n.

Acknowledgments. The authors would like to thank Paola Flocchini for the many helpful discussions and comments.

References

1. D. Angluin. Local and global properties in networks of processors. In *Proc. 12th ACM Symp. on Theory of Computing* (STOC '80), 82–93, 1980.
2. P. Boldi, B. Codenotti, P. Gemmell, S. Shammah, J. Simon, and S. Vigna. Symmetry breaking in anonymous networks: Characterizations. In *Proc. of 4th Israeli Symposium on Theory of Computing and Systems*(ISTCS'96), 16–26, 1996.
3. P. Boldi and S. Vigna. An effective characterization of computability in anonymous networks. In *Proc. 15th Int. Conference on Distributed Computing (DISC'01)*, 33–47, 2001.

4. P. Boldi and S. Vigna. Fibrations of graphs. *Discrete Math.*, 243:21–66, 2002.
5. A. Cardon and M. Crochemore. Partitioning a Graph in $O(|A|log2|V|)$. *Theoretical Computer Science*,19:85–98, 1982.
6. J. Chalopin and Y. Métivier. A bridge between the asynchronous message passing model and local computations in graphs (*extended abstract*). In *Proc. of Mathematical Foundations of Computer Science, MFCS'05*, volume 3618 of *LNCS*, pages 212–223, 2005.
7. D.G. Corneil and C.C. Gotlieb. An Efficient Algorithm for Graph Isomorphism. *Journal of the ACM*, 17(1):51–74, 1970.
8. P. Flocchini, A. Roncato, and N. Santoro. Computing on anonymous networks with Sense of Direction. *Theoretical Computer Science*, 301, 355-379, 2003.
9. E. Godard, Y. Métivier, and A. Muscholl. Characterization of classes of graphs recognizable by local computations. *Theory of Computing Systems*, 37(2):249–293, 2004.
10. R.E. Johnson and F.B. Schneider. Symmetry and similarity in distributed systems. In *Proc. 4th Annual ACM Symp. on Principles of Distributed Computing* (PODC '85), 13–22, 1985.
11. E. Kranakis. Symmetry and computability in anonymous networks: A brief survey. In *Proc. 3rd Int. Conf. on Structural Information and Communication Complexity* (SIROCCO '97), 1–16, 1997.
12. F. T. Leighton. Finite common coverings of graphs. *J. Combin. Theory, Ser. B*, 33:231–238, 1982.
13. A. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61(5):233–239, 1997.
14. S.J. Mullender and P.M.B. Vitanyi. Distributed match-making. *Algorithmica*, 3: 367–391, 1998.
15. N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proc. 18th ACM Symposium on Principles of Distributed Computing* (PODC '99), 173–179, 1999.
16. M. Yamashita and T. Kameda. Computing on anonymous networks: Parts I and II. *IEEE Trans. Parallel and Distributed Systems*, 7(1):69–96, 1996.
17. M. Yamashita and T. Kameda. Computing functions on asynchronous anonymous networks. *Mathematical Systems Theory*, 29:331–356, 1996.

A New Proof of the GHS Minimum Spanning Tree Algorithm

Yoram Moses and Benny Shimony

Department of Electrical Engineering,
Technion—Israel Institute of Technology, Haifa 32000, Israel
moses@ee.technion.ac.il, shimonyb@tx.technion.ac.il

Abstract. This paper provides a proof of correctness for the celebrated Minimum Spanning Tree protocol of Gallager, Humblet and Spira [GHS83]. Both the protocol and the quest for a natural correctness proof have had considerable impact on the literature concerning network protocols and verification. We present an invariance proof that is based on a new intermediate-level abstraction of the protocol. A central role of the intermediate-level configurations in the proof is to facilitate the statement of invariants and other properties of the executions of GHS at the low level. This provides a powerful tool for both the statement and the proof of properties of the algorithm. The result is the first proof that follows the spirit of the informal arguments made in the original paper.

1 Introduction

In a seminal paper, Gallager, Humblet and Spira presented a communication-efficient protocol for computing the minimum spanning tree (MST) of an asynchronous network in which communication links have unique weights [GHS83]. The protocol is very elegant and intuitive. It is also special for employing a very high degree of asynchrony among the nodes of the network, without forcing its execution to proceed in a well-formed sequence of phases of computation. The importance of the GHS protocol has made the task of proving its correctness a natural challenge for verification and formal methods. The arguments given in [GHS83] for why the protocol should be correct, while informal, are convincing. Formalizing them, however, has proven to be extremely elusive. Two PhD theses have offered extensive refinement proofs of its correctness [W88, S01]. Both of these proofs are quite long (circa 170-180 pages). Many other attempts to prove the correctness of the GHS algorithm have been attempted. A few of them resulted in proofs of modified versions of the algorithm, which solve the MST problem with similar complexity [CG88, Hes99, JZ92, SdR87]. In granting the Edsger W. Dijkstra prize[1] in distributed computing: 2004 for [GHS83], the committee pointed out that "Finding a proof [of correctness for the GHS algorithm] that copes with the intricacies of this algorithm in a natural way is still very

[1] See www.podc.org/dijkstra/2004.html

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 120–135, 2006.

much an open problem in protocol verification and formal methods." Our goal in this paper is to present such a proof. The full version of our proof is roughly half the size of the previous proofs, but presenting it in detail is still beyond the scope of this abstract. We will therefore attempt to describe the overall structure of the proof, the novel aspects of the proof technique used, and to provide some insights into what makes proving this particular protocol elusive.

The GHS algorithm grows a minimum-spanning forest by adding MST edges one by one, until the forest turns into the MST.[2] In the spirit of Borůvka's centralized MST algorithm [Bor26], each fragment in the forest computes its minimum-weight outgoing edge (mwe), and adds it to the forest. In order to control the communication costs, the GHS algorithm enables only some MST edges to be added to the forest at any given stage, while blocking others. As a result, one well-known issue underlying the correctness of GHS is the fact that progress in the search for an mwe in one fragment depends on what happens in neighboring fragments. Indeed, only progress of searches in fragments of the lowest existing level is guaranteed, and even this is true only provided that no sleeping node is awakened. Carefully capturing this interdependence is subtle. The GHS algorithm makes use of a number of optimizations (such as not sending a `Reject` message when it can be inferred by the recipient), adding a further layer of complexity to any proof of correctness. We believe that a major barrier to taming the correctness proof of the GHS algorithm comes from a lesser-known subtlety in the algorithm: While the GHS algorithm does not employ all of the nodes of a fragment in the search for the fragment's mwe, it is not possible in general to use a node's local information (local state and local history) in order to determine its role in the current search. More specifically, the following three states of a node v in the network with respect to the search in its current fragment are undistinguishable:

(a) v has participated in the current search and completed its role in the search;
(b) v will participate in the current search, but has not actively started its participation; and
(c) v will not participate in the current search.

This matters in reasoning about the GHS algorithm because nodes in each of the three states satisfy fundamentally *different* invariance properties.

This paper offers a new approach and new invariance proof for the GHS algorithm. Its main contributions are:

- The proof is natural in that
  - Correctness is proven for the original algorithm within the exact model and assumptions of [GHS83];
  - The proof directly formalizes the informal argument for correctness sketched in the original paper [GHS83], and its development provides insight into what makes proving the correctness of GHS difficult; and

[2] We assume that the reader is familiar with the GHS algorithm, as presented in [GHS83].

  - The full proof is roughly half the length of the previous proofs (under 100 pages long, most of which is a simple case analysis to verify that all invariants hold under all possible transitions). Given that the GHS algorithm contains seven possible message types, nine variables per process and over fifty possible branches of control when treating a message, any rigorous manual proof would necessarily involve a few tens of invariants, whose verification would require tens of pages of (boring) detail.
- We introduce a new intermediate-level description of the GHS algorithm, called the *rainbow construction*. It is obtained by abstracting away all communication events and most of the state information in the GHS execution, and maintains just enough detail to properly account for the state of nodes with respect to the search in their fragments. The rainbow construction refines the standard high-level abstraction of the GHS algorithm at the level of a forest of fragments, and its configuration (called a *painted forest*), is much more compact and intuitive than that of the detailed GHS algorithm. The rainbow construction is of independent interest, as it may be a useful level of abstraction for simplifying other proofs of GHS, such as those of [W88, S01].
- We use configurations of the intermediate-level rainbow construction as auxiliary global variables associated with configurations of the detailed GHS algorithm. Safety and Liveness properties of GHS are then stated based on the values and terminology of the intermediate configuration.
- By breaking the GHS algorithm into three levels of abstraction and reasoning about the algorithm at each of the levels, this proof provides insight into the issues and sources of difficulty involved in establishing the correctness of the GHS algorithm.

This paper is structured as follows. Section 2 describes a high-level construction in the style of [Bor26] that is typically used to describe the GHS protocol at the level of fragments of a minimum-spanning forest. Section 3 refines the top-level view to the rainbow construction, which accounts for essential aspects of the GHS algorithm, while abstracting away communicated messages and details of variable's values. Essential structural properties of the executions of GHS are established at the level of the rainbow construction. Section 4 overviews our correctness proof for the GHS algorithm. It describes how the intermediate-level configurations are used for expressing invariants of the GHS algorithm, and sketches the liveness proof that ties all of the pieces together, formalizing the intuitive argument in the original paper [GHS83].

2 The Top-Level View of GHS

Given is an asynchronous network modelled as a weighted, connected undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \_)$, in which the edge weights are denoted by $\underline{e}$ for each edge $e \in \mathbf{E}$. The edge weights are assumed to be pairwise distinct, and hence $\mathbf{G}$ has a unique minimum-spanning tree (see, e.g., [Bol79, Har69]), which we shall denote simply by MST. Every node of $\mathbf{V}$ stands for a processor, and edges stand for bidirectional FIFO communication links. Processors are aware of the ports and weights of the edges they are adjacent on. The problem is to design an efficient distributed algorithm that will result in every node marking each of its local ports (standing for incident edges) as either Branch if it is an MST edge, or as Rejected if it is not.

The GHS algorithm can be viewed as an attempt to distribute Borůvka's centralized MST algorithm [Bor26]. Starting from singleton nodes, it "grows" fragments by adding MST edges one by one. Every edge added to the forest combines a pair of fragments to create a larger fragment. We will denote the number $|\mathbf{V}|$ of processors by n throughout the paper. Once $n-1$ edges have been added, there is a single fragment left, and it is the MST. By assigning a level to each fragment, and taking care that fragments combine only in specific ways, the GHS algorithm ensures that a node will participate in at most $\log n$ searches for the minimum weight edge of a fragment. As a result, the protocol uses $O(|\mathbf{V}|\log|\mathbf{V}|+|\mathbf{E}|)$ short messages.

2.1 Minimum Spanning Forests

The GHS algorithm maintains a forest of *fragments*, each of which corresponds to a triple $F=(L,c,G)$, where L—the *level* of F—is a natural number, $G=(V,E)$—the *graph* of F—is a subtree of the MST, and either

- $F=(0,\{\},(\{v\},\{\}))$, where $v\in\mathbf{V}$ (so F is a singleton node and its core is the empty set), or
- $L>0$ and $c=\{v,w\}\in E$. In this case, we call $c=\{v,w\}$ the *core edge* of F, with v and w being its *core nodes.*

A *minimal spanning forest (msf)* of $\mathbf{G}$ is a set $\mathcal{F}=\{F_1,\ldots,F_k\}$ of fragments whose node sets define a partition of $\mathbf{V}$. Thus, every node of $\mathbf{G}$ appears in exactly one fragment of $\mathcal{F}$. We use $F_v=(L_v,c_v,G_v)$ to denote the fragment of a node v and its components.

The edges of the fragments $F\in\mathcal{F}$ are called *forest edges* (of $\mathcal{F}$). If $\{u,v\}$ is a forest edge, then the directed edge (u,v) is called *inbound* if (i) $\{u,v\}\neq c_v$ and (ii) v is closer to c_v than u is. If (u,v) is inbound, we consider (v,u) to be *outbound.* These notions extend to paths. A directed path of fragment edges is called *inbound* (resp., *outbound*) if all the edges in the path are inbound (resp., outbound). We denote by T_v the graph consisting of all outbound paths that start at v. Thus, T_v is a subtree of F_v rooted at v. With each node in a non-singleton fragment we associate a *parent* node as follows. For nodes v and w on a core edge (i.e., if $c_v=\{v,w\}$), we say that v is the parent of w (and vice-versa), and write $\mathit{parent}(w)=v$. Otherwise, we define $\mathit{parent}(w)=v$ to hold if (w,v) is inbound. As a partial converse, we define the set of *children* of v by $\mathit{child}(v)=\{w:(v,w)\text{ is outbound}\}$. Notice that nodes of a core edge are *not* children of one another.

An edge $e=\{v,w\}\in\mathbf{E}$ is an *external edge* (w.r.t. $\mathcal{F}$) if $F_v\neq F_w$. Let V be a subset of nodes of $\mathbf{V}$. We define $\mathsf{mwe}(V,\mathcal{F})$ to be the edge of minimal weight among the external edges in $\mathcal{F}$ that have at least one node in V. The parameter $\mathcal{F}$ is omitted whenever the forest is clear from context, so e.g., we write $\mathsf{mwe}(V)$ instead of $\mathsf{mwe}(V, \mathcal{F})$. Again for ease of exposition, we shall abuse notation slightly and allow, for any structure X that contains a set of nodes $V(X)$, to write $\mathsf{mwe}(X)$ instead of $\mathsf{mwe}(V(X))$. We denote the weight of $\mathsf{mwe}(V)$ by $\underline{\mathsf{mwe}}(V)$.[3] If there is no external edge in $\mathcal{F}$ with at least one node in V, then $\mathsf{mwe}(V)$ does not exist, and we define $\underline{\mathsf{mwe}}(V) = \infty$. Finally, if $\mathsf{mwe}(V) = \{v, w\}$ where $v \in V$, then we denote by $\mathsf{dmwe}(V)$ the directed edge (v, w). This is the result of *directing* $\mathsf{mwe}(V)$ *away* from V.

2.2 The Merge and Absorb (M&A) Construction

At the level of fragments and forests, the GHS algorithm starts out with an *initial msf* $\mathcal{F}_\iota$ consisting of all singleton fragments. Adding an MST edge to the forest combines two fragments into one. This can occur in two manners: One is a *merge*, in which two fragments of equal level L combine, and form a fragment of level $L + 1$ whose core is the MST edge that was just added. The second is via an *absorb*, in which a fragment of lower level is added into one with a higher level. The resulting fragment maintains the level and core of the latter. The Merge and Absorb operations combining two fragments $F = (L, c, G)$ and $F' = (L', c', G')$ are formally defined in Table 1. Given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ we use $\mathsf{comb}(G_1, e, G_2)$ in the table to denote the graph $G = \big((V_1 \cup V_2), (E_1 \cup E_2 \cup \{e\})\big)$. The Guard field in the table describes the conditions under which an operation may be performed, and the Action field describes the transition caused by this operation.

Table 1. The M&A Construction

Operation	Guard Condition	Action
Merge(F, F')	$L = L'$, $\mathsf{mwe}(F) = \mathsf{mwe}(F') = e$	$\mathcal{F} \leftarrow \mathcal{F} \cup \{F''\} \setminus \{F, F'\}$, where $F'' = \big(L + 1, e, \mathsf{comb}(G, e, G')\big)$
Absorb(F, F')	$L < L'$, $\mathsf{mwe}(F) \cap V' \neq \emptyset$	$\mathcal{F} \leftarrow \mathcal{F} \cup \{F''\} \setminus \{F, F'\}$, where $F'' = \big(L', c', \mathsf{comb}(G, \mathsf{mwe}(F), G')\big)$

We view Table 1 as defining a nondeterministic (sequential) construction that starts with $\mathcal{F}_\iota$ and where at each step one operation whose guard condition is enabled is performed, for as long as enabled operations exist. Obviously, this construction cannot perform more than $n - 1$ operations, since at each step one new MST edge is added to the forest.

We write $\mathcal{F} \mapsto \mathcal{F}'$ if $\mathcal{F}'$ can be obtained from $\mathcal{F}$ by performing a single Merge or Absorb operation as described above. To capture reachability in the M&A construction, we define $\rightsquigarrow$ to be the reflexive transitive closure of $\mapsto$. Thus,

[3] Note that edge-weights are not specified explicitly in subgraphs and fragments, since they are always inherited from $\mathbf{G}$.

$\mathcal{F} \rightsquigarrow \mathcal{F}'$ if $\mathcal{F}'$ can be obtained by starting from $\mathcal{F}$ and performing a sequence of (zero or more) Merge and Absorb operations. We will only be interested in forests $\mathcal{F}$ that are reachable from the initial forest $\mathcal{F}_\iota$, so that $\mathcal{F}_\iota \rightsquigarrow \mathcal{F}$.

A number of properties that are relevant in the analysis of the GHS algorithm can be shown already at the level of the M&A construction. Such, for example, is the well-known fact that fragment levels can never exceed $\log n$. Other properties are useful tools for bounding the dynamic behavior of the GHS algorithm. We think of the GHS algorithm in terms of broadcast and convergecast waves generated from and returning to the core of a fragment. In the M&A construction, the inbound/outbound orientation of a directed forest edge (v, w) can change as a result of a Merge or Absorb operation. It is not immediate that these changes do not interfere with these communication waves. The following claim shows essentially that such changes occur only on the path between a core edge and the fragment's mwe:

Lemma 2.1. *Let (v, w) be an outbound edge in the $\mathcal{F}$, and assume that $\mathcal{F} \mapsto \mathcal{F}'$. If (v, w) is inbound in $\mathcal{F}'$, then $\mathsf{mwe}(T_v, \mathcal{F}) = \mathsf{mwe}(F_v, \mathcal{F})$.*

During the reporting (convergecast) phase of a search, nodes send information about the mwe of their subtree towards the core. A node's subtree, however, might change before this message even arrives at its parent. The following property is used to show that the reported information remains valid. We say that a directed external edge (v, w) is *ascending* if $L_v \leq L_w$, and (v, w) is *strictly ascending* if $L_v < L_w$. We can show

Lemma 2.2. *If $\mathsf{dmwe}(T_v)$ is ascending, then $\mathsf{mwe}(T_v)$ remains unchanged as long as the identity of c_v is unchanged.*

In the convergecast phase of the search for an $\mathsf{mwe}(F_v)$, a node v sends the value of $\underline{\mathsf{mwe}}(T_v)$ to its ancestors only if $\mathsf{dmwe}(T_v)$ is ascending.[4] Lemma 2.2 establishes that this information will remain valid throughout the search for $\mathsf{mwe}(F_v)$.

3 The Rainbow Construction

A central tool in our analysis of the GHS algorithm is a novel intermediate-level abstraction called the *rainbow construction*. The M&A construction abstracts away the process of searching for an $\mathsf{mwe}(F)$. The rainbow construction tries to mimic essential aspects of the search process in the GHS algorithm, while completely abstracting away details of communication between processes and practically all information about local variables. The rainbow construction operates on an enriched ghs forest. In addition to the partition into fragments, it assigns *color* states to nodes in order to keep track of their role in the current fragment's search. In the spirit of the GHS algorithm, a non-singleton fragment is identified with its core edge; in particular, we think of searches as being performed on behalf of a given core edge.

[4] In fact, an outgoing edge node is found out to be external only if it is ascending.

Define the set $\mathsf{Colors} = \{\mathsf{slp}, \mathsf{wkn}, W, G, R, B\}$ of possible colors for nodes, whose elements stand for Asleep, Awake, White, Green, Red and Blue, respectively. The colors slp and wkn will be used to describe the state of a node in a singleton fragment. In non-singleton fragments, a node is White if it will participate in the current search, but hasn't actively joined the search yet. A node turns Green when it starts to actively participate in the search, and then becomes Red once it has completed its role in the search. Finally, a Blue node is excluded from participating in the current search. Roughly speaking, since the search is activated via an outbound (broadcast) wave from the core edge, and completes via in inbound (convergecast) wave, the search completes once both core nodes are Red.

Formally, a configuration of the rainbow construction is represented by a *painted forest*, which is a pair $\mathcal{P} = (\mathcal{F}, \mathsf{col})$ consisting of a ghs forest $\mathcal{F}$ and an assignment $\mathsf{col} : \mathbf{V} \to \mathsf{Colors}$ of colors to the nodes. With respect to a painted forest $\mathcal{P}$, a directed edge (v, w) is *pending*, denoted by $\mathsf{Pend}(v, w)$, if either (i) $L_v = 0$ (so v is a singleton), $\mathsf{col}(v) = \mathsf{wkn}$, and $\mathsf{dmwe}(v) = (v, w)$, or (ii) $L_v > 0$, both nodes of the core edge c_v are Red, and $\mathsf{dmwe}(F_v) = (v, w)$. Pending edges play a role in our definition of the rainbow construction. Intuitively, a pending edge is an MST edge for which the search has completed; the edge will, in due time, be added to the forest.

Table 2. The Rainbow Construction

Op	Guard Condition	Action	description
$\mathbf{M}[v, w]$	$\mathsf{Pend}(w, v)$ and $\mathsf{Pend}(v, w)$	$\mathsf{Merge}(F_v, F_w)$ $\mathsf{col}(V_w \cup V_v) \leftarrow W$	*merge on edge* $\{v, w\}$
$\mathbf{A}[w, v]$	$\mathsf{Pend}(w, v)$	$\mathsf{Absorb}(F_w, F_v)$ $\mathsf{col}(V_w) \leftarrow \begin{cases} B & \mathsf{col}(v) \in \{R, B\} \\ W & \text{otherwise.} \end{cases}$	*absorb on edge* (w, v)
$\mathbf{G}[v]$	$\mathsf{col}(v) = W$ and either (i) $v \in c_v$ or (ii) $v \in \mathit{child}(w)$, $\mathsf{col}(w) = G$	$\mathsf{col}(v) \leftarrow G$	*search at v begins* (`Initiate`(..Find) received)
$\mathbf{R}[v]$	$\mathsf{col}(v) = G$ $\mathsf{mwe}(v)$ is ascending $\forall u \in \mathit{child}(v) : \mathsf{col}(u) = R$	$\mathsf{col}(v) \leftarrow R$	*search at v ends* (`Report` sent by v)
$\mathbf{Wa}[v]$	$\mathsf{col}(v) = \mathsf{slp}$	$\mathsf{col}(v) \leftarrow \mathsf{wkn}$	*wakeup*

The rainbow construction, depicted in Table 2, starts out with the *initial* painted forest $\mathcal{P}_\iota = (\mathcal{F}_\iota, \mathsf{col}_\iota)$, where $\mathcal{F}_\iota$ is the initial msf from the top level construction, and $\mathsf{col}_\iota(v) = \mathsf{slp}$ for all $v \in \mathbf{V}$. The construction consists of five operations that may be applied to a painted forest. As in the M&A construction, each operation is specified by a guard condition, and an action on the painted forest that is enabled (allowed) if the guard is true. Two of the operations add an edge to the forest and perform Merge and the Absorb on fragments, while the other three change the color of nodes to track progress in the search at the node level. In the latter, a node v turns Green by $\mathbf{G}[v]$, which is allowed only if v is White and is either a core node or the child of a Green node. It turns Red via $\mathbf{R}[v]$, which may happen only if it is Green, all of its children are Red, and its local mwe is ascending. More trivially, $\mathbf{Wa}[v]$ corresponds to a singleton node waking up. The Merge and Absorb operations in the rainbow construction extend those of the M&A construction by keeping track of the role of nodes in the fragments whose core is modified by the operation. (In the Table 2, we denote by $\mathsf{col}(V') \leftarrow X$ the simultaneous assignment of the color X to all nodes of V'.) A Merge colors all nodes in the resulting fragment White, while the Absorb $\mathbf{A}[w, v]$ comes in two flavors: It paints the nodes of the absorbed fragment F_w White if they are intended to participate in the search in the combined fragment, and paints them Blue if not. In the spirit of the M&A construction, we interpret the rainbow construction as performing at each step one nondeterministically chosen operation whose guard condition is enabled, for as long as enabled operations exist.

We write $\mathcal{P} \mapsto \mathcal{P}'$ if $\mathcal{P}'$ can be obtained from $\mathcal{P}$ by performing a single step of the rainbow construction, and define $\rightsquigarrow$ on painted forests as the reflexive and transitive closure of $\mapsto$. As with ghs forests, we will be concerned exclusively with painted forests that are reachable from the initial forest $\mathcal{P}_\iota$.

3.1 Properties of the Rainbow Construction

In terms of the colors provided by the rainbow construction, the first phase of a “search” for the mwe in a fragment consists of an outbound moving front from the core edge that turns White nodes to Green. This is followed by an inbound front moving from the outermost formerly White nodes to the core, which turns Green nodes to Red. We think of the search as having completed once both core nodes are Red. The following lemma characterizes the patterns of colors that can appear on outgoing paths starting from the core:

Lemma 3.1. *Let $p = u_0, u_1, \ldots, u_k$ be an outbound path starting at a core node u_0. The sequence $\langle \mathsf{col}(u_0), \ldots, \mathsf{col}(u_k) \rangle$ of colors along p forms a string in the regular language $G^*W^* + G^*R^*RB^*$.*

A major difficulty in any attempt to rigorously reason about the GHS algorithm is caused by the fact that the sets of nodes and edges in a fragment change while the search within the fragment is being performed. An induction on the number of steps performed in the rainbow construction can be used to show:

Lemma 3.2. *Let $\mathsf{col}(v) \in \{W, G, R\}$. Then c_v is unchanged as long as $\mathsf{col}(v)$ does not turn from R to non-R.*

The combination of Lemmas 3.1 and 3.2 guarantees that the outbound paths from the fargment’s core remain stable during the broadcast and convergecast activities involved in the search for a fragment’s mwe. The analogous result supporting the success of broadcast to Blue (non-searching) nodes is slightly more subtle, and it depends on a refinement of Lemma 2.1 from the M&A construction, which states that an edge can change orientation only if both its nodes are Red:

Lemma 3.3. *Assume that $\mathcal{P} \mapsto \mathcal{P}'$ and that (v, w) is not inbound in $\mathcal{P}$ but is inbound in $\mathcal{P}'$. Then $\mathsf{col}(v, \mathcal{P}) = \mathsf{col}(w, \mathcal{P}) = R$.*

Based on Lemma 3.3 and Lemma 3.1, the following lemma will allow us to show that paths near the leaves are stable for long enough to ensure the success of broadcast to non-searching (Blue) nodes in the GHS algorithm:

Lemma 3.4. *Let $\mathsf{col}(w) = B$ and assume that p is a path with color pattern RB^*B ending in node w. Then p remains outbound as long as $\mathsf{col}(w) \neq G$.*

While reasoning about paths in the searching part of the fragment, as done in Lemma 3.2, does not appear to be very complex, the ability to express and prove properties such as Lemma 3.4, concerning the distant "dark side" of the fragment, is another indication of the power of the rainbow construction. Both lemmas are instrumental in establishing liveness of the GHS algorithm.

Lemma 3.1 is used both for proving other structural properties of painted forests and the rainbow construction, and to considerably reduce the case analysis involved in proving invariants of the GHS algorithm. We now present two useful claims whose proofs make essential use of Lemma 3.1.

The Absorb operation $\mathbf{A}[w, v]$ in the rainbow construction has two possible outcomes, that differ in the colors they assign to nodes of the absorbed fragment. Specifically, the nodes of an absorbed fragment F_w are colored Blue if the connecting node v at the absorbing fragment is either Red or Blue; otherwise, the nodes of F_w are colored White. Notice that a node's role in the search process of its current fragment, as determined by the distinction between Blue (non-participant) and White (future participant) and Red (past participant) depends on an event that occurs at the connecting edge in the absorb operation, which may be very distant from the node. In particular, it is independent of the node's actions or history. The distinction between White and Blue nodes is worthwhile, because nodes of these colors satisfy different invariants, as we shall illustrate in the next section. Intuitively, the GHS algorithm ignores the mwe's of Blue nodes during the search for a fragment's mwe. Using Lemma 2.1 and 2.2 from the top-level M&A construction and Lemma 3.1, we can show that this is justified by

Lemma 3.5. *If $\mathsf{col}(u) = R$ then $\mathsf{mwe}(T_u)$ depends only on the non-Blue nodes of T_u.*

As a result, if u has reported a value for $\mathsf{mwe}(T_u)$ in the convergecast phase, and a fragment is absorbed at one of the nodes in u's subtree, then the reported value remains correct, because the absorbed fragment consists exclusively of Blue nodes. By Lemma 3.5, we can now show:

Lemma 3.6. *If $\mathsf{Pend}(v, w)$ then (v, w) is ascending.*

The proof is roughly as follows. By definition of $\mathsf{Pend}(v, w)$, both nodes of c_v are Red. Lemma 3.5 implies that $\mathsf{col}(v) \neq B$, and so by Lemma 3.1 we obtain that $\mathsf{col}(v) = R$. The guard for $\mathbf{R}[v]$ in the rainbow construction ensures that $\mathsf{mwe}(v)$ is ascending when v turns Red, and by Lemmas 2.2 and 3.2 it remains ascending as long as $\mathsf{col}(v) = R$. Lemma 3.6 implies that when the guard conditions for $\mathbf{M}[v, w]$ and $\mathbf{A}[v, w]$ are satisfied in the rainbow construction, the guards of their counterpart Merge and Absorb operations in the M&A construction hold as well.

4 Proving Correctness of the GHS Algorithm

The top-level and the rainbow constructions play two roles in our proof. First of all, they allow us to reason about certain aspects of the algorithm at a higher level of abstraction, avoiding many details of the low level. The second, less common, role is to provide a compact description of the current state of the algorithm at a given time. This provides us with an invaluable tool for reasoning about properties of the algorithm at the detailed low level. Various properties that are very difficult to express solely at the level of the GHS algorithm become expressible in a reasonably compact way. In this section we present a rough overview of the proof, and illustrate the role that the higher-level constructions play in the proof.

The correctness proof of the GHS algorithm is performed by reasoning about executions of the GHS program. For the model of computation we adopt the exact model[5] described in [GHS83]. A *run* of the (standard) GHS algorithm is a sequence $C_0, \alpha_1, C_1, \alpha_2, C_2, \ldots$, where C_0 is an initial global state, and the α_i are scheduler actions. Processor activations are interleaved and each response to a scheduler action executes atomically to completion. Formally, the MST problem is specified as follows:

Definition 4.1. MST Problem Specification: *For every weighted network* $\mathbf{G} = (\mathbf{V}, \mathbf{E}, \_)$ *with unique weights, if at least one node spontaneously awakens, the algorithm must eventually reach a configuration in which:*

ST1 Every node in $\mathbf{V}$ *marks each of its ports as either* Branch *(an MST edge) or* Rejected *(a non-MST edge); and*
ST2 All channels are empty.

In order to use the rainbow construction as a description of the GHS executions, we augment the GHS algorithm by adding, for each of the five rainbow construction operations, a single line at an appropriately chosen location in the text of the GHS algorithm, at which the rainbow operation is performed. An execution of the augmented algorithm constructs an execution of the rainbow construction

[5] We make one additional assumption regarding an initial value for one of the local variables, since without this extra assumption the GHS algorithm is incorrect, as discovered by Chou [C88].

in tandem with the execution of the standard GHS algorithm. Starting from an extended initial configuration of the form $\mathcal{C}_0 = (C_0, \mathcal{P}_\iota)$, the augmented algorithm maintains an extended configuration of the form $\mathcal{C} = (C, \mathcal{P})$, where C is a global state of the original GHS algorithm, and $\mathcal{P}$ is a painted forest. As we prove in the full paper, whenever a rainbow operation is reached in an execution of the augmented algorithm, its guard is satisfied. As a result, the auxiliary painted forest is able to usefully track the standard configuration. Since the original GHS algorithm has no access to the painted forest component of the configuration, and neither it nor the rainbow construction modify the other's state, that there is an isomorphism between the set of executions of the GHS algorithm and that of the augmented algorithm. of the configuration.

Reasoning about GHS executions can thus be performed using the extended configuration of the *augmented* GHS program. The proof has two main parts. The first establishes a list of invariant (safety) properties that hold throughout the execution. The second part consists of a proof of liveness, guaranteeing that the algorithm will achieve its goal: compute the MST and terminate.

4.1 Invariants

Since a painted forest is associated with every configuration of GHS, every node has a well-defined and easily accessible fragment and color in each configuration. The proof makes essential use of the painted forest for stating invariants and for proving liveness properties. We shall now illustrate this use on one of the invariants of the GHS algorithm. Each node (processor) v maintains a number of local variables in the GHS algorithm. These include the variables $\mathtt{FN}_v$, $\mathtt{LN}_v$ and $\mathtt{SN}_v$ that stand for v's local view of its fragment ($\mathtt{FN}_v$), the fragment's level ($\mathtt{LN}_v$), and whether v is actively searching ($\mathtt{SN}_v = \mathsf{Find}$) or not ($\mathtt{SN}_v = \mathsf{Found}$). We use a table that states an invariant for the value of each local variable as a function of the node's color. To illustrate this, the row stating the invariant properties for the level variable $\mathtt{LN}_v$ is copied here in Table 3. In Green and Red nodes v, the variable $\mathtt{LN}_v$ must portray the true level L_v of v's current fragment F_v. For a White node v, which is yet to start its search, the value of $\mathtt{LN}_v$ must be strictly lower than L_v, and if v is a (White) core node then necessarily $\mathtt{LN}_v = L_v - 1$. For a Blue node v, which does not participate in the search on behalf of c_v, the invariant for $\mathtt{LN}_v$ is that $\mathtt{LN}_v \leq L_v$. Since whether a node is Red, White, or Blue is independent of the node's local history, distinctions such as those presented in Table 3 are at best very cumbersome to express without access to the terminology provided by using the painted forest as an auxiliary global history variable.

Table 3. Invariant properties for the $\mathtt{LN}_v$ variable

Color:	slp	wkn	G	R	W	B
Property:	$\mathtt{LN}_v = ?$ (don't care)	$\mathtt{LN}_v = 0$	$\mathtt{LN}_v = L_v$	$\mathtt{LN}_v = L_v$	$\mathtt{LN}_v < L_v \wedge$ $v \in c_v \Rightarrow \mathtt{LN}_v = L_v - 1$	$\mathtt{LN}_v \leq L_v$

Another useful invariant of the GHS algorithm states that the values of the $\mathtt{LN}_v$ variables decrease in a weakly monotone fashion along outbound paths from the core to leaves of a fragment. Formally,

Lemma 4.1. *If $v \in child(w)$ then (i) $\mathtt{LN}_v \leq \mathtt{LN}_w$, and (ii) $\mathtt{LN}_v = \mathtt{LN}_w$ iff $\mathtt{FN}_v = \mathtt{FN}_w$.*

The claim in Lemma 4.1 is expressed in terms of the forest and is independent of the colors in the painted forest. Proving that the lemma is an invariant, however, makes use of colors. The subtlety involved in proving this lemma stems from the fact that fragments are not static: They change when fragments combine. Moreover, Merge and Absorb cause some inbound paths to become outbound paths and vice-versa. To prove the lemma we must demonstrate that the changes that occur do not invalidate the truth of this invariant. Lemma 3.3 implies that a forest edge can change orientation in the rainbow construction only if both of its nodes are Red. By Table 3 we have that $\mathtt{LN}_v = L_v$ holds for Red nodes v. For a forest edge $\{v, w\}$ both nodes are in the same fragment, hence in particular $L_v = L_w$. It follows that whenever an edge flips orientation, the LN variables at both ends are equal, and so the monotonicity property stated in Lemma 4.1 is not foiled.

In addition to the invariants on variable values mentioned above, we present 17 lemmas that state invariant properties of channels and communication. These provide constraints on what messages and what message sequences can appear in channels of different types. They are expressed via extensive use of the painted forest to determine fragment information as well as to distinguish between channel states based on the colors of the nodes at their endpoints. One of the lemmas, for example, implies that a Changeroot message can appear only in an outbound channel connecting two Red nodes. Other invariants can be used to show that only Changeroot messages can appear in such a channel. Special treatment is given to the merging activity at core edges, to the testing performed at Green nodes with a complicating optimization employed by GHS, and to the patterns of Initiate messages along outgoing paths. The Initiate (broadcast) messages are the only type of message of which multiple instances can appear in a single channel.

An extended configuration is called *legal* if it satisfies all of the stated invariants. The invariant theorem states that every reachable configuration is legal. Roughly speaking, it is proven via one joint induction argument. The basis establishes that the invariants are all true in the initial (extended) configuration of every GHS execution, and the inductive step shows that each single atomic step of GHS—consisting of one of the Response procedures of the GHS algorithm that starts out in a legal configuration, yields a legal configuration. The proof of the inductive step is a long case analysis of limited interest. It is a natural candidate for a computer-aided proof.

4.2 Liveness

We now sketch the liveness claims for the proof, formalizing the informal arguments given in [GHS83]. A basic low-level claim, which we omit, shows based on the model of communication that every message in a channel will be repeatedly received at the destination unless it is consumed. One of the ingenious aspects of the GHS algorithm has to do with the fact that many messages are not immediately responded to. Rather, they are returned to the channel to be received again and considered at a later time. As a result, progress in the algorithm is not *a priori* guaranteed. The two scenarios in which such delays may occur are as follows.

Testing: A Green node u that has not yet discovered the value of $\underline{\mathsf{mwe}}(u)$ sends a $\mathtt{Test}(L_\mu, F_\mu)$ message, where $L_\mu = \mathtt{LN}_u$ and $F_\mu = \mathtt{FN}_u$, on its lightest candidate (port) for an external edge. A node v receiving such a message will reply to it provided $\mathtt{LN}_v \geq L_\mu$. The message will be delayed otherwise.

Connecting: A fragment's $\mathsf{mwe}(F_v)$ edge, which is directed from the fragment's node v, is discovered once the $\mathsf{Pend}(v, w)$ condition holds. Given the reliability of communication, we can use the invariants and the rainbow construction to show by induction on the distance between c_v and v that:[6]

Lemma 4.2. $\mathsf{Pend}(v, w) \;\Rightarrow\; \Diamond\big(\mathtt{Connect}(L_v) \in (v, w)\big)$.

Consumption by w of the $\mathtt{Connect}(L_v)$ message, that the lemma guarantees will be in the channel (v, w), is, however, delayed by the GHS algorithm until $\mathtt{LN}_w$ grows enough to satisfy $L_v \leq \mathtt{LN}_w$, and often even until $L_v < \mathtt{LN}_w$.

To establish that progress takes place in spite of this inhibiting behavior of GHS, perhaps the most essential liveness property, which can be proved using Lemma 3.1, Lemma 3.2, and Lemma 3.4, is:

Lemma 4.3. $\big(L_v = \hat{l} > 0 \,\wedge\, c_v = \hat{e}\big) \;\Rightarrow\; \Diamond\big(\mathtt{LN}_v = \hat{l} \,\wedge\, \mathtt{FN}_v = \underline{\hat{e}}\big)$.

Lemma 4.3 states that, for every node of a given fragment, the local variables will eventually display the current level and fragment name. (For a Blue node, this may happen at a time when the node's fragment may already have a different name and level.) The lemma can be used to show that progress is guaranteed in testing on ascending edges and when trying to connect on ascending or strictly ascending edges.

We now consider progress in the search process in a non-singleton fragment. Using invariants on the possible contents of core channels to reason about core nodes, and Lemma 3.2 for non-core nodes, we can show by induction on the distance between v and c_v that:

Lemma 4.4. $\big(\mathsf{col}(v) = W \,\wedge\, c_v = \hat{e}\big) \;\Rightarrow\; \Diamond\big(\mathsf{col}(v) \in \{G, R\} \,\wedge\, c_v = \hat{e}\big)$.

Once Green, a node v searches its untested ports for a $\mathsf{mwe}(v)$, and awaits mwe reports from its children. Using Lemma 4.3 we can show that v's local search will complete successfully if $\mathsf{dmwe}(v)$ is ascending. Moreover, the node v will eventually turn Red if $\mathsf{dmwe}(w)$ is ascending for every $w \in T_v$.

[6] We use the temporal logic $\Diamond$ operator for "*eventually*" [MP95]. In addition, all of our liveness claims are implicitly universally quantified—they apply to all configurations $\mathcal{C}$ and all nodes v, w, etc.

Let us denote by *WakeNo* and *EdgeNo* the number of non-sleeping nodes, and the number of forest edges, respectively. Clearly, $0 \leq \mathit{WakeNo} \leq n$ and $0 \leq \mathit{EdgeNo} \leq n-1$. Moreover, $\mathit{EdgeNo} < n-1$ iff there is more than one fragment in the forest, so that at least one more MST edge needs to be added. Think of a fragment as being *active* if it is not a sleeping singleton node. Let $L_{min}(\mathcal{C})$ be the minimal level of any active fragment in $\mathcal{C}$. In [GHS83] the authors argue that, if $\mathit{EdgeNo} < n-1$ then one fragment of level L_{min} will be guaranteed to succeed in adding an MST edge. We now present a slightly more rigorous argument.

For a fragment of level $L_{min}(\mathcal{C})$, every external edge is either ascending, or connected to a sleeping singleton. It follows that in a fragment of minimal level, every local search at a Green node will either succeed, or will wake up a sleeping singleton. Formally, we can show:

Lemma 4.5. $L_v = L_{min} \wedge \mathsf{col}(v) = G \quad \Rightarrow$
$\Diamond\big(\mathit{WakeNo\ will\ strictly\ increase} \ \vee\ (\mathsf{col}(v) = R)\big).$

Applying Lemma 4.5 by induction on the structure of F_v we obtain that minimal-level fragments complete their unless new nodes are awaken.

Lemma 4.6. $L_v = L_{min} \wedge \underline{\mathsf{mwe}}(F_v) \neq \infty \quad \Rightarrow$
$\Diamond\big(\mathit{WakeNo\ will\ strictly\ increase} \ \vee\ \mathsf{Pend}(\mathsf{dmwe}(F_v))\big).$

Combining this with Lemmas 4.2 and 4.3, we can establish:

Lemma 4.7. $\mathit{EdgeNo} < n-1 \quad \Rightarrow$
$\Diamond(\mathit{WakeNo\ will\ strictly\ increase} \ \vee\ \mathit{EdgeNo\ will\ strictly\ increase}).$

Sketch of proof: The problem specification assumes that at some point at least one node wakes up. Thus, initially we have $\mathit{WakeNo} = 0$ and are guaranteed that *WakeNo* will increase. Ultimately, at most n nodes can be woken up overall. Thus, in every execution there must be some point after which no sleeping node ever wakes up. Let $\mathcal{C}$ be a such a configuration with $\mathit{EdgeNo} < n-1$ (so there are at least two fragments in $\mathcal{C}$). By Corollary 4.6 all searches in fragments of level L_{min} are guaranteed to complete, and by Lemma 4.2 their dmwe's will contain `Connect` messages. If any of these dmwe's point to fragments of strictly higher levels, then we have by Lemma 4.3 and the GHS behavior that an Absorb will result, and *EdgeNo* will increase. Otherwise, all of these dmwe's point to fragments of level L_{min}. Because edge weights are distinct, one of these mwe's is the lightest of them all. This edge must be the mwe of both fragments it is incident on. A Merge will thus take place on this edge, and *EdgeNo* will increase.

Since $0 \leq \mathit{WakeNo} + \mathit{EdgeNo} \leq 2n-1$ with equality only when the MST is complete, we immediately obtain:

Lemma 4.8. *In every execution of the GHS algorithm, the forest eventually consists of the MST.*

Once the forest consists of a single tree, we have that $\mathit{WakeNo} = n$ and can use Lemma 4.5 to prove that the search will complete with both core nodes Red. Further reasoning using the invariants then shows that at that point all ports are properly marked, and all messages remaining in the channels are consumed without modifying the port markings, and so eventually all channels are empty. We thus obtain our main theorem:

Theorem 4.1. *Every run of the GHS algorithm reaches a state satisfying the specification of the MST problem given by conditions ST1 and ST2 in Definition 4.1.*

5 Conclusions

It is no accident that the GHS algorithm has become a notorious challenge for verification of distributed protocols. The considerable nondeterminism inherent in the scheduler's timing choices results in the runs of GHS lacking a natural phase structure. A fragment may concurrently be engaged in many activities involving outbound broadcasts, inbound convergecasts, and local testing. Moreover, while all of these are taking place, the structure of the fragment may change as a result of other fragments being absorbed into it.

The main novelty of our proof comes from the introduction of the intermediate-level of abstraction that we called the rainbow construction. Executions of the rainbow construction refine the standard high-level description of GHS in terms of Merge and Absorb at the graph level, and they are refined by executions of the detailed GHS algorithm. We do not, however, prove refinement mappings between the three levels. Rather, we prove structural properties at the abstract levels, and use configurations of the rainbow construction as global auxiliary history variables when reasoning about the detailed GHS algorithm. The auxiliary configurations of the rainbow construction enable cleaner and simpler statement of invariance properties, and facilitates the reasoning about progress guarantees in executions of GHS. Structural properties such as the characterization of color patterns along outgoing path in Lemma 3.1 considerably reduce the amount of case analysis required in the proofs.

We believe that the rainbow construction as well as some of our analysis can help in simplifying the proofs of GHS in other frameworks, such as the I/O automata approach of [W88], and the Petri-net based proof of [S01]. There is also hope that our decomposition of the problem will enable an efficient mechanized proof for GHS.[7]

Earlier attempts at proving the GHS algorithm have led some of the researchers involved to conclude that, in the end, there is no alternative to a large invariance proof in this case, and such a proof is necessarily tedious and boring. This paper suggests that generating a manageable invariance proof of GHS is itself nontrivial. In our proof, the use of the rainbow construction made this task tractable. The fact that our liveness proof is a direct formalization of the argument described in [GHS83] suggests that our correctness proof is "natural."

[7] Hesselink [Hes99] set out to produce such a proof, and ended up proving correctness of a related MST algorithm.

References

[Bol79] B. Bollobás: Graph theory, An introductory course. *Graduate Texts in Mathematics* 63: 1-180 (1979).

[Bor26] O. Borůvka: O jistém problému minimálním. *Práce Mor. Prírodoved Spol. v Brne (Acta Societ. Scient. Natur. Moravicae)* 3: 37-58 (1926) *(Czech language)*.

[C88] C-T. Chou: A Bug in the Distributed Minimum Spanning Tree Algorithms of Gallager, Humblet, and Spira, unpublished manuscript, 1988.

[CG88] C-T. Chou and E. Gafni: Understanding and Verifying Distributed Algorithms Using Stratified Decomposition. PODC 1988: 44-65.

[GHS83] R. G. Gallager, P. A. Humblet and P. M. Spira: A Distributed Algorithm for Minimum-Weight Spanning Trees. *TOPLAS* 5(1): 66-77 (1983).

[Har69] F. Harary: *Graph Theory* Addison-Wesley 1969.

[Hes99] W. H. Hesselink: The Verified Incremental Design of a Distributed Spanning Tree Algorithm: Extended Abstract. *Formal Aspects of Computing*, 11(1): 45-55 (1999).

[JZ92] W. Janssen and J. Zwiers: From Sequential Layers to Distributed Processes: Deriving a Distributed Minimum Weight Spanning Tree Algorithm (Extended Abstract). PODC 1992: 215-227.

[Kr56] R. C. Prim: On the shortest subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc. 7* (1956), 48-50.

[Ly95] N. A. Lynch: *Distributed Algorithms* Morgan Kaufmann 1996.

[MP95] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems ⊙ Safety ⊙*, Springer Verlag 1995.

[Pr57] R. C. Prim: Shortest connection networks and some generalizations. *Bell System Technical Journal* 36: 1389-1401, (1957).

[SdR87] F. A. Stomp and W. P. de Roever: A Correctness Proof of a Distributed Minimum-Weight Spanning Tree Algorithm (extended abstract). ICDCS 1987, 440-447.

[S01] S. Peuker: *Halbordnungsbasierte Verfeinerung zur Verifikation verteilter Algorithmen.* Ph.D. thesis, Humboldt University of Berlin, April 2001.

[S02] S. Peuker: Transition Refinement for Deriving a Distributed Minimum Weight Spanning Tree Algorithm, *Proc. 23*$^{\text{rd}}$ *Intl. Conf. Ap. & Th. Petri Nets (ICATPN 2002)*, 2002, 374-393.

[W88] J. L. Welch: *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms.* Ph.D. thesis, MIT June 1988.

[WLL88] J. L. Welch, L. Lamport and N. A. Lynch: A Lattice-Structured Proof Technique Applied to a Minimum Spanning Tree Algorithm (Extended Abstract). PODC 1988: 28-43.

A Knowledge-Based Analysis of Global Function Computation

Joseph Y. Halpern and Sabina Petride

Department of Computer Science
Cornell University
Ithaca, NY 14853
{halpern, petride}@cs.cornell.edu

Abstract. Consider a distributed system N in which each agent has an input value and each communication link has a weight. Given a global function, that is, a function f whose value depends on the whole network, the goal is for every agent to eventually compute the value $f(N)$. We call this problem *global function computation*. Various solutions for instances of this problem, such as Boolean function computation, leader election, (minimum) spanning tree construction, and network determination, have been proposed, each under particular assumptions about what processors know about the system and how this knowledge can be acquired. We give a necessary and sufficient condition for the problem to be solvable that generalizes a number of well-known results [3, 28, 29]. We then provide a *knowledge-based (kb) program* (like those of Fagin, Halpern, Moses, and Vardi [8, 9]) that solves global function computation whenever possible. Finally, we improve the message overhead inherent in our initial kb program by giving a *counterfactual belief-based program* [15] that also solves the global function computation whenever possible, but where agents send messages only when they believe it is necessary to do so. The latter program is shown to be implemented by a number of well-known algorithms for solving leader election.

1 Introduction

Consider a distributed system N in which each agent has an input value and each communication link has a weight. Given a global function, that is, a function f whose value depends on the whole network, the goal is for every agent to eventually compute the value $f(N)$. We call this problem *global function computation*. Many distributed protocols involve computing some global function of the network. This problem is typically straightforward if the network is known. For example, if the goal is to compute the spanning tree of the network, one can simply apply one of the well-known algorithms proposed by Kruskal or Prim. However, in a distributed setting, agents may have only local information, which makes the problem more difficult. For example, the algorithm proposed by Gallager, Humblet and Spira [11] is known for its complexity.[1] Moreover,

[1] Gallager, Humblet, and Spira's algorithm does not actually solve the minimum spanning tree as we have defined it, since agents do not compute the minimum spanning tree, but only learn relevant information about it, such as which of its edges lead in the direction of the root.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 136–150, 2006.

the algorithm does not work for all networks, although it is guaranteed to work correctly when agents have distinct inputs and no two edges have identical weights.

Computing shortest paths between nodes in a network is another instance of global function computation that has been studied extensively. [10, 4]. The well-known *leader election problem* [21] can also be viewed as an instance of global computation in all systems where agents have distinct inputs: the leader is the agent with the largest (or smallest) input. The difficulty in solving global function computation depends on what processors know. For example, when processors know their identifiers (names) and all ids are unique, several solutions for the leader election problem have been proposed, both in the synchronous and asynchronous settings [6, 19, 25]. On the other hand, Angluin [1], and Johnson and Schneider [18] proved that it is impossible to deterministically elect a leader if agents may share names. In a similar vein, Attiya, Snir and Warmuth [3] prove that there is no deterministic algorithm that computes a non-constant Boolean global function in a ring of unknown and arbitrarily large size if agents' names are not necessarily unique. Attiya, Gorbach, and Moran [2] characterize what can be computed in what they call *totally anonymous shared memory systems*, where access to shared memory is anonymous.

We aim to better understand what agents need to know to compute a global function. We do this using the framework of *knowledge-based (kb) programs*, proposed by Fagin, Halpern, Moses and Vardi [8, 9]. Intuitively, in a kb program, an agent's actions may depend on his knowledge. To say that the agent with identity i knows some fact φ we simply write $K_i\varphi$. For example, if agent i sends a message msg to agent j only if he does not know that j already has the message, then the agent is following a kb program that can be written as

$$\textbf{if } K_i(has_j(msg)) \textbf{ then } \mathsf{skip} \textbf{ else } send(msg).$$

Knowledge-based programs abstract away from particular details of implementation and generalize classes of standard programs. They provide a high-level framework for the design and specification of distributed protocols. They have been applied to a number of problems, such as *atomic commitment* [14], *distributed commitment* [22], Byzantine agreement [7, 16], sequence transmission [17], and analyzing the TCP protocol [27].

We first characterize when global function computation is solvable, i.e., for which networks N and global functions f agents can eventually learn $f(N)$. As we said earlier, whether or not agents can learn $f(N)$ depends on what they initially know about N. We model what agents initially know as a set $\mathcal{N}$ of networks; the intuition is that $\mathcal{N}$ is the set of all networks such that it is common knowledge that N belongs to $\mathcal{N}$. For example, if it is commonly known that the network is a ring, $\mathcal{N}$ is the set of all rings; this corresponds to the setting considered by Attiya, Snir and Warmuth [3]. If, in addition, the size n of N is common knowledge, then $\mathcal{N}$ is the (smaller) set of all rings of size n. Yamashita and Kameda [28] focus on three different types of sets $\mathcal{N}$: (1) for a given n, the set of all networks of size n, (2) for a fixed d, the set of all networks of diameter at most d, and (3) for a graph G, the set of networks whose underlying graph is G, for all possible labelings of nodes and edges. In general, the more that is initially known, the smaller $\mathcal{N}$ is. Our problem can be rephrased as follows: given N and f, for which sets $\mathcal{N}$ is it possible for all agents in N to eventually learn $f(N)$?

For simplicity, we assume that the network is finite and connected, that communication is reliable, and that no agents fail. Consider the following simple protocol, run by each agent in the network: agents start by sending what they initially know to all of their neighbors; agents wait until they receive information from all their neighbors; and then agents transmit all they know on all outgoing links. This is a *full-information protocol*, since agents send to their neighbors everything they know. Clearly with the full-information protocol all agents will eventually know all available information about the network. Intuitively, if $f(N)$ can be computed at all, then it can be computed when agents run this full-information protocol. However, there are cases when this protocol fails; no matter how long agents run the protocol, they will never learn $f(N)$. This can happen because

1. although the agents actually have all the information they could possibly get, and this information suffices to compute the value of f, the agents do not know this;
2. although the agents have all the information they could possibly get (and perhaps even know this), the information does not suffice to compute the function value.

In Section 2, we illustrate these situations with simple examples. We show that there is a natural way of capturing what agents know in terms of *bisimilarity relations* [23], and use bisimilarity to characterize exactly when global function computation is solvable. We show that this characterization provides a significant generalization of results of Attiya, Snir, and Warmuth [3] and Yamashita and Kameda [29].

We then show that the simple program where each agent just forwards all the new information it obtains about the network solves the global function computation problem whenever possible. It is perhaps obvious that, if anything works at all, this program works. We show that the program terminates with each agent knowing the global function value iff the condition that we have identified holds.

Our program, while correct, is typically not optimal in terms of the number of messages sent. Generally speaking, the problem is that agents may send information to agents who already know it or will get it via another route. For example, consider an oriented ring. A simple strategy of always sending information to the right is just as effective as sending information in both directions. Thus, roughly speaking, we want to change the program so that an agent sends whatever information he learns to a neighbor only if he does not know that the neighbor will eventually learn it anyway.

Since agents decide which actions to perform based on what they know, this will be a kb program. While the intuition behind this kb program is quite straightforward, there are subtleties involved in formalizing it. One problem is that, in describing kb programs, it has been assumed that names are commonly known. However, if the network size is unknown, then the names of all the agents in the network cannot be commonly known. Things get even more complicated if we assume that identifiers are not unique. For example, if identifiers are not unique, it does not make sense to write "agent i knows φ"; $K_i\varphi$ is not well defined if more than one agent can have the id i.

We deal with these problems using techniques introduced by Grove and Halpern [12, 13]. Observe that it makes perfect sense to talk about each agent acting based on his own knowledge by saying "if *I* know φ, then ...". I here represents the name each agent uses to refer to himself. This deals with self-reference; by using relative names appropriately, we can also handle the problem of how an agent refers to other agents.

A second problem arises in expressing the fact that an agent should send information to a neighbor only if the neighbor will not eventually learn it anyway. As shown by Halpern and Moses [15] (HM from now on), the most obvious way of expressing it does not work; to capture this intuition correctly we must use *counterfactuals*. These are statements of the form $\varphi > \psi$, which are read "if φ then ψ", but the "if ... then" is not treated as a standard material implication. In particular, the formula is not necessarily true if φ is false. In Section 3.1, we provide a kb program that uses counterfactuals which solves the global function computation problem whenever possible, while considerably reducing communication overhead.

As a reality check, for the special case of leader election in networks with distinct ids, we show in Section 4 that the kb program is essentially implemented by the protocols of Lann, Chang and Roberts [19, 6], and Peterson [25], which all work in rings (under slightly different assumptions), and by the optimal flooding protocol [21] in networks of bounded diameter. Thus, the kb program with counterfactuals shows the underlying commonality of all these programs and captures the key intuition behind their design.

The rest of this paper is organized as follows. In Section 2, we give our characterization of when global function computation is possible. In Section 3 we describe the kb program for global function computation, and show how to optimize it so as to minimize messages. In Section 4, we show that the program essentially implements some standard solutions to leader election in a ring. For space reasons, we defer the detailed formal definitions and the proofs of results to the full paper.

2 Characterizing When Global Function Computation is Solvable

We model a network as a directed, simple (no self-loops), connected, finite graph, where both nodes and edges are labeled. Each node represents an agent; its label is the agent's input, possibly together with the agent's name (identifier). Edges represent communication links; edge labels usually denote the cost of message transmission along links. Communication is reliable, meaning that every message sent is eventually delivered and no messages are duplicated or corrupted.

We assume that initially agents know their *local information*, i.e., their own input value, the number of outgoing links, and the weights associated with these links. However, agents do not necessarily know the weights on non-local edges, or any topological characteristics of the network, such as size, upper bound on the diameter, or the underlying graph. Additionally, agents may not know the identity of the agents they can directly communicate with, or if they share their names with other agents. In order to uniquely identify agents in a network N of size n, we label agents with "external names" $1, \ldots, n$. Agents do not necessarily know these external names; we use them for our convenience when reasoning about the system. In particular, we assume that the global function f does not depend on these external names; $f(N) = f(N')$ for any two networks N and N' that differ only in the way that nodes are labeled.

Throughout the paper we use the following notation: We write $V(N)$ for the set of agents in N and $E(N)$ for the set of edges. For each $i \in V(N)$, let $Out_N(i)$ be the set of i's neighbors on outgoing links, so that $Out_N(i) = \{j \in V(N) \mid (i,j) \in E(N)\}$; let $In_N(i)$ be the set of i's neighbors on incoming links, so that $In_N(i) = \{j \in$

$V(N) \mid (j,i) \in E(N))\}$; let $in_N(i)$ denote i's input value. Finally, if e is an edge in $E(N)$, let $w_N(e)$ denote e's label.

We want to understand, for a given network N and global function f, when it is possible for agents to eventually know $f(N)$. This depends on what agents know about N. As mentioned in the introduction, the general (and unstated) assumption in the literature is that, besides their local information, whatever agents know initially about the network is *common knowledge*. We start our analysis by making the same assumption, and characterize the initial common knowledge as a set $\mathcal{N}$ of networks.

In this section, we assume that agents are following a full-information protocol. We think of the protocol as proceeding in *rounds*: in each round agents send to all neighbors messages describing all the information they have; messages are stamped with the round number; round k for agent i starts after he has received all round $k-1$ messages from his neighbors (since message delivery is reliable, this is guaranteed to happen). The round-based version of the full-information protocol makes sense both in synchronous and asynchronous settings, and for any assumptions about the order in which messages are delivered.

Intuitively, the full-information protocol reduces uncertainty. For example, suppose that $\mathcal{N}$ consists of all unidirectional 3-node rings, and let N be a three node ring in which agents have inputs a, b, and c, and all edges have the same weight w. Let i be the external name of the agent with input a. Initially, i considers possible all 3-nodes rings in which the weight on his outgoing edge is w and his input is a. After the first round, i learns from his incoming neighbor, who has external name j, that j's incoming edge also has weight w, and that j has input c. Agent j learns in the first round that his incoming neighbor has input b and that his incoming edge also has weight w. Agent j communicates this information to i in round 2. At the end of round 2, i knows everything about the network N, as do the other two agents. Moreover, he knows exactly what the network is. But this depends on the fact that i knows that the ring has size 3.

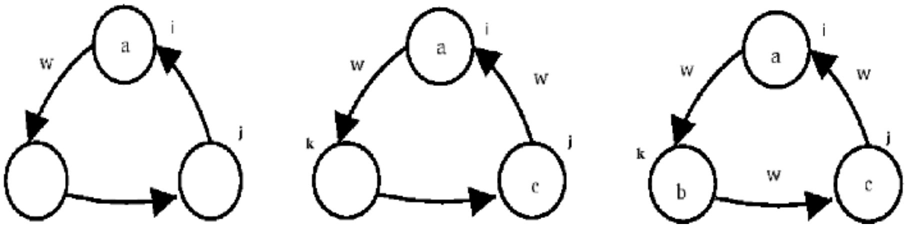


Fig. 1. How i's information changes with the full-information protocol

Now consider the same network N, but suppose that agents do not know the ring size, i.e., $\mathcal{N}$ is the set of all unidirectional rings, of all possible sizes and for all input and weight distributions. Again, at the end of round 2, agent i has all the information that he could possibly get, as do the other two agents. However, at no point are agents able to distinguish the network N from a 6-node ring N' in which agents look just like the agents on the 3-node ring (see Figure 2). Consider the pair of agents i in N and i' in N'. It is easy to check that these agents get exactly the same messages in every round of the full-information protocol. Thus, they have no way of distinguishing which is the true situation. If the function f has different values on N and N', then the agents cannot compute $f(N)$. On the other hand, if $\mathcal{N}$ consists only of networks where inputs are distinct, then i realizes at the end of round 2 that he must be k's neighbor, and then he knows the network configuration.

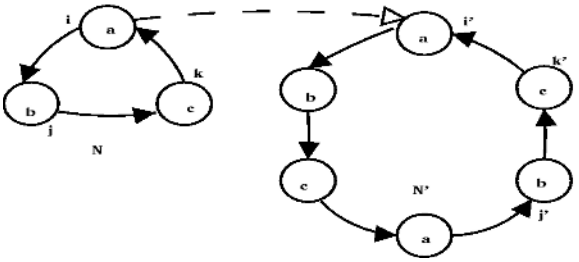


Fig. 2. Two indistinguishable networks

We want to characterize when agent i in network N thinks he could be agent i' in network N'. Intuitively, at round k, i thinks it possible that he could be i' if there is a bijection μ that maps i's incoming neighbors to i''s incoming neighbors such that, at the previous round $k-1$, each incoming neighbor j of i thought that he could be $\mu(j)$.

Definition 1. *Given networks N and N' and agents $i \in V(N)$ and $i' \in V(N')$, i and i' are* 0-bisimilar, *written $(N,i) \sim_0 (N',i')$, iff*

- *$in_N(i) = in_{N'}(i')$;*
- *there is a bijection $f^{out} : Out_N(i) \longrightarrow Out_{N'}(i')$ that preserves edge-labels; that is, for all $j \in Out_N(i)$, we have $w_N(i,j) = w_{N'}(i', f^{out}(j))$.*

For $k > 0$, i and i' are k-bisimilar, *written $(N,i) \sim_k (N',i')$, iff*

- *$(N,i) \sim_0 (N',i')$, and*
- *there is a bijection $f^{in} : In_N(i) \longrightarrow In_{N'}(i')$ such that for all $j \in In_N(i)$*
  - *$w_N(j,i) = w_{N'}(f^{in}(j), i')$,*
  - *the (j,i) edge is bidirectional iff the $(f^{in}(j), i')$ edge is bidirectional, and*
  - *$(N,j) \sim_{k-1} (N', f^{in}(j))$.*

Note that $\sim_k$ is an equivalence relation on the set of pairs (N,i) with $i \in V(N)$, and that $\sim_{k+1}$ is a refinement of $\sim_k$.

The following lemma relates bisimilarity and the full-information protocol:

Lemma 1. *The following are equivalent:*

(a) $(N,i) \sim_k (N',i')$.

(b) Agents $i \in V(N)$ and $i' \in V(N')$ have the same initial local information and receive the same messages in each of the first k rounds of the full-information protocol.

(c) If the system is synchronous, then i and i' have the same initial local information and receive the same messages in each of the first k rounds of every deterministic protocol.

Intuitively, if the function f can be computed on N, then it can be computed using a full-information protocol. The value of f can be computed when f takes on the same value at all networks that the agents consider possible. The round at which this happens may depend on the network N, the function f, and what it is initially known. Moreover, if it does not happen, then f is not computable. Using Lemma 1, we can characterize if and when it happens.

Theorem 1. *The global function f can be computed on networks in $\mathcal{N}$ iff, for all networks $N \in \mathcal{N}$, there exists a constant $k_{\mathcal{N},N,f}$, such that, for all networks $N' \in \mathcal{N}$, all $i \in V(N)$, and all $i' \in V(N')$, if $(N,i) \sim_{k_{\mathcal{N},\ ,}} (N',i')$ then $f(N') = f(N)$.*

Proof: First suppose that the condition in the statement of the theorem holds. At the beginning of each round k, each agent i in the network proceeds as follows. If i received the value of f in the previous round, then i forwards the value to all of its neighbors and terminates; otherwise, i computes f's value on all the networks N' such that there exists an i' such that agent i' would have received the same messages in the first $k-1$ rounds in network N' as i actually received. (By Lemma 1, these are just the pairs (N',i') such that $(N',i') \sim_{k-1} (N,i)$.) If all the values are equal, then i sends the value to all his neighbors and terminates; otherwise, i sends whatever new information he has received about the network to all his neighbors.

Let k_i be the first round with the property that for all $N' \in \mathcal{N}$ and i' in N', if $(N,i) \sim_k (N',i')$, then $f(N') = f(N)$. (By assumption, such a k_i exists and it is at most $k_{\mathcal{N},N,f}$.) It is easy to see that, by round k_i, i learns the value of $f(N)$, since either i gets the same messages that it gets in the full-information protocol up to round k_i or it gets the function value. Thus, i terminates by the end of round $k_i + 1$ at the latest, after sending the value of f, and the protocol terminates in at most $k_{\mathcal{N},N,f} + 1$ rounds. Clearly all agents learn $f(N)$ according to this protocol.

Now suppose that the condition in the theorem does not hold and, by way of contradiction, that the value of f can be computed by some protocol P on all the networks in $\mathcal{N}$. There must exist some network N for which the condition in the theorem fails. Consider a run where all messages are delivered synchronously. There must be some round k such that all agents in N have computed the function value by round k. Since the condition fails, there must exist a network $N' \in \mathcal{N}$ and agents $i \in V(N)$ and $i' \in V(N')$ such that $(N,i) \sim_k (N',i')$ and $f(N) \neq f(N')$. By Lemma 1, i and i' have the same initial information and receive the same messages in the first k rounds of protocol P. Thus, they must output the same value for the function at round k. But since $f(N) \neq f(N')$, one of these answers must be wrong, contradicting our assumption that P computes the value of f in all networks in $\mathcal{N}$. ∎

Intuitively, $k_{\mathcal{N},N,f}$ is a round at which each agent i knows that f takes on the same value at all the networks i considers possible at that round. Since we are implicitly assuming that agents do not forget, the set of networks that agent i considers possible never grows. Thus, if f takes on the same value at all the networks that agent i considers possible at round k, then f will take on the same value at all networks that i considers possible at round $k' > k$, so every agent knows the value of $f(N)$ in round $k_{\mathcal{N},N,f}$. In some cases, we can provide a useful upper bound on $k_{\mathcal{N},N,f}$. For example, if $\mathcal{N}$ consists only of networks with distinct identifiers, or, more generally, of networks in which no two agents are *locally* the same, i.e., $(N, i) \not\sim_0 (N, j)$ for all $i \neq j$, then we can take $k_{\mathcal{N},N,f} = diam(N) + 1$, where $diam(N)$ is the diameter of N.

Theorem 2. *If initially it is common knowledge that no two agents are locally the same, then all global functions can be computed; indeed, we can take* $k_{\mathcal{N},N,f} = diam(N) + 1$.

Attiya, Snir, and Warmuth [3] prove an analogue of Lemma 1 in their setting (where all networks are rings) and use it to prove a number of impossibility results. In our language, these impossibility results all show that there does not exist a k such that $(N, i) \sim_k (N', i')$ implies $f(N) = f(N')$ for the functions f of interest, and thus are instances of Theorem 1.[2]

Yamashita and Kameda characterize when global functions can be computed in undirected networks (which have no weights associated with the edges), assuming that an upper bound on the size of the network is known. They define a notion of *view* and show that two agents have the same information whenever their views are *similar* in a precise technical sense; $f(N)$ is computable iff for all networks N' such that agents in N and N' have similar views, $f(N') = f(N)$. Their notion of similarity is essentially our notion of bisimilarity restricted to undirected networks with no edge labels. Thus, their result is a special case of Theorem 1 for the case that $\mathcal{N}$ consists of undirected networks with no edge labels of size at most n^* for some fixed constant n^*; they show that $k_{\mathcal{N},N,f}$ can be taken to be n^* in that case. Not only does our result generalize theirs, but our characterization is arguably much cleaner.

Theorem 2 sheds light on why the well-known protocol for minimum spanning tree construction proposed by Gallager, Humblet, and Spira [11] can deal both with systems with distinct ids (provided that there is a commonly-known ordering on ids) and for networks with identical ids but distinct edge-weights. These are just instances of situations where it is common knowledge that no two agents are locally the same.

3 Standard and kb Programs for Global Function Computation

3.1 Dealing with Shared Names

A *knowledge-based (kb) program* Pg_{kb} has the form

$$\begin{array}{l} \textbf{if } t_1 \wedge k_1 \textbf{ then } \mathsf{act}_1 \\ \textbf{if } t_2 \wedge k_2 \textbf{ then } \mathsf{act}_2 \\ \ldots, \end{array}$$

[2] We remark that Attiya, Snir, and Warmuth allow their global functions to depend on external names given to agents in the network. This essentially amounts to assuming that the agent's names are part of their input.

where the t_js are standard tests (not involving knowledge or beliefs, but possibly involving temporal operators such as $\Diamond$ and counterfactuals), the k_js are knowledge tests (that could also involve belief, as we shall see), and the act_js are actions. The intended interpretation is that agent i runs this program forever. At each point in time, i nondeterministically executes one of the actions act_j such that the test $t_j \wedge k_j$ is satisfied; if no such action exists, i does nothing. We sometime use obvious abbreviations like **if** ... **then** ... **else**. A *standard* program is one with no knowledge tests.

Even in a standard program, there are issues of naming if we work in networks where the names of agents are not common knowledge (see [24, 13]). Following Grove and Halpern [12, 13] (GH from now on), we distinguish between agents and their names. We assume that programs mention only names, not agents (since in general the programmer will have access only to the names, which can be viewed as denoting roles). We use $\mathbf{N}$ to denote the set of all possible names and assume that one of the names is I. In the semantics, we associate with each name the agent who has that name. With each *run* (or execution) of the program, we associate the set of agents that exist in that run. For simplicity, we assume that the set of agents is constant over the run; that is, we are not allowing agents to enter the system or leave the system. However, different sets of agent may be associated with different runs. We assume that each agent has a way of naming his neighbors, and gives each of his neighbors different names. However, two different agents may use the same name for different neighbors. For example, in a ring, each agent may name his neighbors L and R; in an arbitrary network, an agent whose outdegree is d may refer to his outgoing neighbors as 1, 2, ..., d. We allow actions in a program to depend on names, so the meaning of an action may depend on which agent is running it. For example, in our program for global function computation, if i uses name $\mathbf{n}$ to refer to his neighbor j, we write i's action of sending message msg to j as $send_{\mathbf{n}}(msg)$. Similarly, if A is a set of names, then we take $send_A(msg)$ to be the action of sending msg to each of the agents in A (and not sending anything to any other agents). For convenience, let $\mathbf{Nbr}$ denote the neighbors of an agent, so that $send_{\mathbf{Nbr}}(msg)$ is the action of sending msg to all of an agent's neighbors.

We assume that message delivery is handled by the channel (and is not under the control of the agents). In the program, we use a primitive proposition $some\_new\_info$ that we interpret as true for agent i iff i has received some new information; in our setting, that means that i has learned about another agent in the network and his input, has learned the weight labeling some edges, or has learned that there are no further agents in the network. (Note that in the latter case, i can also compute the function value. For example, in doing leader election on a unidirectional ring, if i gets its id back after sending it around the network, then i knows that it has heard from all agents in the network, and can then compute which agent has the highest id.) Note that $some\_new\_info$ is a proposition whose truth is relative to an agent. As already pointed out by GH, once we work in a setting with relative names, then both propositions and names need to be interpreted relative to an agent. In the program, the action $send_{\mathbf{n}}(new\_info)$ has the effect of i sending $\mathbf{n}$ whatever new information i learned.

With this background, we can describe the program for global function computation, which we call Pg^{GC}; each agent runs the program

$$\textbf{if } some\_new\_info \textbf{ then } send_{\mathbf{Nbr}}(new\_info); receive,$$

where the *receive* action updates the agent's state by receiving any messages that are waiting to be delivered. Next we prove that Pg^{GC} is correct.

Theorem 3. *Pg^{GC} solves the global function computation problem whenever possible. That is, if $\mathcal{N}$ and f satisfy the condition in Theorem 1, then with Pg^{GC} eventually all agents know the function value; otherwise, no agent ever knows the function value.*

As written, Pg^{GC} does not terminate; however, we can easily modify it so that it terminates if agents learn the function value. (They will send at most one message after learning the function value.)

3.2 Improving Message Overhead

While sending only the new information that an agent learns at each step reduces the size of messages, it does not preclude sending unnecessary messages. One way of reducing communication is to have agent i not send information to the agent he names $\mathbf{n}$ if he *knows* that $\mathbf{n}$ already *knows* the information.

To capture this, assume first that there is a modal operator $K_{\mathbf{n}}$ in the language for each name $\mathbf{n} \in \mathbf{N}$. When interpreted relative to agent i, $K_{\mathbf{n}}\varphi$ is read as "the agent i named $\mathbf{n}$ knows fact φ". Let $\mathit{cont}(\mathit{new\_info})$ be a primitive proposition that characterizes the content of the message $\mathit{new\_info}$. For example, suppose that N is a unidirectional ring, and $\mathit{new\_info}$ says that i's left neighbor has input value v_1. Then $\mathit{cont}(\mathit{new\_info})$ is true at all points where i's left neighbor has input value v_1. (Note that $\mathit{cont}(\mathit{new\_info})$ is a proposition whose truth is relative to an agent.) Thus, it seems that the following kb program should solve the global function computation problem, while decreasing the number of messages:

```
if $\mathit{some\_new\_info}$ then
  for each $\mathit{nonempty\ subset}$ A $\mathit{of\ agents}$ do
  if $A = \{\mathbf{n} : \neg K_I K_{\mathbf{n}}(\mathit{cont}(\mathit{new\_info}))\}$ then $\mathit{send}_A(\mathit{new\_info})$; $\mathit{receive}$.
```

While this essentially works, there are some subtleties in interpreting this kb program. As observed by GH, once we allow relative names, we must be careful about scoping. For example, suppose that, in an oriented ring, i's left neighbor is j and j's left neighbor is k. What does a formula such as $K_I K_L(\mathit{left\_input} = 3)$ mean when it is interpreted relative to agent i? Does it mean that i knows that j knows that k's input is 3, or does it mean that i knows that j knows that j's input is 3? That is, do we interpret the "left" in *left_input* relative to i or relative to i's left neighbor j? Similarly, to which agent does the second L in $K_I K_L K_L \varphi$ refer? That, of course, depends on the application. Using a first-order logic of naming, as in [12], allows us to distinguish the two interpretations readily. In a propositional logic, we cannot do this. In the propositional logic, Grove and Halpern [13] assumed *innermost scoping*, so that the *left* in *left_input* and the second L in $K_I K_L K_L \varphi$ are interpreted relative to the "current" agent considered when they are evaluated (which is j). As we will see, in our interpretation, we want to interpret it relative to I (in this case, i). As we will see, in a formula such as $K_I K_{\mathbf{n}}$ $\mathit{cont}(\mathit{new\_info})$, we want to interpret $\mathit{cont}(\mathit{new\_info})$ relative to "I", the agent i that sends the message, not with respect to the agent j that is the interpretation of $\mathbf{n}$. To capture this, we add limited quantification over names to the language. In particular, we allow formulas of the form $\exists \mathbf{n}'(\mathit{Calls}(\mathbf{n}, I, \mathbf{n}') \wedge K_{\mathbf{n}}(\mathbf{n}'\text{'s}\varphi))$, which is interpreted as "there exists a name $\mathbf{n}'$ such that the agent I names $\mathbf{n}$ gives name $\mathbf{n}'$ to the agent that currently has name I and $\mathbf{n}$ knows that φ interpreted relative to $\mathbf{n}'$ holds". Then, instead of writing $K_I K_{\mathbf{n}} \mathit{cont}(\mathit{new\_info})$, we write $K_I(\exists \mathbf{n}'(\mathit{Calls}(\mathbf{n}, I, \mathbf{n}') \wedge K_{\mathbf{n}}(\mathbf{n}'\text{'s}\mathit{cont}(\mathit{new\_info}))))$.

We can further reduce message complexity by not sending information not only if the recipient of the message already knows the information, but also if he will *eventually* know the information. It seems relatively straightforward to capture this: we simply add a $\Diamond$ operator and replace the test $K_I(\exists \mathbf{n}'(\mathit{Calls}(\mathbf{n}, I, \mathbf{n}') \wedge K_{\mathbf{n}}(\mathbf{n}'\text{'s}\mathit{cont}(\mathit{new\_info}))))$ by $K_I \Diamond(\exists \mathbf{n}'(\mathit{Calls}(\mathbf{n}, I, \mathbf{n}') \wedge K_{\mathbf{n}}(\mathbf{n}'\text{'s}\mathit{cont}(\mathit{new\_info}))))$. Unfortunately, as already observed by HM, this modification will not work. To see why, we need to give some background on the semantics of kb programs.

A *protocol* for agent i is a function from the states of agent i to actions. We can associate with every *joint protocol* P (that is, a tuple consisting of a protocol for each agent) the *system* $\mathcal{R}(P)$ that *represents* P, which consists of the runs of P. We can determine whether a knowledge test is true at each point in the system. Thus, given a kb program Pg_{kb} and a system $\mathcal{I}$, we can "run" Pg_{kb}, using $\mathcal{I}$ to determine the truth of the knowledge tests. The set of runs of the resulting protocol determine another system. $\mathcal{I}$ *represents* a kb program Pg_{kb} if $\mathcal{I}$ is a fixed point of this process; that is, running Pg_{kb} with respect to $\mathcal{I}$ gives back $\mathcal{I}$. A protocol P *de facto implements* a kb program Pg_{kb} if P and Pg_{kb} act the same at all points in the system that represents P; see [15].

As observed by HM, once we add the $\Diamond$ operator, the resulting kb program has no representation. For suppose it is represented by a system $\mathcal{I}$. Does i (the agent represented by I) send $\mathit{new\_info}$ to $\mathbf{n}$ in $\mathcal{I}$? If it does, then $\mathcal{I}$ can't represent the program because, in $\mathcal{I}$, i knows that $\mathbf{n}$ will eventually know $\mathit{new\_info}$, should send $\mathit{new\_info}$ to $\mathbf{n}$. Similarly it follows that no agent should send $\mathit{new\_info}$ to $\mathbf{n}$ in $\mathcal{I}$. On the other hand, if no one sends $\mathit{new\_info}$ to $\mathbf{n}$, then $\mathbf{n}$ will not know it, and i should send it. Roughly speaking, i should send the information iff i does not send the information.

HM suggest the use of counterfactuals to deal with this problem. As we said in the introduction, a counterfactual has the form $\varphi > \psi$, which is read as "if φ were the case then ψ". As is standard in the philosophy literature (see, for example, [20, 26]) such a statement is taken to be true if, in the closest worlds to the current world where φ is true, ψ is also true. In particular, we discuss their concrete interpretation of "closest worlds". Once we have counterfactuals, we must consider systems with runs that are not runs of the program. These are runs where, for example, counter to fact, the agent does not send a message (although the program says it should). We make these runs unlikely relative to the actual runs in the system. But the presence of these runs makes it more convenient to consider belief, rather than knowledge. Roughly speaking, this is because there will always be runs that the agent considers possible that are not runs of the program; using belief allows us to exclude these runs. We write $B_{\mathbf{n}}\varphi$ to denote that the agent named $\mathbf{n}$ believes φ, although this is perhaps better read as "the agent named $\mathbf{n}$ knows that φ is (almost certainly) true".

Using counterfactuals, we can modify the program to say that agent i should send the information only if i does not believe "if I do not send the information, then $\mathbf{n}$ will eventually learn it anyway". To capture this, we use the proposition $\mathit{do}(\mathit{send}_{\mathbf{n}}(\mathit{new\_info}))$, which is true if i has sent new_info to $\mathbf{n}$. If there are only finitely many possible values of f, say $v_1, \ldots, v_k$, then the formula $B_{\mathbf{n}}(f = v_1) \vee \ldots \vee B_{\mathbf{n}}(f = v_k)$ captures the fact that the agent with name $\mathbf{n}$ knows the value of f. However, in general, we want to allow an unbounded number of function values. For example, if agents have distinct numerical ids, we are trying to elect as leader the agent with the highest id, and there is no bound on the size of the network, then the set of possible values of f is unbounded. We deal with this problem by allowing limited quantification over values. In particular, we also use formulas of the form $\exists v B_{\mathbf{n}}(f = v)$, which intuitively say that the agent with name $\mathbf{n}$ knows the value of f. Let Pg_{cb}^{GC} denote the following modification of Pg^{GC}:

```
    if $some\_new\_info$ then
      for each $nonempty\ subset$ A $of\ agents$ do
      if $A = \{\mathbf{n} : \neg B_I(\neg do(send_{\mathbf{n}}(new\_info)) > \Diamond(\exists \mathbf{n}'(Calls(\mathbf{n}, I, \mathbf{n}')$
                $\wedge B_{\mathbf{n}}(\mathbf{n}'\text{'s}\, cont(new\_info))) \vee \exists v B_{\mathbf{n}}(f = v))\}$
      then $send_A(new\_info); receive$
```

In this program, the agent i representing I sends $\mathbf{n}$ the new information if i does not believe that $\mathbf{n}$ will eventually learn the new information or the function value in any case. This improved program still solves the global function computation problem whenever possible.

Theorem 4. Pg_{cb}^{GC} *solves the global function computation problem whenever possible: for all $\mathcal{N}$ and f such that the condition in Theorem 1 is satisfied and all protocols P that de facto implement* Pg_{cb}^{GC}*, in every run r of the system that represents P, eventually all agents know $f(N_r)$.*

4 Case Study: Leader Election

In this section we focus on leader election. If we take the function f to describe a method for computing a leader, and require that all agents eventually know who is chosen as leader, this problem becomes an instance of global function computation. We assume that agents have distinct identifiers (which is the context in which leader election has been studied in the literature). It follows from Corollary 2 that leader election is solvable in this context; the only question is what the complexity is. Although leader election is only one instance of the global function computation problem, it is of particular interest, since it has been studied so intensively in the literature. We show that a number of well-known protocols for leader election in the literature essentially implement the program Pg_{cb}^{GC}. In particular, we consider a protocol combining ideas of Lann [19] and Chang and Roberts [6] (LCR from now on) presented by Lynch [21], which works in unidirectional rings, and Peterson's [25] protocol P1 for unidirectional rings and P2 for bidirectional rings. We briefly sketch the LCR protocol and Peterson's protocol P2, closely following Lynch's [21] treatment; we omit the description of P1 for space reasons.

The LCR protocol works in unidirectional rings, and does not assume a bound on their size. Each agent starts by sending its id along the ring; whenever it receives a value, if the value is larger than the maximum value seen so far, then the agent forwards it; if not, it does nothing, except when it receives its own id. If this id is M, the agent then sends the message "the agent with id M is the leader" to its neighbor. Each agent who receives such a message forwards it until it reaches the agent with id M again. The LCR protocol is correct because it ensures that the maximum id travels along the ring and is forwarded by each agent until some agent receives its own id back. That agent then knows that its id is larger than that of any other agent, and thus becomes the leader.

Peterson's protocol P2 for bidirectional rings operates in phases. In each phase, agents are designated as either *active* or *passive*. Intuitively, the active agents are those still competing in the election. Once an agent becomes passive, it remains passive, but continues to forward messages. Initially all agents are active. In each phase, an active agent compares its id with the ids of the closest active agent to its right and the closest active agent to its left. If its id is the largest of the three, it continues to be active; otherwise, it becomes passive. Just as with the LCR protocol, when an agent receives back its own id, it declares itself leader. Then if its id is M, it sends the message "the agent with id M is the leader", which is forwarded around the ring until everyone knows who the leader is.

Peterson shows that, at each phase, the number of active agents is at most half that of the previous phase, and always includes the agent with the largest id. It follows that, eventually, the only active agent is the one with the largest id. Peterson's protocol terminates when the agent that has the maximum id discovers that it has the maximum id by receiving its own id back. The message complexity of Peterson's protocol is thus $O(n \log n)$, where n is the number of agents.

We remark that although they all work for rings, the LCR protocol is quite different from P1 and P2. In the LCR protocol, agents forward their values along their unique outgoing link. Eventually, the agent with the maximum input receives its own value and realizes that it has the maximum value. In P1 and P2, agents are either *active* or *passive*; in each round, the number of active agents is reduced, and eventually only the agent with the maximum value remains active.

Despite their differences, LCR, P1, and P2 all essentially implement Pg_{cb}^{GC}. There are two reasons we write "essentially" here. The first, rather trivial reason is that, when agents send information, they do not send all the information they learn (even if the agent they are sending it to will never learn this information). For example, in the LCR protocol, if agent i learns that its left neighbor has value N and this is the largest value that it has seen, it passes along N without passing along the fact that its left neighbor has this value. We can easily deal with this by modifying the protocols so that all the agents send new_info rather than whatever message they were supposed to send. However, this modification does not suffice. The reason is that the modified protocols send some "unnecessary" messages. This is easiest to see in the case of LCR. Suppose that j is the processor with highest id. When j receives the message with its id back and sends it around the ring again (this is essentially the message saying that j is the leader), in a full-information protocol, j's second message will include the id j' of the processor just before j. Thus, when j' receives j's second message, it will not need to forward it to j. If LCR$'$ is the modification of LCR where each process sends new_info rather than $maxid$, and the last message in LCR is not sent, then we can show that LCR$'$ indeed de facto implements Pg_{cb}^{GC}. The modifications to P2 that are needed to get a protocol P2$'$ that de facto implements Pg_{cb}^{GC} are similar in spirit, although somewhat more complicated. We leave details to the full paper. x

Theorem 5. *The following all hold:*

(a) Given parameter d, the optimal flooding protocol de facto implements Pg_{cb}^{GC} in contexts where (i) all networks have diameter at most d and (ii) all agents have distinct identifiers.
(b) LCR$'$ de facto implements Pg_{cb}^{GC} in all contexts where (i) all networks are unidirectional rings and (ii) agents have distinct identifiers.
(c) There exists a protocol P1$'$ that agrees with P1 up to the last phase (except that it sends new_info) and implements Pg_{cb}^{GC} in all contexts where (i) all networks are unidirectional rings and (ii) agents have distinct identifiers.
(d) There exists a protocol P2$'$ that agrees with P2 up to the last phase (except that it sends new_info) and de facto implements Pg_{cb}^{GC} in all contexts where (i) all networks are bidirectional rings and (ii) agents have distinct identifiers.

Theorem 5 brings out the underlying commonality of all these protocols. Moreover, it emphasizes the connection between counterfactual reasoning and message optimality. Finally, it shows that reasoning at the kb level can be a useful tool for improving the message complexity of protocols. For example, although P2$'$ has the same order of magnitude message complexity as P2 ($O(n \log n)$), it typically sends $O(n)$ fewer messages. While this improvement comes at the price of possibly longer messages, it does suggest that this approach can result in nontrivial improvements. Moreover, it suggests that starting with a high-level kb program and then trying to implement it using a standard program can be a useful design methodology. Indeed, our hope is that we will be able to synthesize standard programs by starting with high-level kb specifications, synthesizing a kb program that satisfies the specification, and then instantiating the kb program as a standard program. We have some preliminary results along these lines that give us confidence in the general approach [5]; we hope that further work will lend further credence to this approach.

References

1. D. Angluin. Local and global properties in netwroks of processors. In *Proc. 12th ACM Symp. on Theory of Computing*, pages 82–93, 1980.
2. H. Attyia, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.
3. H. Attyia, M. Snir, and M. K. Warmuth. Computing on an anonymous ring. *Journal of ACM*, 35(4):845–875, 1988.
4. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
5. M. Bickford, R. L. Constable, J. Y. Halpern, and S. Petride. Knowledge-based synthesis of distributed systems using event structures. In *Proc. 11th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, Lecture Notes in Computer Science, vol. 3452, pages 449–465. Springer-Verlag, 2005.
6. E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

7. C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 88(2):156–186, 1990.
8. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, Mass., 1995. A revised paperback edition was published in 2003.
9. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
10. L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
11. R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5(1):66–77, 1983.
12. A. J. Grove. Naming and identity in epistemic logic II: a first-order logic for naming. *Artificial Intelligence*, 74(2):311–350, 1995.
13. A. J. Grove and J. Y. Halpern. Naming and identity in epistemic logics, Part I: the propositional case. *Journal of Logic and Computation*, 3(4):345–378, 1993.
14. V. Hadzilacos. A knowledge-theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, 1987.
15. J. Y. Halpern and Y. Moses. Using counterfactuals in knowledge-based programming. *Distributed Computing*, 17(2):91–106, 2004.
16. J. Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. *SIAM Journal on Computing*, 31(3):838–865, 2001.
17. J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols. *Journal of the ACM*, 39(3):449–478, 1992.
18. R. E. Johnson and F. B. Schneider. Symmetry and similarity in distributed systems. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*, pages 13–22, 1985.
19. Gerard Le Lann. Distributed systems–towards a formal approach. In *IFIP Congress*, volume 7, pages 155–160, 1977.
20. D. K. Lewis. *Counterfactuals*. Harvard University Press, Cambridge, Mass., 1973.
21. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1997.
22. M. S. Mazer and F. H. Lochovsky. Analyzing distributed commitment by reasoning about knowledge. Technical Report CRL 90/10, DEC-CRL, 1990.
23. R. Milner. *Communication and Concurrency*. Prentice Hall, Hertfordshire, 1989.
24. Y. Moses and G. Roth. On reliable message diffusion. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 119–128, 1989.
25. G. L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Trans. on Programming Languages and Systems*, 4(4):758–762, 1982.
26. R. C. Stalnaker. A semantic analysis of conditional logic. In N. Rescher, editor, *Studies in Logical Theory*, pages 98–112. Oxford University Press, 1968.
27. F. Stulp and R. Verbrugge. A knowledge-based algorithm for the Internet protocol (TCP). *Bulletin of Economic Research*, 54(1):69–94, 2002.
28. M. Yamashita and T. Kameda. Computing on anonymous networks. I. Characterizing the solvable cases. *IEEE Trans. on Parallel and Distributed Systems*, 7(1):69–89, 1996.
29. M. Yamashita and T. Kameda. Leader election problem on networks in which processor identity numbers are not distinct. *IEEE Trans. on Parallel and Distributed Systems*, 10(9):878–887, 1999.

Checking a Multithreaded Algorithm with $^+$CAL

Leslie Lamport

Microsoft Research

Abstract. A colleague told me about a multithreaded algorithm that was later reported to have a bug. I rewrote the algorithm in the $^+$CAL algorithm language, ran the TLC model checker on it, and found the error. Programs are not released without being tested; why should algorithms be published without being model checked?

1 Introduction

On a Wednesday afternoon in March of this year, my colleague Yuan Yu told me that Detlefs et al. [1] had described a multithreaded algorithm to implement a shared two-sided queue using a double-compare-and-swap (DCAS) operation, and that Doherty et al. [2] later reported a bug in it. I decided to rewrite the algorithm in $^+$CAL [3] and check it with the TLC model checker, largely as a test of the $^+$CAL language. After working on it that afternoon, part of Sunday, and a couple of hours on Monday, I found the bug. This is the story of what I did. A $^+$CAL specification of the algorithm and an error trace that it produced are available on the Web [4]. I hope my experience will inspire computer scientists to model check their own algorithms before publishing them.

$^+$CAL is an algorithm language, not a programming language. It is expressive enough to provide a practical alternative to informal pseudo-code for writing high-level descriptions of algorithms. It cannot be compiled into efficient executable code, but an algorithm written in $^+$CAL can be translated into a TLA$^+$ specification that can be model checked or reasoned about with any desired degree of formality. Space does not permit me to describe the language and its enormous expressive power here. The two-sided queue algorithm is a low-level one, so its $^+$CAL version looks much like its description in an ordinary programming language. The features of $^+$CAL relevant to this example are explained here. A detailed description of the language along with the translator and model-checker software are on the Web [3].

1.1 The Story

I began by converting the algorithm from the C code of the paper into $^+$CAL as a collection of procedures, the way Detlefs et al. described it. (They actually extended C with an `atomically` statement to represent the DCAS operation, and they explained in the text what operations were considered atomic.) Other

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 151–163, 2006.

than minor syntax errors, the only bug in my first try was an incorrect modeling of the DCAS operation caused by my confusion about C's "&" and "*" operators. I found my errors by running TLC on small instances of the algorithm, and I quickly fixed them.

I next wrote a small test harness consisting of a collection of processes that nondeterministically called the procedures. It kept an upper-bound approximation to the multi-set of queued elements and checked to make sure that the element returned by a *pop* was in that multi-set. It also kept a lower bound on the number of queued elements and checked for a *pop* returning "empty" when it shouldn't have. However, my test for an incorrect "empty" was wrong, and there was no simple way to fix it. So, I eliminated that test.

Running TLC on an instance of the algorithm with 2 enqueable values, 2 processes, and a heap of size 3 completed in 17 minutes, finding no bug. (Except where noted, execution times are for a 2.4 or 3 GHz personal computer.) The algorithm uses a fixed "dummy" node, so the maximum queue length is one less than the heap size. My next step was to check it on a larger model. I figured that checking with only a single enqueable value should suffice, because a *pop* that correctly removed an element was unlikely to return anything other than that element's correct value. I started running TLC on a model with 3 processes and 4 heap locations just before leaving for a three-day vacation. I returned to find that my computer had crashed after running TLC for two or three hours. I rebooted and restarted TLC from a checkpoint. A day later I saw that TLC had not yet found an error and its queue of unexamined states was still growing, so I stopped it.

I next decided to write a higher-level specification and let TLC check that the algorithm implemented this specification under a suitable refinement mapping [5] (often called an abstraction function). I also wrote a new version of the algorithm, without procedure calls, to reduce the size of the state space. This turned out to be unnecessary; TLC would easily have found the bug without that optimization.

I made a first guess at the refinement mapping based on the pictures in the Detlefs et al. paper showing how the implementation worked, but it was wrong. Correcting it would have required understanding the algorithm, and I didn't want to take the time to do that. Instead, I decided that an atomic change to the queue in the abstract specification was probably implemented by a successful DCAS operation. So, I added a dummy variable *queue* to the algorithm that is modified in the obvious way when the DCAS operation succeeds, and I wrote a simple refinement mapping in which the abstract specification's queue equaled *queue*. However, this refinement mapping didn't work right, producing spurious error reports on *pop* operations that return "empty".

A *pop* should be allowed to return "empty" if the abstract queue was empty at any time between the call to and return from the operation. I had to add another dummy variable to the algorithm to record if the queue had been empty between the call and return, and to modify the specification. Having added these dummy variables, I realized that I could check correctness by simply adding assertions to the algorithm; there was no need for a high-level specification and refinement mapping. I added the assertions, and TLC found the bug in about 20 seconds for an instance with 2 processes and 4 heap locations. (TLC would have found the bug with only 3 heap locations.) The bug was manifest by a *pop* returning "empty" with a non-empty queue—a type of error my first attempt couldn't detect.

After finding the error, I looked at the paper by Doherty et al. to check that I had found the same error they did. I discovered that I had found one of two errors they reported. I then removed the test that caught the first error and tried to find the second one. TLC quickly discovered the error on a model with 3 processes and 4 heap locations. As explained below, getting it to do this in a short time required a bit of cleverness. Using a naive, straightforward approach, it took TLC $1\frac{2}{3}$ days to find the error. Explicit-state model checking is well suited for parallel execution, and TLC can make use of shared-memory multiprocessing and networked computing. Run on a 384 processor Azul Systems computer [6], TLC found the error in less than an hour.

1.2 The Moral

I started knowing only that there was a bug in the algorithm. I knew nothing about the algorithm, and I had no idea what the bug was—except that Yuan Yu told me that he thought a safety property rather than a liveness property was violated. (I would have begun by looking for a safety violation anyway, since that is the most common form of error.) I still know essentially nothing about how the algorithm was supposed to work. I did not keep a record of exactly how much time I spent finding the error, but it probably totaled less than 10 hours. Had Detlefs et al. used $^{+}$CAL as they were developing their algorithm, model checking it would have taken very little extra time. They would certainly have found the first error and would probably have found the second.

There are two reasons I was able to find the first bug as quickly as I did, despite not understanding the algorithm. The obvious reason is that I was familiar with $^{+}$CAL and TLC. However, because this is a very low-level algorithm, originally written in simple C code, very little experience using $^{+}$CAL was needed. The most difficult part of $^{+}$CAL for most people is its very expressive mathematical expression language, which is needed only for describing more abstract, higher-level algorithms. The second reason is that the algorithm was expressed in precise code. Published concurrent algorithms are usually written in very informal pseudo-code, and it is often necessary to understand the algorithm from its description in the text in order to know what the pseudo-code is supposed to mean. In this case, the authors clearly stated what the algorithm did.

Section 2 describes the algorithm's translation from C to $^{+}$CAL, and Section 3 describes how I checked it. The translation is quite straightforward. Had $^{+}$CAL been available at the time, I expect Detlefs et al. would have had no trouble doing it themselves. However, they would have gotten more benefit by using $^{+}$CAL from the start instead of C (or, more precisely, pseudo-C). Before devising the published algorithm, they most likely came up with other versions that they later found to be wrong. They probably would have discovered those errors much more quickly by running TLC on the $^{+}$CAL code. Algorithms are often developed by trial and error, devising a plausible algorithm and checking if it works in various scenarios. TLC can do the checking for small instances much faster and more thoroughly than a person.

2 Translating from the C Version

A major criterion for the $^{+}$CAL language was simplicity. The measure of a language's simplicity is how simple its formal semantics are. A $^{+}$CAL algorithm is translated to a TLA^{+} specification [7], which can then be checked by TLC. The TLA^{+} translation defines the meaning of a $^{+}$CAL algorithm. (Because TLA^{+} is based on ordinary mathematics, its formal semantics are quite simple.) The simplicity of $^{+}$CAL was achieved by making it easy to understand the correspondence between a $^{+}$CAL algorithm and its translation. The translator itself, which is implemented in Java, is specified in TLA^{+}.

Simplicity dictated that $^{+}$CAL eschew many common programming language concepts, like pointers, objects, and types. (Despite its lack of such constructs, and in part because it is untyped, $^{+}$CAL is much more expressive than any programming language.) C's pointer operations are represented in the $^{+}$CAL version using an explicit array (function) variable *Heap* indexed by (with domain) a set of addresses. A pointer-valued variable like *lh* in the C version becomes an address-valued variable, and the C expression `lh->L` is represented by the $^{+}$CAL expression $Heap[lh].L$.

The only tricky part of translating from pointers to heap addresses came in the DCAS operation. Figure 1 contains the pseudo-C version. Such an operation

```
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. The DCAS operation in pseudo-C

is naturally defined in $^{+}$CAL as a macro. A $^{+}$CAL macro consists of a sequence of statements, not an expression. I therefore defined a DCAS macro with an additional first argument, so the $^{+}$CAL statement

```
DCAS(result, a1, ... )
```

represents the C statement

```
result = DCAS(a1, ... )
```

The difficulty in writing the DCAS macro came from the pointer arguments *addr*1 and *addr*2. A direct translation of the original DCAS operation would have required an extra layer of complexity. A pointer-valued variable like *lh* would have had to be represented by a variable whose value was not a heap address, but rather the address of a memory location containing a heap address. However, this complication was unnecessary because, in all the algorithm's uses of the DCAS operation, the first two arguments are &-expressions. In the translation, I essentially defined the DCAS macro as if the "*"s were removed from the `*addr1` and `*addr2` parameters in Figure 1, and the "&"s were removed from the uses of the macro. This led me to the macro definition of Figure 2. (The "||" multiple-assignment construct is explained in Section 2.3 below; for now, consider it to be a semicolon.) The statement

```
macro DCAS(result, addr1, addr2, old1, old2, new1, new2) {
  if ( (addr1 = old1) ∧
       (addr2 = old2)) {
    addr1 := new1 ||
    addr2 := new2 ;
    result := TRUE;
  } else result := FALSE; }
```

Fig. 2. The DCAS operation in $^{+}$CAL

```
if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy)) ...
```

is then represented in $^{+}$CAL as

```
DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy) ;
if (temp) ...
```

The $^{+}$CAL translator replaces the `DCAS` statement by the syntactic expansion of the `DCAS` macro. (As explained below, the `atomically` is implicit in the $^{+}$CAL version.)

Most of my colleagues cannot immediately see that the result of the substitution is a correct translation of the C version. Since the expanded $^{+}$CAL macro is quite simple, any difficulty must lie in understanding C's "*" and "&" operators. A language for describing algorithms should be simple, and its operators should be easy to understand.

As an illustration of the translation, Figure 3 shows the original C version of the *popLeft* procedure, exactly as presented by Detlefs et al., and my $^{+}$CAL

C Version

```
 1 val popLeft() {
 2   while (true) {
 3     lh = LeftHat;
 4     rh = RightHat;
 5     if (lh->L == lh) return "empty";
 6     if (lh == rh) {
 7       if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy))
 8         return lh->V;
 9     } else {
10       lhR = lh>R;
11       if (DCAS(&LeftHat, &lh>R, lh, lhR, lhR, lh)) {
12         result = lh->V;
13         lh->L = Dummy;
14         lh->V = null;
15         return result;
16 } } } }
```

$^+$CAL Version

```
 procedure popLeft()
 variables rh, lh, lhR, temp, result; {
02:  while (TRUE) {
       lh := LeftHat;
04:    rh := RightHat;
05:    if (Heap[lh].L = lh) {rVal[self] := "empty";  return};
06:    if (lh = rh) {
         DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy);
         if (temp) {
08:        rVal[self] := Heap[lh].V; return}
       } else {
010:     lhR := Heap[lh].R;
011:     DCAS(temp, LeftHat, Heap[lh].R, lh, lhR, lhR, lh);
         if (temp) {
012:       result := Heap[lh].V;
013:       Heap[lh].L := Dummy ||
           Heap[lh].V := null;
           rVal[self] := result;  015: return ;
} } } }
```

Fig. 3. The C and $^+$CAL versions of the *popLeft* procedure

version. This was my original translation, before I added dummy variables for checking. (While it is essentially the same as my first version, I have made a number of cosmetic changes—mainly reformatting the code and changing label names to correspond to the line numbers of the corresponding control points in the C version.) The non-obvious aspects of the $^+$CAL language that appear in this example are explained in Sections 2.1–2.3 below.

A cursory examination shows how similar the two versions are. I of course formatted the $^+$CAL version to look as much as possible like the C version. (To this end, I used $^+$CAL's more compact c-syntax rather the alternative p-syntax that is a bit easier to understand.) The $^+$CAL version is three lines longer because of the extra line added in translating each DCAS operation and because of the local variable declarations that are missing from the pseudo-C code.

2.1 Labels and the Grain of Atomicity

Labels are used to specify the grain of atomicity in a $^+$CAL algorithm. Execution of a single atomic step begins at a label and continues until the next label that is encountered. For example, execution of a step starting with control at label `O6`:

- ends at `O8` if $lh = rh$ evaluates to TRUE and the DCAS operation sets *temp* to TRUE.
- ends at `O2` if $lh = rh$ evaluates to TRUE and the DCAS operation sets *temp* to FALSE.
- ends at `O10` if $lh = rh$ evaluates to FALSE.

Because the DCAS macro contains no labels, its execution is atomic. ($^+$CAL does not permit labels in a macro definition.)

To simplify model checking and reasoning about an algorithm, one wants to write it with the coarsest possible grain of atomicity that permits all relevant interleavings of actions from different processes. Detlefs et al. assume that a read or write of a single memory value is atomic. (Presumably, a heap address and an enqueued value each constitute a single memory value.)

I have adopted the standard method of using the coarsest grain of atomicity in which each atomic action contains only a single access to a shared data item. The shared variables relevant for this procedure are *Heap*, *LeftHat*, and *RightHat*. However, the labeling rules of $^+$CAL required some additional labels. In particular, the label `O2` is required, even though the call of the procedure affects only the process's local state and could be made part of the same action as the evaluation of *LeftHat*.

2.2 Procedures

To maintain the simplicity of its translation to TLA$^+$, a $^+$CAL procedure cannot return a value. Values are passed through global variables. In this algorithm, I have used the variable *rVal* to pass the value returned by a procedure. When executed by a process p, a procedure returns the value v by setting $rVal[p]$ to v. In $^+$CAL code, `self` equals the name of the current process.

2.3 Multiple Assignment

One of $^+$CAL's restrictions on labeling/atomicity, made to simplify the TLA$^+$ translation, is that a variable can be assigned a value by at most one statement during the execution of a single atomic step. A single multiple assignment statement can be used to set the values of several components of a single variable. A multiple assignment like

```
Heap[lh].L := Dummy || Heap[lh].V := null
```

is executed by first evaluating all the right-hand expressions, and then performing all the indicated assignments.

3 Checking the Algorithm

To check the correctness of the algorithm, I added two global history variables:

- *queue*, whose value is the state of the abstract queue.
- *sVal*, where *sVal*[*p*] is set by process *p* to remember certain information about the state for later use.

Adding dummy variables means rewriting the algorithm by adding statements that set the new variables but do not change the behavior if the values of those variables are ignored [5]. The +CAL code for the *popLeft* procedure with the dummy variables appears in Figure 4. (The original code is in gray.)

The *queue* variable is modified in the obvious way by an atomic step that contains a successful DCAS operation (one that sets *temp* to TRUE). The `assert` statements in steps `O8` and `O12` check that the value the procedure is about to return is the correct one.

The `assert` statement in step `O5` attempts to check that, when the procedure is about to return the value `"empty"`, it is permitted to do so. An `"empty"` return value is legal if the abstract queue was empty at some point between when the procedure was called and when it returns. The assertion actually checks that the queue was empty when operation `O2` was executed or is empty when the procedure is about to execute the return in operation `O5`. This test is pessimistic. The assertion would fail if operations of other processes made the queue empty and then non-empty again some time between the execution of those two operations, causing TLC incorrectly to report an error. For a correct test, each process would have to maintain a variable that is set by other processes when they remove the last item from the queue. However, the shared variables whose values determine if the procedure returns `"empty"` are read only by these two operations. Such a false alarm therefore seemed unlikely to me, and I decided to try this simpler test. (Knowing for sure would have required understanding the algorithm.) TLC returns a shortest-length path that contains an error. I therefore knew that, if the assertion could reveal an error, then TLC would produce a trace that showed the error rather than a longer trace in which another process happened to empty the queue at just the right time to make the execution correct. This assertion did find the bug—namely, a possible execution containing the following sequence of relevant events:

```
 procedure popLeft()
 variables rh, lh, lhR, temp, result; {
02:  while (TRUE) {
       lh := LeftHat;
       sVal[self] := (queue = << >>);
04:    rh := RightHat;
05:    if (Heap[lh].L = lh) {
         assert sVal[self] ∨ (queue = ⟨⟩) ;
         rVal[self] := "empty";  return ;} ;
06:    if (lh = rh) {
         DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy);
         if (temp) {
           sVal[self] := Head(queue);
           queue := Tail(queue);
08:        rVal[self] := Heap[lh].V;
           assert rVal[self] = sVal[self];
           return}
       } else {
010:     lhR := Heap[lh].R;
011:     DCAS(temp, LeftHat, Heap[lh].R, lh, lhR, lhR, lh);
         if (temp) {
           sVal[self] := Head(queue) ;
           queue := Tail(queue) ;
012:       result := Heap[lh].V;
           assert result = sVal[self] ;
013:       Heap[lh].L := Dummy ||
           Heap[lh].V := null;
           rVal[self] := result;  015: return ;
} } } }
```

Fig. 4. The *popLeft* procedure with checking

- Process 1 begins a *pushRight* operation.
- Process 1's *pushRight* operation completes successfully.
- Process 1 begins a *pushLeft* operation.
- Process 2 begins a *popLeft* operation
- Process 1's *pushLeft* operation completes successfully.
- Process 1 begins a *popRight* operation.
- Process 2's *popLeft* operation returns the value "empty".

The actual execution trace, and the complete +CAL algorithm that produced it, are available on the Web [4].

After finding the bug, I read the Doherty et al. paper and found that there was another error in the algorithm that caused it to pop the same element twice from the queue. I decided to see if TLC could find it, using the version without procedures. The paper's description of the bug indicated that it could occur in a much coarser-grained version of the algorithm than I had been checking. (Since an execution of a coarse-grained algorithm represents a possible execution of a finer-grained version, an error in the coarse-grained version is an error in the original algorithm. Of course, the converse is not true.) To save model-checking time, I removed as many labels as I could without significantly changing the code, which was about 1/3 of them. I then ran TLC on an increasing sequence of models, and in a few hours it found the error on a model with 3 processes and 4 heap locations, reporting an execution that described the following sequence of events:

- Process 1 begins a *pushRight* operation.
- Process 2 begins a *popRight* operation.
- Process 1's *pushRight* operation completes successfully.
- Process 1 begins a *popRight* operation.
- Process 3 begins and then successfully completes a *pushLeft* operation.
- Process 3 begins a *popLeft* operation.
- Process 1's *popRight* operation completes successfully.
- Process 1 begins and then successfully completes a *pushLeft* operation.
- Process 1 begins and then successfully completes a *popLeft* operation.
- Process 2's *popRight* operation completes successfully.
- Process 3's *popLeft* operation tries to remove an item from an empty queue.

I found the second bug quickly because I knew how to look for it. However, checking models on a coarser-grained version when the fine-grained version takes a long time is an obvious way of speeding up the search for bugs. It is not often done because, when written in most model-checking languages, changing an algorithm's grain of atomicity is not as easy as commenting out some labels. Someone who understands the algorithm will have a sense of how coarse a version is likely to reveal errors.

I decided to see how long it would take TLC to find the bug by checking the fine-grained version. I started it running shortly before leaving on a long trip. When I returned, I found that it had indeed found the error—after running for a little more than a month. Examining the +CAL code, I realized that it had two unnecessary labels. They caused some operations local to a process to be separate steps, increasing the number of reachable states. I removed those labels. I estimate that TLC would have found the error in the new version in about two weeks. However, by observing processor utilization, it was easy to see that TLC was spending most of its time doing disk I/O and was therefore memory-bound. I had been running it on a 2.4 GHz personal computer with 1 GByte of memory. I switched to a 3 GHz machine with 4 GBytes of memory, and TLC found the error in 40 hours. Running the model checker in the background for a couple of days is not a problem.

TLC can be instructed to use multiple processors. We have found that it can obtain a factor of n speedup by using n processors, for n up to at least 8. TLC can therefore take full advantage of the coming generation of multicore computers. (Inefficiencies of the Java runtimes currently available for personal computers significantly reduce the speedup obtained with those machines.) Using a version of TLC modified for execution with a large number of processors, the 40-hour uniprocessor execution was reduced to less than an hour on a first-generation 384-processor Azul Systems computer [6]. Since each of that computer's processors is much slower than 3 GHz, this probably represents a speedup by close to a factor of 384. (Azul Systems does not reveal the actual speed of their processors.) It is likely that, within a few years, computers will be widely available on which TLC runs 10 times faster than it does today.

There is an amusing footnote to this story. After doing the checking, I noticed that I had inadvertently omitted a label from the *pushRight* operation, letting one atomic action access two shared variables. I added the missing label and ran TLC on the slightly finer-grained algorithm, using the same personal computer as before. Because TLC does a breadth-first exploration of the state space, I knew that it would find a counterexample with one additional step. Indeed, it found exactly the same error trace, except with one of the original 46 steps split into two. However, instead of 40 hours, TLC took only 37.2 hours! It found the error after examining 148 million distinct states rather than 157 million. Figuring out how this could happen is a nice puzzle.

4 Conclusion

$^{+}$CAL was not needed for this example. The algorithm could have been written in other languages with model checkers. Promela, the language of the Spin model checker, would probably have been a fine choice [8]. In fact, Doherty did use Spin to demonstrate the bug, although he wrote the Promela version expressly to find the bug he had already discovered [9]. However, most concurrent algorithms are not written as low-level pseudo-C programs. They are often written as higher-level algorithms, which are naturally described using mathematical concepts like quantification, sets, and sequences rather than the primitive operators provided by languages like C and Java. For such algorithms, $^{+}$CAL is clearly superior to Promela and similar model-checking languages.

One can model check not only the algorithm, but also its proof. Because TLA^{+} is based on mathematics, TLC is well suited to check an algorithm's proof. Rigorous proofs require invariance reasoning and may also involve showing that the algorithm implements a higher-level specification under a refinement mapping. TLC can check both invariance and implementation under a refinement mapping. Since they understood the algorithm, Detlefs et al. would have been able to define *queue* as a refinement mapping instead of adding it as a dummy variable the way I did. An error in an invariant or refinement mapping usually manifests itself before the algorithm does something wrong, allowing a model checker to find the problem sooner. Checking a refinement mapping might have revealed the two-sided queue algorithm's second error quickly, even on the fine-grained version.

Model checking is no substitute for proof. Most algorithms can be checked only on instances of an algorithm that are too small to give us complete confidence in their correctness. Moreover, a model checker does not explain why the algorithm works.

Conversely, a hand proof is no substitute for model checking. As the two-sided queue example shows, it is easy to make a mistake in an informal proof. Model checking can increase our confidence in an algorithm—even one that has been proved correct.

How much confidence model checking provides depends upon the algorithm. A simple, easy-to-use model checker like TLC can verify only particular instances—for example, 3 processes and 4 heap locations. The number of reachable states, and hence the time required for complete model checking, increases exponentially with the size of the model. Only fairly small instances can be checked. However, almost every error manifests itself on a very small instance—one that may or may not be too large to model check. TLC can also check randomly generated executions on quite large instances. However, such checking can usually catch only simple errors.

My own experience indicates that model checking is unlikely to catch subtle errors in fault-tolerant algorithms that rely on replication and unbounded counters. It does much better on traditional synchronization algorithms like the two-sided queue implementation. However, even when it cannot rule out subtle errors, model checking is remarkably effective at catching simpler errors quickly. One can waste a lot of time trying to prove the correctness of an algorithm with a bug that, in retrospect, is obvious.

Model checking can be viewed as a sophisticated form of testing. Testing is not a substitute for good programming practice, but we don't release programs for others to use without testing them. For years, model checking has been a standard tool of hardware designers. Why is it seldom used by algorithm designers? With $^{+}$CAL, there is no longer the excuse that the language of model checkers is too low-level for describing algorithms. Model checking algorithms prior to submitting them for publication should become the norm.

Acknowledgment

I want to thank Homayoon Akhiani for model checking the two-sided queue algorithm with TLC on an Azul Systems multiprocessor.

References

1. Detlefs, D.L., Flood, C.H., Garthwaite, A.T., Martin, P.A., Shavit, N.N., Jr., G.L.S.: Even better dcas-based concurrent deques. In Herlihy, M., ed.: Distributed Algorithms. Volume 1914 of Lecture Notes in Computer Science., Toledo, Spain, ACM (2000) 59–73
2. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: Dcas is not a silver bullet for nonblocking algorithm design. In Gibbons, P.B., Adler, M., eds.: SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, Barcelona, ACM (2004) 216–224

3. Lamport, L.: The $^+$CAL algorithm language. (URL `http://research.microsoft.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "`-`" from `uid-lamportpluscalhomepage`.)
4. Lamport, L.: An example of using $^+$CAL to find a bug. (URL `http://research.microsoft.com/users/lamport/tla/dcas-example.html`. The page can also be found by searching the Web for the 28-letter string formed by concatenating `uid` and `lamportdcaspluscalexample`.)
5. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science **82**(2) (1991) 253–284
6. Azul Systems: Web page. (`http://www.azulsystems.com`)
7. Lamport, L.: Specifying Systems. Addison-Wesley, Boston (2003)
8. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
9. Moir, M.: Private communication. (2006)

Capturing Register and Control Dependence in Memory Consistency Models with Applications to the Itanium Architecture

Lisa Higham[1], LillAnne Jackson[1,2], and Jalal Kawash[3,1]

[1] Department of Computer Science, The University of Calgary, Calgary, Canada
[2] Department of Computer Science, The University of Victoria, Victoria, Canada
[3] Department of Computer Science, American University of Sharjah, UAE
higham@cpsc.ucalgary.ca, jackson@cpsc.ucalgary.ca, jkawash@aus.edu

Abstract. A *complete* framework for modelling memory consistency that includes register and control dependencies is presented. It allows us to determine whether or not a given computation could have arisen from a given program running on a given multiprocessor architecture. The framework is used to provide an exact description of the computations of (a subset of) the Itanium instruction set on an Itanium multiprocessor architecture. We show that capturing register and control dependencies is crucial: a producer/consumer problem is solvable without using strong synchronization primitives on Itanium multiprocessors, but is impossible without exploiting these dependencies.

Keywords: Multiprocessor memory consistency, register and control dependency, Itanium, process coordination.

1 Introduction

To overcome inefficiency bottlenecks, modern shared memory multiprocessors have complicated memory organization such as replication in caches and write buffers, multiple buses, and out-of-order memory accesses. This causes processors to have differing views of the shared memory, which satisfy only some weak consistency guarantees. To program such systems, programmers need a precise specification of these guarantees, expressed as constraints on the outcomes of executions of programs. We provide a general and intuitive framework for defining the complete memory consistency model of a multiprocessor architecture including its control and register dependencies. The framework facilitates the specification of the exact set of computations that can arise from a multiprocessor system.

As a running example, we illustrate our framework with the Itanium programming language and architecture. Our techniques are exploited to specify the sets of computations that can arise from a multiprocessor program that uses a subset of the Intel Itanium instruction set when it is executed on an Itanium multiprocessor architecture. Since Itanium multiprocessors have a complicated architecture for which a complete correct description of its memory consistency, at the level of indivisible Itanium instructions, does not exist in the literature,

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 164–178, 2006.

this is a second contribution of this paper. Another contribution is to show that including register and control dependencies in memory consistency models is crucial. We prove that a certain producer/consumer coordination problem is solvable without using strong synchronization primitives on Itanium multiprocessors, but is impossible without exploiting these dependencies.

Our complete framework is an extension of our previous work [5]. It maintains the ability to describe systems at different levels of abstraction and to prove their equivalence. But our previous framework could only capture constraints that arise from shared memory; it did not capture the constraints that arise from control dependencies in an individual processor's program, nor consistency constraints due to private register dependencies. While our previous framework could be used to determine if a given computation could arise from a given shared memory architecture, it did not extend to answering if the computation could arise from executing a given program on that architecture. These shortcoming are corrected in this paper. We have also used similar ideas to explore the possibility or impossibility of implementing some objects on various weak memory consistency machines [8,9,10,11]. A common feature of all of these is the notion of abstract specifications and implementations, as a generalization of the methodology used for algorithm design in linearizable [4] or sequentially consistent systems. Even though the systems we consider are substantially weaker than these, we can compose our implementations to achieve implementations of abstract objects on various multiprocessors with weak memory models (or to prove impossibilities).

Local dependencies are modelled in [1,2]. Indeed, the framework of [1] contains many of the features of our framework, including the association of a program with an instantiation, which is similar to our computations, and the capturing of branching and register dependencies. We are unaware, however, of how to use that framework to describe systems at various levels and to prove equivalence.

There are several models for Itanium computations in the literature. Chatterjee and Gopalakrishnan [3] presented an operational model that they describe as a simple formal model for Itanium. Their model is simplified and does not include data dependencies. Yang, Gopalakrishnan, *et. al.* [18] specify Itanium memory ordering rules in terms that can be applied to verification using Prolog with a finite domain constant solver and a boolean Satisfiability checker. Joshi, Lamport, *et. al.* [16] applied the TLA+ specification language to the Itanium manual specifications [14] and use the TLC model checker on the resulting specifications. It was this work that brought about a clear description of the Itanium memory order [12], but it does not inclued a formal definition of local dependence order.

We use only enough of a simplified Itanium system to illustrate our framework with an Itanium running example. Other work [7] (also [15] in progress) focuses on the completed Itanium memory consistency. Thus we omit store release, load acquire, semaphore or memory fence instructions, and we do not consider procedure stack adjustments. Furthermore, we assume that memory accesses that are not to identical locations do not overlap, all memory is in the same coherence domain, and all memory locations are cacheable (called WB in the Itanium manuals).

2 Modelling Multiprocessor Systems and Their Computations

We model a multiprocessor system as a collection of *programs* operating on a collection of *objects* under some constraints called a *memory consistency model.*

Informally, as each processor of such a system executes its program, it issues a sequence of *operation invocations* on the objects of the system, and receives *operation responses* for each invocation. Matching each response with its invocation provides each processor with a sequence of *operations*, which is its "view" of the execution. We think of a *computation* as the collection of these views — one for each processor in the system. Because the response of an invocation is determined by the programs of the other processors, the asynchronous interaction of the processors, and the memory organization of the system, there are typically many computations possible for each system. For any such system we seek a way to determine exactly what computations are possible.

Although our framework is general, and can be used for any multiprocessor system, this paper focuses on the computations that can arise from a given program running on some typical multiprocessor architecture, say $\mathcal{A}$, where the objects are private registers and shared memory locations. We use a running example based on the Intel Itanium multiprocessor architecture to illustrate the definitions and their use.

Instructions and programs: The *programming language* of $\mathcal{A}$ consists of a collection of (machine specific) operations, called *instructions*, that perform a variety of load, store, arithmetic and logical functions. Instructions are partitioned into two classes:

- Branch instructions are all instructions that contain a conditional or unconditional branch to a specified label including procedure calls and procedure returns, and
- operational instructions are all other instructions.

Branch instructions transfer program control to a target instruction specified by a label, by operating on the value in the program counter register. Operational instructions move data from one register/memory location to another and/or perform arithmetic or logical functions on the data in registers/memory locations.[1]

Example - Itanium instruction: The Itanium operational instruction "ld8 $r2$=[$r1$]" takes the value from the memory location whose address is stored in register $r1$, and places that value in register $r2$. In any execution, this instruction

[1] The instructions of a program are assumed to be in a format that has been prepared to be processed by an assembler, i.e., some instructions are preceded by labels and the target of a branch is specified by such a label.

```
1. loop:       ld8     r1 = [r3]        ;load r1 with data whose address is in r3
2.             add     r1 = #1, r1      ;add 1 to r1
3.             cmp.ge  p1, p0=#3,r1     ;if 3 is greater than or equal to the value in r1 then
4.        (p1) br      loop             ; go back to loop
5.             st8     [r4], r1         ;store value of r1 into memory whose address is in r4
```

Fig. 1. An Itanium individual program

has associated with it some value v in $r1$, some value w in the memory location with address v and the value u stored in $r2$.

An *individual program* is a finite sequence of instructions from the programming language of $\mathcal{A}$.

Example - Itanium individual program: Figure 1 is an example of an Itanium individual program.

A *multiprogram* for an n processor machine with architecture $\mathcal{A}$ is a collection of n individual programs— one assigned to each processor.

Computations: An instruction that is augmented with *arbitrary* associated values (from the allowed domain), for all the registers and memory locations that it reads is a *completed instruction.* For a conditional branch instruction, "br(*cond*) label", the domain of register *cond* is $\{0, 1\}$, and associating one of these values resolves the conditional branch. So the completed form of this instruction is shortened to exactly one of two read operations that access *cond* and return 0 or 1 (namely, 0=read(*cond*) or 1=read(*cond*)). An *individual computation* of an individual processor is *any* sequence of completed instructions. We do not (yet) care what the instructions are or what values are associated with the instructions. A *(multiprocessor) computation* is a set of individual computations, one for each processor. Notice that a computation is defined non-operationally. It is not a single sequence describing an execution of the whole system, nor is there any relationship required between the register and memory values associated with the different completed instructions. Informally, a multiprocessor computation simply records the sequence of operational instructions performed (and therefore completed) by each processor and the values in condition registers that it read along the way.

Example - 2-processor computations: Three 2-processor computations are given in Figures 2, 3 and 4. (The actual values recorded in these computation can be bizarre, but they are still computations.)

Central goal: We aim to construct a comprehensive framework that is capable of capturing the control and register dependences of any multiprocessor architecture $\mathcal{A}$, and the constraints that arise from the memory organization of $\mathcal{A}$. The definition of any such $\mathcal{A}$ must be precise enough to decide the following *Architecture-Computation* decision problem.

Input: a multiprogram P, containing an individual program for each processor of $\mathcal{A}$, and a multiprocessor computation C.

Question: Could C arise from executing P on $\mathcal{A}$?

p_1	p_2
ld8 $r1 = [r3]$ (ν_3=1284, $\nu_{(1284)}$=7)	ld8 $r1 = [r3]$ (ν_3=1027, $\nu_{(1027)}$=12)
add $r1 = \#1,r1$ (ν_1=15)	cmp.ge $p1,p0 = \#3,r1$(ν_1=4)
cmp.ge $p1,p0 = \#3,r1$ (ν_1=7)	add $r1 = \#1,r1$ (ν_1=6)
1=read($p1$)	0=read($p1$)
add $r1 = \#1,r1$ (ν_1=7)	st8 $[r4] = r1$ (ν_4=1104, ν_1=7)
cmp.ge $p1,p0 = \#3, r1$ (ν_1=21)	
0=read($p1$)	
st8 $[r4] = r1$ (ν_4=101, ν_1=17)	

Fig. 2. Computation 1 (arbitrary values are associated with instructions)

p_1	p_2
ld8 $r1 = [r3]$ (ν_3=1284, $\nu_{(1284)}$=7)	ld8 $r1 = [r3]$ (ν_3=1027, $\nu_{(1027)}$=12)
add $r1 = \#1,r1$ (ν_1=15)	add $r1 = \#1,r1$ (ν_1=4)
cmp.ge $p1,p0 = \#3,r1$ (ν_1=7)	cmp.ge $p1,p0 = \#3,r1$(ν_1=6)
1=read($p1$)	0=read($p1$)
add $r1 = \#1,r1$ (ν_1=7)	st8 $[r4] = r1$ (ν_4=1104, ν_1=7)
cmp.ge $p1,p0 = \#3, r1$ (ν_1=21)	
0=read($p1$)	
st8 $[r4] = r1$ (ν_4=101, ν_1=17)	

Fig. 3. Computation 2

p_1	p_2
ld8 $r1 = [r3]$ (ν_3=1284, $\nu_{(1284)}$=2)	ld8 $r1 = [r3]$ (ν_3=1280, $\nu_{(1280)}$=5)
add $r1 = \#1,r1$ (ν_1=2)	add $r1 = \#1,r1$ (ν_1=5)
cmp.ge $p1,p0 = \#3,r1$ (ν_1=3)	cmp.ge $p1,p0 = \#3,r1$(ν_1=6)
1=read($p1$)	0=read($p1$)
add $r1 = \#1,r1$ (ν_1=3)	st8 $[r4] = r1$ (ν_4=1284, ν_1=0)
cmp.ge $p1,p0 = \#3, r1$ (ν_1=4)	
0=read($p1$)	
st8 $[r4] = r1$ (ν_4=1280, ν_1=1)	

Fig. 4. Computation 3

Example - An instance of the Architecture-Computation Problem: Let $IP = \{prog_1, prog_2\}$ where $prog_1$ and $prog_2$ are both the program in Figure 1. When applied to our running example, the Architecture-Computation problem becomes: "Could Computation 1 (respectively, 2 or 3) in Figure 2 (respectively, 3 or 4) arise from running IP on an Itanium multiprocessor?"

Two major steps are required to answer the Architecture-Computation question. The first is to determine if each processor's individual computation (ignoring the associated values) could have arisen from its program. If the answer is yes, then the second step is to determine whether or not the individual program sequences could have interacted in such a way under architecture $\mathcal{A}$ as to produce the values associated with each of the instructions in the computation.

Step 1: Program graphs and computational forms. To answer the first question, each processor's program is modelled as a directed graph, such that any computation of an individual program must correspond to some path through its graph. Specifically, for any individual program *prog* associate a directed node-labeled graph called the *program-graph of* prog, denoted $\mathcal{G}(prog)$, as follows.

Nodes of $\mathcal{G}(prog)$: For every operational instruction `inst`, there is a vertex $\eta($`inst`$)$ with label `inst`. For every conditional branch instruction `br` of the form "br(*cond*) label:", there are two vertices, $\eta_0($`br`$)$ with label 0=read(*cond*) and $\eta_1($`br`$)$ with label 1=read(*cond*). Henceforth, the labels of vertices of $\mathcal{G}(prog)$ are referred to as *node-labels* to distinguish them from the labels of instructions in *prog*. Notice that unconditional branches do not correspond to a node.

Directed edges of $\mathcal{G}(prog)$: For any instruction `inst` in *prog*, associate a set of vertices, vertices(`inst`), by:

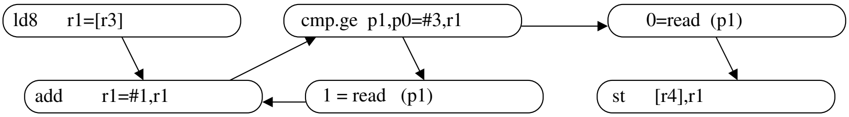
- for an operational instruction `inst`, vertices(`inst`) $= \{\eta(\texttt{inst})\}$,
- for a conditional branch instruction `br`, vertices(`br`) $= \{\eta_0(\texttt{br}), \eta_1(\texttt{br})\}$,
- for an unconditional branch instruction, say "br continue-here", then vertices$(brcontinue - here)$ = vertices$(\widehat{\texttt{inst}})$ where $\widehat{\texttt{inst}}$ is the instruction with label "continue-here:".

Denote by `p-succ(inst)` the instruction in *prog* that follows `inst`. For each vertex η in $\mathcal{G}(prog)$, define `g-succ(`η`)` by:

- for an operational instruction, `inst`,
  `g-succ(`η`(inst))` = vertices(`p-succ(inst)`)
- for a conditional branch instruction, `br`= br(*cond*) label", where $\widehat{\texttt{inst}}$ is the instruction with label "label:", `g-succ(`η_0`(br))` = vertices(`p-succ(br)`) and `g-succ(`η_1`(br))` = vertices$(\widehat{\texttt{inst}})$.

Finally, for each vertex η in $\mathcal{G}(prog)$, there is a directed edge from η to every vertex in `g-succ(`η`)`.

Example - Program graph for an Itanium program: The program graph of the Itanium individual program of Figure 1 is:



Let `start` be the first instruction in an individual program *prog*. An *individual program sequence for* prog is the sequence of the node-labels on any (possibly non-simple) directed path in $\mathcal{G}(prog)$ that begins with any vertex in vertices(`start`). Let $P = \{prog_1, prog_2, ..., prog_n\}$ be a multiprogram. Then *CF*= $\{prog\text{-}seq_1, prog\text{-}seq_2, ..., prog\text{-}seq_n\}$ is a *computational form of P* if and only if for $1 \le i \le n$, $prog\text{-}seq_i$ is an individual program sequence for $prog_i$. A computation C *agrees with CF* if, for each i, the ith sequence of instructions in C, (ignoring values) is the same as $prog\text{-}seq_i$. From these definitions we have:

Claim. A multiprocessor computation can be a computation of a multiprogram P only if it agrees with a computational form of P.

prog-seq$_1$		*prog-seq*$_2$	
ld8	$r1$ = [$r3$]	ld8	$r1$ = [$r3$]
add	$r1$ = #1,$r1$	add	$r1$ = #1,$r1$
cmp.ge	$p1$,$p0$ = #3,$r1$	cmp.ge	$p1$,$p0$ = #3,$r1$
1=read($p1$)		0=read($p1$)	
add	$r1$ = #1,$r1$	st8	[$r4$] = $r1$
cmp.ge	$p1$,$p0$ = #3,$r1$		
0=read($p1$)			
st8	[$r4$] = $r1$		

Fig. 5. A computational form

Example - Computational form for an Itanium 2-processor multiprogram: CF= {*prog-seq*$_1$, *prog-seq*$_2$}, where *prog-seq*$_1$ and *prog-seq*$_2$ are given in Figure 5, is a computational form of the Itanium multiprogram IP in the running example.

Notice that a computational form contains only operational instructions; there are no branch instructions. In the program graph each conditional branch instruction is replaced by one node for each of the two possible outcomes. The outgoing edges of these nodes lead to the next instruction that is correct given the value of the condition. Each unconditional branch is replaced only by edges.

Example - Agreement of computations with an Itanium computational form: Both computations in Figures 3 and 4 agree with the computational form for IP in Figure 5 but the computation in Figure 2 does not because the cmp.ge and add instructions of p_2 are in the opposite order from the order of the add and cmp.ge node-labels in the program graph.

Step 2: From computational forms to computations: To answer the second question, the rules of interaction for the machine with architecture $\mathcal{A}$ are modelled as a collection of constraints on various subsets of the instructions of a computation. For each subset there is a specified partial order, for which there must be a total order extension that is *valid* (to be defined next). Furthermore there may be agreement properties required between some of the total order extensions.

Validity

Meaning of instructions: To define validity, each entry in a computational form (i.e. each node-label, ηl) is assigned a *meaning*, denoted $\mathcal{M}(\eta l)$, by mapping it to a short program that uses only read, write, arithmetic and logic operations on variables. Specifically, the programming language `Trivial`:

- has two kinds of objects: single-reader/single-writer atomic variables and multi-reader/multi-writer atomic variables, and
- supports five types of operations: read and write operations on atomic variables, arithmetic and logic operations on single-reader/single-writer atomic variables, and "if-conditional-then-operations-else-operations"

Recall that each node-label, ηl, is either an operational instruction or 0=read(*cond*), or 1=read(*cond*). For $\eta l \in \{0 = \text{read}(\textit{cond}), 1 = \text{read}(\textit{cond})\}$ define $\mathcal{M}(\eta l)$ to be the identity function. If ηl is an operational instruction

$\mathcal{M}(\textit{ ld8 r1=[r3]})$	$\mathcal{M}(\textit{add r1=\#1,[r3]})$	$\mathcal{M}(\textit{cmp.eq p1,p2=r1,r3})$	$\mathcal{M}(\textit{st8 [r4]=r5})$
$\nu_1 = \mathrm{read}(r3)$ $\nu_2 = \mathrm{read}(\nu_1)$ $\mathrm{write}(r1, \nu_2)$	$\nu_1 = \mathrm{read}(r3)$ $\nu_2 = \mathrm{read}(\nu_1)$ $\nu_3 = 1 + \nu_2$ $\mathrm{write}(r1, \nu_3)$	$\nu_1 = \mathrm{read}(r2)$ $\nu_2 = \mathrm{read}(r3)$ if ($\nu_1 {==} \nu_2$) then { $\mathrm{write}(p2, 1)$ $\mathrm{write}(p1, 0)$ } else { $\mathrm{write}(p2, 0)$ $\mathrm{write}(p1, 1)$ }	$\nu_1 = \mathrm{read}(r5)$ $\nu_2 = \mathrm{read}(r4)$ $\mathrm{write}(mem(\nu_2), \nu_1)$
(a)	(b)	(c)	(d)

Fig. 6. Meaning of four Itanium instructions

then $\mathcal{M}(\eta l)$ is defined to be the short `Trivial` program that re-expresses the semantics of the instruction as specified by the instruction manual of $\mathcal{A}$.

Example - Meaning of some Itanium operational instructions: Figure 6 gives four examples. These can be extracted from the Itanium Instruction Set Reference manual [13].

Recall that a completed instruction, `inst`, has values associated with each register or shared memory location that it reads, and the variables in $\mathcal{M}$(`inst`) are assigned by reading, writing, or performing arithmetic/logic operations on the values in these registers. If the register values in the completed instruction `inst` are used to compute the values of the corresponding variables in $\mathcal{M}$(`inst`), the resulting sequence of `Trivial` operations is called the *derived completed meaning of* `inst`, denoted $\mathcal{DM}$(`inst`). Because the operations are completed, the conditional operations in this sequence can be resolved, so the derived completed meaning of `inst` is reduced to a sequence I of completed read, write and arithmetic/logic operations. Define the *remote derived completed meaning of* `inst`, denoted $\mathcal{RDM}$(`inst`), to be the subsequence of I consisting of only the completed write operations to shared memory locations.

Example - Derived completed meanings: The derived completed meaning of the completed instruction ld8 $r1 = [r3]$ (ν_{r3}=1284, $\nu_{m(1284)}$=7) is "1284 = read($r3$), 7 = read(1284), write($r1, 7$)" whereas its remote derived completed meaning is the empty list. For st8 $[r4] = r5$ (ν_{r4}=1280, ν_{r5}=1) the derived completed meaning is 1 = read($r5$), 1280 = read($r4$), write(mem(1280), 1) and the remote derived meaning is just write(mem(1280), 1).

Definition of validity of a sequence of completed instructions: Let $S = s_1, s_2, \ldots, s_k$ be any sequence of completed instructions of a computation. The *computed meaning of S* is the sequence formed by concatenating $\mathcal{DM}(s_i)$ for i from 1 to k. For a given individual program p, the *computed meaning of S for p* is the sequence formed by concatenating as follows. Replace each s_i that is an instruction by p with $\mathcal{DM}(s_i)$; replace each s_i that is an instruction by a processor different from p with $\mathcal{RDM}(s_i)$. Notice that both the computed meaning of S and the computed meaning of S for p are sequences of read, write and arithmetic/logic operations on atomic variables, so it is straightforward to determine if such a sequence is valid. Specifically, it is *valid* if each read returns the value of the last write to the same variable and all arithmetic/logic operations on variables are correct. Finally, a sequence S of completed instructions from a computation is *valid (respectively, valid for p)* if the computed meaning of S (respectively, the computed meaning of S for p) is valid.

Partial orders and memory consistency models: The final task is to capture the rules that govern executions of multiprograms under architecture $\mathcal{A}$ as defining properties of the computations that can be produced. These properties, collectively called a memory consistency model, are different for every multiprocessor machine, but have the same general structure. They are expressed as a collection of partial orders relations on the completed instructions of a computation. Each of these partial orders is required to have a total order extension that

- is valid and
- shares some agreement properties with other partial orders.

Example - The Itanium memory consistency model[2]: Let $I(\mathcal{C})$ be the set of all completed instructions in a computation C. $I(\mathcal{C})|p$ denotes the subset of $I(\mathcal{C})$ in processor p's program sequence; $I(\mathcal{C})|x$ denotes the subset whose meaning contains a read or write operation on (register or shared memory) variable x; and $I(\mathcal{C})|br$ is the subset whose meaning is 0=read(*cond*) or 1=read(*cond*), where cond is a (condition) register. $I(\mathcal{C})|r$ denotes the subset containing only the instruction instances that contain a read operation on a shared memory variable; $I(\mathcal{C})|w$ the subset containing only the instruction instances that contain a write operation on a shared memory variable. The relation $(I(\mathcal{C}), \xrightarrow{prog})$, called *program order*, is the set of all pairs (i, j) of completed instructions that are in the same individual computation of C and such that i precedes j in that sequence. For any partial order relation $(I(\mathcal{C}), \xrightarrow{y})$, the notation $i \xrightarrow{y} j$ is used interchangeably with $(i, j) \in (I(\mathcal{C}), \xrightarrow{y})$.

Define the following partial orders:

- Local dependence order $(I(\mathcal{C}), \xrightarrow{dep})$: For $i, j \in I(\mathcal{C})|p$, $i \xrightarrow{dep} j$ if $i \xrightarrow{prog} j$ and either
  Register: $i, j \in I(\mathcal{C})|x$, where x is a register, or
  Branch: $i \in I(\mathcal{C})|br$.
- Orderable order $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{ord})$ for each $p \in P$: $i \xrightarrow{ord} j$ if $i, j \in I(\mathcal{C})|p \cup I(\mathcal{C})|w$ and $i \xrightarrow{prog} j$ and $i, j \in I(\mathcal{C})|x$ and $(i \in I(\mathcal{C})|w$ or $j \in I(\mathcal{C})|w)$

Itanium memory consistency definition: A computation C satisfies Itanium consistency if for each $p \in P$, there is a total order $\xrightarrow{S}$ of the operations $I(\mathcal{C})|p \cup I(\mathcal{C})|w$ that is valid for p, such that

1. $(I(\mathcal{C})|p, \xrightarrow{dep}) \subseteq (I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S})$, (Local requirement) and
2. $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{ord}) \subseteq (I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S})$, (Orderable requirement) and
3. If $i_1, i_2 \in I(\mathcal{C})|x|w$ and $i_1 \xrightarrow{S} i_2$ then $i_1 \xrightarrow{S} i_2$, $\forall q \in P$, (Same Memory agreement) and

[2] Itanium provides a rich instruction set, which includes semaphore and fence instructions. The definition formulated here ignores acquire, release, and fence instructions. The development and proofs for the general Itanium definition are elsewhere [7].

4. There does not exist a cycle of $i_1, i_2 \ldots i_k \in I(\mathcal{C})|w$ where $i_j \in I(\mathcal{C})|p_j, \forall j \in \{1, 2, \ldots k\}$ and $k \leq n$ such that: $i_k \xrightarrow{S_1} i_1$, and $i_1 \xrightarrow{S_2} i_2$, and $i_2 \xrightarrow{S_3} i_3 \ldots$ and $i_{k-1} \xrightarrow{S} i_k$ (Cycle-free agreement)

Define Itanium-Dep (Itanium minus dependence) to be identical to the preceding definition, without the Local requirement.

Example - Itanium-Dep and Itanium consistency: Computation 3 of Figure 4 satisfies Itanium-Dep. The following sequences satisfy the Orderable requirement, the Same Memory agreement, and Cycle-free agreement properties. show that Computation 3 satisfies Itanium-Dep requirements and agreement properties:

S_{p_1} : ld8 $r1$ = [$r3$] (ν_{r3}=1284, $\nu_{m(1284)}$=2) $\xrightarrow{S\ _1}$ st8$_{p_2}$ [$r4_{p_2}$] = $r1_{p_2}$ ($\nu_{r4\ _2}$=1284, $\nu_{r1\ _2}$=0) $\xrightarrow{S\ _1}$ st8 [$r4$] = $r1$ (ν_{r4}=1280, ν_{r1}=2) $\xrightarrow{S\ _1}$ add $r1$ = #1,$r1$ (ν_{r1}=2) $\xrightarrow{S\ _1}$ cmp.ge $p1$,$p0$ = #3,$r1$ (ν_{r1}=3) $\xrightarrow{S\ _1}$ 1=read($p1$) $\xrightarrow{S\ _1}$ add $r1$ = #1,$r1$ (ν_{r1}=3) $\xrightarrow{S\ _1}$ cmp.ge $p1$,$p0$ = #3, $r1$ (ν_{r1}=4) $\xrightarrow{S\ _1}$ 0=read($p1$)

S_{p_2} : st8 [$r4$] = $r1$ (ν_{r4}=1284, ν_{r1}=0, $\nu_{m(1284)}$←0) $\xrightarrow{S\ _2}$ ld8 $r1$ = [$r3$] (ν_{r3}=1280, $\nu_{m(1280)}$=5, ν_{r1}←5) $\xrightarrow{S\ _2}$ add $r1$ = #1,$r1$ (ν_{r1}=5 ν_{r1}←6) $\xrightarrow{S\ _2}$ cmp.ge $p1$,$p0$ = #3, $r1$(ν_{r1}=6, ν_{p1}←0) $\xrightarrow{S\ _2}$ 0=read($p1$) $\xrightarrow{S\ _2}$ st8$_{p_1}$ [$r4_{p_1}$] = $r1_{p_1}$ (ν_{r4}=1280, $\nu_{r1\ _1}$= 1, $\nu_{m(1280)}$←1)

To show that these sequences are valid we provide `Trivial` instructions for the derived completed meaning or remote derived completed meaning for each instruction in the sequence above. Square brackets, [ ], delineate the operations for each instruction.

Validity Sequence for S_{p_1}: [1284 = read($r3$), 2 = read(1284), write($r1, 2$)] [write(1284, 0)] [2 = read($r1$), 1280 = read($r4$), write(1280, 2)] [2 = read($r1$), 3 = 1 + 2, write($r1, 3$)] [3 = read($r1$), if (#3 *geq* > 3) then write($p1, 1$) else write($p1, 0$)] [1=read($p1$)] [3 = read($r1$), 4 = 1 + 3, write($r1, 4$)] [4 = read($r1$), if (#3 *geq* > 4) then write($p1, 1$) else write($p1, 0$)] [0=read($p1$)]

Validity Sequence for S_{p_2}: [0 = read($r1$), 1284 = read($r4$), write(1284, 0)] [1280 = read($r3$), 5 = read(1280), write($r1, 5$)] [5 = read($r1$), 6 = 1 + 5, write($r1, 6$)] [6 = read($r1$), if (#3 *geq* 6) then write($p1, 1$) else write($p1, 0$)] [0=read($p1$)] [write(1280, 1)]

However, Computation 3 does not satisfy Itanium because it must extend Local dependence order, which requires that the st8 instruction in each sequence follows the add instructions in that sequence, which would break validity. A computation that is similar to Computation 3 except that the st8 instruction by p_1 has values (ν_{r4}=1280, ν_{r1}=3) and the st8 instruction by p_2 has values (ν_{r4}=1280, ν_{r1}=6) satisfies Itanium.

3 Consequences of Ignoring Dependence Order

Ignoring register and control dependences can lead to erroneous conclusions about the capabilities or limitations of the architecture under consideration. This section illustrates such an error using a simple producer-consumer example and the Itanium architecture.

3.1 A Simple Producer-Consumer Multiprocessor System

Informally, the simple producer-consumer (SPC) multiprocessor system defined here has one producer program, producing items, which are consumed by a single consumer program. The producer and consumer take turns in producing and consuming items. This specialized system can be defined using the framework of Section 2 as follows:[3]

Objects: A *SPC object* X supports two operations P-write(X,ι) and ι=C-read(X). The semantics of these operations are exactly the same as the read and write operations on an atomic variable. However, the validity requirement is substantially stronger.
Validity: A sequence of (C-read and P-write) operations on SPC object X is valid if: (1) it starts with a P-write operation, (2) alternates between C-read and P-write operations, (3) each C-read returns the value written by the immediately preceding P-write operation, and (4) it has an equal number of P-write and C-read operations.[4]

Programs: There are two programs: the producer repeatedly 'P-writes' the SPC object, and the consumer repeatedly 'C-reads' it, for a specified (possibly infinite) number of rounds.

SPC System: The SPC multiprocessor system consists of one SPC object X and a multiprogram of one producer and one consumer. The memory consistency model is sequential consistency[17].

3.2 Proving Relationships Between Multiprocessor Systems

Given a multiprocessor system with multiprogram P, objects J, and a memory consistency model M, we call (P, J) a *multiprocess* and when J consists entirely of atomic variables and registers, we call (J, M) an M *platform.*

The *specified* SPC multiprocess (P, J) gives rise to a set of allowable (specification) computations, $\mathcal{C}$ in Figure 7, under sequential consistency. We model the execution of the *specified* multiprocess on the *target* Itanium architecture, as a transformation, τ. This transformation replaces the specified objects, operations, and memory consistency model respectively with target (Itanium) objects, instructions, and memory consistency model. Specifically, an operation on a specified object is *transformed* to the target objects by providing a subroutine for the operation's invocation where this subroutine uses only instruction invocations on the target objects, $\widehat{J}$. If the specified operation returns an output, then the subroutine must return a value of the same type as this output. An object in J is

[3] A general producer-consumer system definition may allow several producers and consumers and more complex shared objects such as queues, where producers enqueue items and consumers dequeue them. This very restricted definition (two programs and a queue of size one) suffices for this section. Furthermore, without exploiting additional Itanium synchronization mechanisms, such as fences, acquires, or releases, a solution for the general case is not possible.

[4] Note that strict alternation between P-write and C-read operations is not necessary at lower levels. This specification-level requirement can be satisfied at an implementation-level as long as operations appear as if they do alternate.

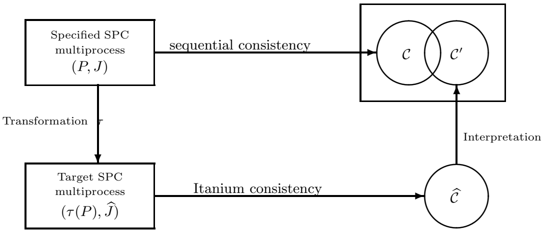


Fig. 7. Program transformation and computation interpretation. If $\mathcal{C}' \subseteq \mathcal{C}$, then the transformation is an implementation.

transformed to the target object(s) in $\widehat{J}$ by transforming each of its operations. A transformation of each of the objects of a specified multiprocess to objects of the target multiprocess can be naturally extended to a *program transformation* by replacing each operation invocation in the specified multiprogram with the subroutine for that operation invocation. The transformed multiprogram together with the target objects $(\tau(P), \widehat{J})$ comprise the *transformed multiprocess.*

Hence, each P-write and C-read operations will be transformed to subroutines that only make use of Itanium instructions. The transformed multiprocess gives rise under Itanium consistency to an allowable set of computations, $\widehat{\mathcal{C}}$. Any computation in $\widehat{\mathcal{C}}$ can be *interpreted* as a computation of the specified multiprogram by attaching to each operation invocation of the specified multiprogram the value returned by the corresponding subroutine. Thus, the set $\hat{C}$ of computations of the target system provides a set of interpreted computations $\mathcal{C}'$ of the specified multiprocess.

A transformation of a specified multiprocess is called an *implementation* of the specified system, if, informally, (in Figure 7) the set $\mathcal{C}'$ of computations produced by traveling the long way around is a non-empty subset of the computations $\mathcal{C}$ allowed by the specified multiprocessor system. The formal details, including stronger (than 'implementations') relations between systems, are elsewhere [5].

3.3 In the Presence of Dependence

Use p and c to refer to the producer and consumer programs, respectively. For the transformation of the SPC multiprocess given in Figure 8, consider the last load in two consecutive iterations of p. We use dependence order and validity to prove that at least one copy of c's store at line 10 occurs in p's sequence $(I(\mathcal{C})|p \cup I(\mathcal{C})|w, \xrightarrow{S})$ between these two loads. Also consider the last load in two consecutive iterations of c. We use dependence order and validity to prove that at least one copy of p's store at line 4 occurs in c's sequence $(I(\mathcal{C})|c \cup I(\mathcal{C})|w, \xrightarrow{S})$ between these loads. Orderable order and the same

Produce

Pseudo Code	$\tau((P{-}write(\mathrm{X}, item)))$:		
	0	mov r2=Q	; r2 contains the address of Q
while (Q ≠ 0) wait;	1 L:	ld8 r1=[r2]	; load r1 with Q
	2	cmp.neq p1,p0 = r1,r0	; is r1 == 0?
	3	(p1) br L:	; If not, go back to L
Q ⟵ item	4	st8 [r2]= item	; store item into Q

Consume

Pseudo Code	$\tau((C - read(\mathrm{X})))$ returns item in $r3$:		
	5	mov r2=Q	; r2 contains the address of Q
while (Q = 0) wait;	6 M:	ld8 r1=[r2]	; load r1 with Q
	7	cmp.eq p1,p0 = r1, r0	; is r1 == 0?
	8	(p1) br M:	; If so, go back to M
item ⟵ Q	9	ld8 r1=[r2]	; load r1 with Q
Q ⟵ 0	10	st8 [r2]=r0	; store 0 into Q
	11	mov r3, r1	; put consumed item into r3

Fig. 8. A transformation of the specified SPC object, X, on Itanium, using the target objects $\{Q, r1_p, r2_p, p1_p, r1_c, r2_c, r3_c, p1_c\}$

memory agreement property force the stores to strictly alternate in each sequence, ensuring that the c loads every produced value. The proof of the following theorem is long and tedious and is elsewhere [6].

Theorem 1. *Transformation τ in Figure 8 is an implementation of the SPC system on an Itanium platform.*

3.4 In the Absence of Dependence

While the SPC system has an implementation on an Itanium platform, we show in this section that such an implementation does not exist on an Itanium-Dep platform.

First of all, we show how any transformation respecting the pseudo code in Figure 8 fails under only Itanium-Dep. Specifically, in the C-read pseudo code, the Orderable order does not always require program order between two load instructions, even if they access the same atomic variable (Q in this case). The consumption load (load of Q in 'item ← Q') may be ordered in S_c before any load instruction resulting from the spin-loop, including the load that writes a value that consequently ends the loop. This load signals to the consumer to proceed with consumption. However under Itanium-Dep, the consumer can proceed to consumption before even performing any loads or checking the condition in the spin-loop. That is, a consumer may consume a never-produced or a previously consumed item. This 'early' load decides the value to be returned in the interpretation by the corresponding C-read. Any attempt to construct a valid total order for the producer-consumer object X will fail. The failure of this algorithm is general as we argue next.

Theorem 2. *There does not exist an implementation of the SPC system on an Itanium-Dep platform.*

Proof: Assume there exists an implementation β of the SPC system on an Itanium-Dep platform. Then the interpretation of any computation of the transformed SPC multiprocess will be valid for the SPC object X. By this validity requirement, the β(C-read(X)) subroutine does not return before it finds a new produced and never consumed before item. Hence, c must *wait* in β(C-read(X)) until a new produced item is available for consumption. In Itanium-Dep, the only way by which c waits for p is through the use of a spin-loop. The spin-loop has at least one load instruction, *spin-load*.

Furthermore, there has to be a load instruction after which the value to be returned by β(C-read(X)) will be decided and does not change until β(C-read(X)) returns. Call this load the *consumption load*. Obviously, the consumption load must follow in program order some of the spin-loads. We argue that Itanium-Dep can give rise to a computation that behaves as if the consumption load is completed before the spin-loop starts.

If between a spin-load and the following (in program order) consumption load there are any stores, then these stores cannot be to the same variable that the consumption load is accessing, otherwise the communicated produced item may be changed by the consumer and lost. A store that deterministically leaves the value of an atomic variable, say v, unchanged does not necessarily write the same value back to v in Itanium-Dep. Writing the same value stored in v back to v (something of the form: $v \leftarrow v$) requires a load that precedes in program order the store. What we have now in $\mathit{prog\text{-}seq}_{\beta(c)}$ is:

ld8 r_1=[r] ; begin spin-loads, register r has the address of v
ld8 r_2=[r]
. . .
ld8 r_k=[r] ; another spin-load (not necessarily the last of the loads in the loop)

ld8 r_{k+1}=[r] ; load resulting from writing v to v
st8 [r]=r_{k+1} ; writing v back to v
. . .
ld8 r_{k+i}=[r]; the consumption load

Itanium-Dep is insufficient to enforce any ordering between ld8 r_{k+1}=[r] and any of the k spin-loads in $(I(\mathcal{C})|c \cup I(\mathcal{C})|w, \xrightarrow{S})$. Hence, it is always possible for r_{k+1} to be assigned a never-produced (the item is not the result of β(P-write(X, ι))) item or an item that has already been consumed.

Finally, any store to a different variable between (in program order) the k spin-loads and the consumption load cannot restore the lost program order in S_c between the k spin-loads and the consumption load. That is, consumption can happen earlier than it should be, and we can end up with an invalid sequence of P-write and C-read operations. Hence, the transformation β cannot be an implementation, since in Figure 8, there is at least one computation in $\mathcal{C}'$ that cannot be in $\mathcal{C}$. ■

References

1. A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. on Parallel and Distributed Systems*, 14(5):502–515, 2003.

2. H. Attiya and R. Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proc. 6th Int'l Symp. on Parallel Algorithms and Architectures*, pages 157–166, 1994. Technical Report LPCR 9411, Computer Science Department, Technion.
3. P. Chatterjee and G. Gopalakrishnan. Towards a formal model of shared memory consistency for intel itaniumtm. In *Proc. 2001 IEEE International Conference on Computer Design (ICCD)*, pages 515–518, Sept 2001.
4. M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.
5. L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. on Computer Systems.* In Press.
6. L. Higham, L. Jackson, and J. Kawash. Capturing register and control dependence in memory consistency models with applications to the Itanium architecture. Technical Report 2006-837-30, Department of Computer Science, The University of Calgary, July 2006.
7. L. Higham, L. Jackson, and J. Kawash. Programmer-centric conditions for Itanium memory consistency. Technical Report 2006-838-31, Department of Computer Science, The University of Calgary, July 2006.
8. L. Higham and J. Kawash. Tight bounds for critical sections on processor consistent platforms. *IEEE Trans. on Parallel and Distributed Systems.* In Press.
9. L. Higham and J. Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. 1997 Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, December 1997.
10. L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.
11. L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int'l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.
12. Intel Corporation. A formal specification of the intel itanium processor family memory ordering. http://www.intel.com/, Oct 2002.
13. Intel Corporation. *Intel Architecture Architecture Software Developer's Manual: Volume 3: Instruction Set Reference.* Intel Corporation, Oct. 2002.
14. Intel Corporation. Intel itanium architecture software developer's manual, volume 2: System architecture. http://www.intel.com/, Oct 2002.
15. L. Jackson. Complete framework for memory consistency with applications to Itanium multiprocessors. Ph.D. Dissertation, in preperation.
16. R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla, 2003.
17. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.
18. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proc. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2003.

Conflict Detection and Validation Strategies for Software Transactional Memory*

Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{spear, vmarathe, scherer, scott}@cs.rochester.edu

Abstract. In a software transactional memory (STM) system, *conflict detection* is the problem of determining when two transactions cannot both safely commit. *Validation* is the related problem of ensuring that a transaction never views inconsistent data, which might potentially cause a doomed transaction to exhibit irreversible, externally visible side effects. Existing mechanisms for conflict detection vary greatly in their degree of speculation and their relative treatment of read-write and write-write conflicts. Validation, for its part, appears to be a dominant factor—perhaps *the* dominant factor—in the cost of complex transactions.

We present the most comprehensive study to date of conflict detection strategies, characterizing the tradeoffs among them and identifying the ones that perform the best for various types of workload. In the process we introduce a lightweight heuristic mechanism—the *global commit counter*—that can greatly reduce the cost of validation and of single-threaded execution. The heuristic also allows us to experiment with *mixed invalidation*, a more opportunistic interleaving of reading and writing transactions. Experimental results on a 16-processor SunFire machine running our RSTM system indicate that the choice of conflict detection strategy can have a dramatic impact on performance, and that the best choice is workload dependent. In workloads whose transactions rarely conflict, the commit counter does little to help (and can even hurt) performance. For less scalable applications, however—those in which STM performance has traditionally been most problematic—it can improve transaction throughput many fold.

1 Introduction

Thirty years of improvement in the speed of CMOS uniprocessors have recently come to an end. In the face of untenable heat dissipation and waning gains in ILP, hardware vendors are turning to multicore, multithreaded chips for future speed improvements. As a result, concurrent programming is suddenly on the critical path of every major software vendor, and traditional lock-based programming methodologies are looking decidedly unattractive. A growing consensus views transactional memory (TM) [12], implemented in hardware or software, as the most promising near-term technology to simplify the construction of correct multithreaded applications. Transactions eliminate

* This work was supported in part by NSF grants CCR-0204344 and CNS-0411127, by financial and equipment grants from Sun Microsystems Labs, and by financial support from Intel.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 179–193, 2006.

the semantic problems of deadlock and priority inversion. They also address the performance problems of convoying and of preemption or page faults in critical sections. Perhaps most important, they eliminate the need to choose between the conceptual simplicity of coarse grain locks and the concurrency of fine grain locks.

Unfortunately, hardware implementations of transactional memory have yet to reach the market, and the performance of current software transactional memory (STM) systems leaves much to be desired. In recent work we introduced a comparatively lightweight system, RSTM, and carefully analyzed its costs [19]. In addition to copying overhead, which appears to be unavoidable in a nonblocking STM system, we found the two principal sources of overhead to be *bookkeeping* and *incremental validation*. Bookkeeping serves largely to implement *conflict detection*—that is, to identify pairs of concurrent transactions which, if permitted to commit, would not be linearizable [13]. Validation serves to ensure that transactions never see or make decisions based on inconsistent data; we use the term "incremental" to indicate strategies in which the overhead of validation is proportional to the number of objects previously accessed.

Two concurrent transactions are said to conflict if they access the same object and at least one of them modifies that object. When an STM system identifies a conflict, it typically allows one transaction to continue, and delays or aborts the other. If the system is nonblocking, the choice may be based on a built-in policy (as, for example, in the lock-free OSTM [3]), or it may be deferred to a separate *contention manager* (as, for example, in the obstruction-free DSTM [11]). The design of contention managers has received considerable attention in recent years [4, 5, 6, 22, 23, 24]. Conflict detection and validation have not been as thoroughly or systematically studied.

Conflict detection. An STM system may notice potential conflicts early in the life of the conflicting transactions, or it may delay such notice until one of the transactions attempts to commit. The choice may depend on whether the conflict is between two writers or between a reader and a writer. In the latter case, it may further depend on whether the reader or the writer accesses the object first. If transactions S and T conflict, aborting S early may avoid fruitless further computation. In general, however, there is no way to tell whether T will ever commit; if it doesn't, then S might have been able to do so if it had been permitted to continue.

We have recently studied the semantics of several alternative strategies for conflict detection, and have identified existing systems that implement these strategies [25]. In this study we suggested that it might make sense to detect write-write conflicts early (since at most one of the conflicting transactions can ever commit), but read-write conflicts late (since both may commit if the reader does so first). We refer to this hybrid strategy as *mixed invalidation*; to the best of our knowledge, it has not been explored in any prior TM system.

Validation. Since a transaction that commits successfully has no visible side effects prior to the commit, it is tempting to assume that an aborted transaction will have no visible effects whatsoever. Problems arise, however, in the presence of transaction conflicts. Suppose, for example, that `f()` is a virtual method of class `A`, from which are derived subclasses `B` and `C`. Suppose further that while `B.f()` can safely be called in transactional code, `C.f()` cannot (perhaps it performs I/O, acquires a lock, or modifies global data under the assumption that some lock is already held). Now suppose that transaction T reads objects x and y. Object y contains a reference to an object of class `A`. Object x contains information implying that the reference in y points to a transaction-safe `B` object. Unfortunately, after T reads x but before it reads y, another transaction modifies both objects, putting a `C` reference into y and recording this fact in x. Because x has been modified, T is doomed to abort. If it does not notice this fact right away, however, T may read the `C` reference in y and call its unsafe method `f()`.

While this example is admittedly contrived, it illustrates a fundamental problem: even in a typesafe, managed language, a transaction that is about to perform a potentially unsafe operation must verify the continued validity of any previously read objects on which that operation has a control or data dependence. Unfortunately, straightforward *incremental validation*—checking all previously read objects on each new object reference—leads to $O(n^2)$ total cost when opening n objects, an extraordinary burden for transactions that access many objects. Similarly, visible readers—which allow a writer to identify and explicitly abort the transactions with which it conflicts—incur very heavy bookkeeping and cache eviction penalties; in our experiments, for all but the largest transactions, these penalties, though linear, are worse than the quadratic cost of incremental validation.

Static analysis of data flow and safety may allow a compiler-based STM system to avoid validation in many important cases, but library-based STM has traditionally been stuck with one of two alternatives: (1) require the programmer to validate manually wherever necessary, or (2) accept the quadratic cost of incremental validation. Option (1), we believe, is unacceptable: identifying the places that require validation is too much to expect of the typical programmer. We prefer instead to find ways to avoid or reduce the cost of incremental validation.

Contributions. This paper makes two principal contributions. First, we present the most thorough evaluation to date of strategies for conflict detection, all in the context of a single STM system. We consider *lazy acquire*, in which conflicts are noticed only at commit time; *eager acquire*, in which conflicts are noticed as soon as two transactions attempt to use an object in incompatible ways; and *mixed invalidation*, in which conflicts are noticed early, but not acted upon until commit time in the read-write case. We also consider both *visible* and *invisible* readers. Invisible readers require less bookkeeping and induce fewer cache misses, but require that read-write conflicts be noticed by the reader. Visible readers allow such conflicts to be noticed by writers as well.

Second, we introduce a lightweight heuristic mechanism—the *global commit counter*—that eliminates much of the overhead of incremental validation. Specifically, we validate incrementally only if some other transaction has committed writes since the previous validation. In multithreaded experiments, the savings ranges from negligible in very short transactions to enormous in long-running applications (95% reduction in validation overhead for our RandomGraph "torture test"). Because it allows us to overlook the fact that a previously read object is being written by an as-yet-uncommitted transaction, the commit counter provides a natural approximation of mixed invalidation. It also allows us to notice when a transaction is running in isolation, and to safely elide bookkeeping, validation, and contention management calls. This elision dramatically reduces the cost of STM in the single-threaded case.

Section 2 provides an overview of our RSTM system, including a description of eager and lazy acquire, visible and invisible readers, and mixed invalidation. Section 3 then presents the global commit counter heuristic. Performance results appear in Section 4, related work in Section 5, and conclusions in Section 6.

2 Overview of RSTM

The Rochester Software Transactional Memory System (RSTM) is a fast, nonblocking C++ library that seeks to maximize throughput, provide a simple programming interface, and facilitate experimentation. To first approximation, its metadata organization (Figure 1) resembles that of DSTM [11], but with what the latter calls a "Locator" merged into the newest copy of the data. Detailed description can be found in a previous paper [19]; we survey the highlights here.

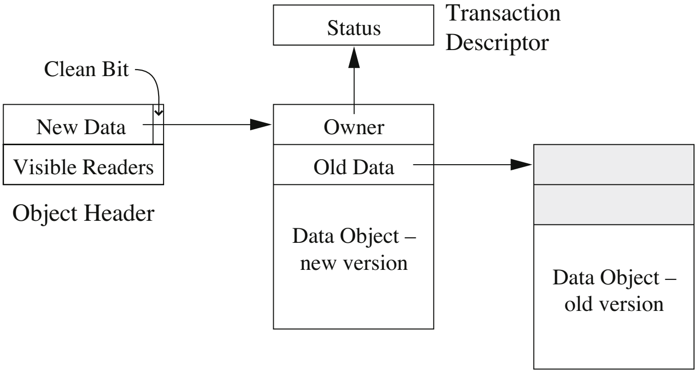


Fig. 1. RSTM metadata. Visible Readers are implemented as a bitmap index into a global table. Up to 32 concurrent transactions can read visibly, together with an unlimited number of invisible readers. The Clean Bit, when set, indicates that the new Data Object is valid; the Transaction Descriptor need not be inspected.

As in most other nonblocking STMs, an object is accessed through an *object header*, which allows transactions to identify the last committed version of the object and, when appropriate, the current speculative version. The metadata layout is optimized for read-heavy workloads; in the common case, the header points directly to the current version of the object. When an object is being written, one additional level of indirection is needed to reach the last committed version.

Each thread maintains a *transaction descriptor* that indicates the status (active/ committed/aborted) of the thread's most recent transaction, together with lists of objects opened (accessed) for reading and for writing. To minimize memory management overhead, descriptors are allocated statically and reused in the thread's next transaction. RSTM currently supports nested transactions only via subsumption in the parent.

Data object versions are dynamically allocated from a special per-thread heap with lazy generational reclamation. As in OSTM [2] or McRT [14], "deleted" objects are not reclaimed until every thread is known to have been outside any potentially conflicting transaction.

Acquisition. A transaction never modifies a data object directly; instead, it clones the object and makes changes to the copy. At some point between open time (initial access) and commit time, the transaction must *acquire* the object by making the object header point to the new version of the data (which in turn points to the old). Since each new version points to the transaction's descriptor, atomically CAS-ing the descriptor's status from active to committed has the effect of updating every written object to its new version simultaneously. Eager (open-time) acquire allows conflicts to be detected early. As noted in Section 1, the timing of conflict detection enables a tradeoff between, on the one hand, avoiding fruitless work, and, on the other, avoiding spurious aborts.

Reader visibility. The programmer can specify whether reads should be visible or invisible. If reads are visible, the transaction arbitrates for one of 32 visible reader tokens. Then, when it opens an object for reading, the transaction sets the corresponding bit in the object's visible reader bitmap. Thus while the system as a whole may contain an arbitrary number of threads, at most 32 of them can be visible readers concurrently (the rest can read invisibly). The bitmap is simpler and a little bit faster than an alternative mechanism we have described [19] that supports an arbitrary number of visible readers.

Before it can acquire an object for writing, a transaction must obtain permission from its contention manager to abort all visible readers. It performs these aborts immediately after acquisition. A transaction that has performed only visible reads is thus guaranteed that if it has not been aborted, all of its previously read objects are still valid. By contrast, as described in Section 1, an invisible reader must (absent static analysis) incrementally validate those objects on every subsequent open operation, at $O(n^2)$ aggregate cost.

In practice, visible readers tend to cause a significant increase in memory traffic, since the write by which a reader announces its presence necessarily evicts the object header from every other reader's cache. In several of our microbenchmarks, visible readers perform worse than invisible readers at all thread counts higher than one.

Mixed invalidation. If two transactions attempt to write the same object, one argument for allowing both to proceed (as in lazy acquire) holds that any execution history in which both remain active can, in principle, be extended such that either commits (aborting the other); there is no a priori way for an implementation to tell which transaction "ought" to fail. This is a weak argument, however, since both cannot succeed. When a reader and a writer conflict, however, there is a stronger argument for allowing them to proceed concurrently: both can succeed if the reader commits first. We therefore consider a *mixed invalidation* strategy [25] in which write-write conflicts are detected eagerly but read-write conflicts are ignored until commit time. The following section considers the implementation of mixed invalidation and a heuristic that cheaply approximates its behavior.

3 The Global Commit Counter Heuristic

As noted in Section 1, a transaction must validate its previously-opened objects whenever it is about to perform an operation that may be unsafe if the values of those objects are mutually inconsistent. We take the position that validation must be automatic—that it is unreasonable to ask the programmer to determine when it is necessary. In either case, the question arises: how expensive must validation be?

With visible readers, validation is very inexpensive: a reader need only check to see whether it has been aborted. With invisible readers and eager acquire, naive (incremental) validation takes time linear in the number of open objects. In a poster at PODC'04 [15], Lev and Moir suggested a heuristic that could reduce this cost in important cases. Specifically, they suggest per-object reader counters coupled with a global *conflict counter*. Readers increment and decrement the per-object counters at open and commit time, respectively. Writers increment the conflict counter whenever they acquire an object whose reader counter is nonzero. When opening a new object, a reader can skip incremental validation if the global conflict counter has not changed since the last time the reader checked it.

The conflict counter is a useful improvement over visible readers in systems like DSTM [11] and SXM [5], where visible readers require the installation of a new Locator and thus are very expensive. Unfortunately, every update of a reader counter will invalidate the counter in every other reader's cache, leading to cache misses at commit time even when there are no writers. In the absence of any contention, a transaction T_1 reading R objects will skip all validation but must perform $2R$ atomic increment/decrement operations. For each object that is also read by T_2, T_1 will incur at least one cache miss, regardless of whether the counter is stored with the object metadata or in a separate cache line.

We observe that if one is willing to detect read-write conflicts lazily, a more lightweight optimization can employ a global *commit counter* that records only the number of writer transactions that have attempted to commit. When a transaction acquires an object, it sets a local flag indicating that it must increment the counter before attempting to commit. Now when opening a new object, a reader can skip incremental validation if the global commit counter has not changed since the last time the reader checked it. If the counter has changed, the reader performs incremental validation.

In comparison to the Lev and Moir counter, this heuristic requires no atomic operations by readers, and the same amount of bookkeeping. A transaction T_1 that reads R objects will validate by checking the global counter R times. Reading the counter will only be a cache miss if a writing transaction commits during the execution of T_1, in which case an incremental validation is necessary. For a successful transaction T_1, the cost of validation with the global commit counter is a function of four variables: the number of objects read by T_1 (R), the number of writer transactions that commit during the execution of T_1 ($||\{T_w\}|| = W$), the cost of validating a single object (a cache hit and a single word comparison C_v, which we also use as the cost of detecting that the counter has not changed), and the cost of a cache miss (C_{miss}). Assuming that all R objects fit in T_1's cache, the baseline cost of incremental validation without the commit counter is $C_v \sum_{i=1}^{R} i = C_v \frac{R(R+1)}{2}$. Assuming a uniform distribution of writer commits across the duration of T_1, the cost of validation is the cost of W successful validations of $R/2$ objects, W cache misses, and $R - W$ successful checks of the global counter. For workload and machine configurations in which

$C_v(R - W) + W(C_{miss} + \frac{C\_R}{2}) < C_v\frac{R(R+1)}{2}$, we expect the commit counter to offer an advantage.

Mixed invalidation. The global commit counter gets us partway to mixed invalidation: readers will notice conflicting writes only if (a) the writer acquires the object before the reader opens it, or (b) some transaction (not necessarily the writer) commits after the writer acquires and before the reader attempts to commit.

For comparison purposes, we have also built a full implementation of mixed invalidation. This implementation permits a transaction T to read the old version of object O even if O has been acquired by transaction S, so long as S has not committed. To correctly permit this "read through" operation, we augmented RSTM with a two-stage commit, similar to that employed by OSTM [3]. A writer transaction S that is ready to commit first CAS-es its status from active to finishing. S then attempts to CAS the global commit counter to one more than the value S saw when it last validated. If the increment fails, S revalidates its read set and re-attempts the increment. If the increment succeeds, S attempts to CAS its status from finishing to committed.

If transaction T reads O, then when S increments the counter, we are certain that T will validate before accessing any new state; this preserves consistency. Furthermore, although T can validate O against the old version when acquirer S is active, once S changes its status to finishing and increments the counter, T will fail validation. To preserve non-blocking properties, any transaction can (with permission from the contention manager) abort S even if it is finishing. In particular, if T's validation fails and T restarts, it will have the opportunity to abort S if it tries to open O.

Single thread optimization. A transaction can easily count the number of times that it commits a writing transaction without ever needing incremental validation. If this occurs many times in succession, the thread can assume that it is running in isolation and skip all bookkeeping and contention management calls (it must still increment the counter at the end of each write transaction). Should the global counter change due to activity in another thread, such an opportunistic transaction will have to abort and retry.

Using this optimization, transactions with large read and write sets can skip the $O(n)$ time and space overhead of bookkeeping, resulting in significant speedup for single-threaded transactional code.

4 Experimental Evaluation of Conflict Detection and Validation Strategies

In this section we evaluate the effectiveness of six different conflict detection strategies. For comparison, we also plot results for coarse-grained locks and for the Lev and Moir conflict counter. We consider different lookup / insert / remove ratios for benchmarks that include a lookup operation, and show that as the read ratio increases, so does the relative benefit of the global commit counter. Thus while no single conflict detection strategy offers consistently superior performance, we believe that our approximation of mixed invalidation constitutes an important new point in the design space. We also show that due to the cost of atomic operations on the critical path of every read, the

Lev and Moir heuristic performs roughly at the level of visible readers in RSTM, rarely outperforming even the baseline RSTM system with invisible reads and eager acquire.

We performed all experiments on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III CPUs. All code was compiled with *GCC v3.4.4* using –O3 optimizations. For each benchmark and lookup / insert / remove mix, we averaged the throughput of three 10-second executions. For RSTM benchmarks, we used the *Polka* contention manager [23].

4.1 Strategies Considered

RSTM supports both visible and invisible readers, and both eager and lazy acquire. We examine every combination other than visible reading with lazy acquire, which offers poor performance for our benchmarks and has comparatively weak motivation: while visibility allows readers to avoid incremental validation even when (unrelated) writers have committed, the effort they expend making themselves visible to writers is largely ignored, since writers delay conflict detection until commit time.

Visible readers with eager acquire (**Vis-Eager**) provides early detection of all conflicts without incremental validation. Invisible readers with eager acquire (**Invis-Eager**) also results in eager detection of all conflicts. Since reads are invisible, however, an acquiring transaction cannot detect that an object is being read; consequently, the acquirer cannot perform contention management but instead acquires the object obliviously, thereby implicitly dooming any extant invisible readers. To ensure consistency, transactions must incrementally validate their read set on every API call.

Invisible reads with lazy acquire (**Invis-Lazy**) results in lazy detection of all conflicts. This permits a high degree of concurrency between readers and writers, but requires incremental validation.

We also evaluate three heuristic validation methods, all based on a global commit counter.

In **Invis-Eager + Heuristic**, a transaction T validates incrementally only if some writer transaction W has committed since the last time T validated. In addition to reducing the frequency of incremental validations, this permits some lazy detection of read-write conflicts. If T reads O and then W acquires O, T may still complete if no other writing transaction commits between when W acquires O and when T commits.

Invis-lazy + Heuristic detects all conflicts lazily (at commit time). However, the heuristic permits a reduction in the overhead of validation: rather than incrementally validating on every API call, a transaction can validate trivially when no writer transaction W has committed since the last time T validated.

In **Mixed Invalidation**, read-write conflicts are detected lazily while write-write conflicts are detected eagerly. In contrast to Invis-Eager + Heuristic, Mixed Invalidation has precise conflict detection. For example, if T reads O, then S acquires O, then W acquires some other object P and commits, T will not fail its validation; it will detect that S has not committed, and that its version of O is valid.

4.2 Benchmarks

We tested our conflict detection strategies against six microbenchmarks: a web cache simulation using least-frequently-used page replacement (LFUCache [22]), an adjacency list-based undirected graph (RandomGraph), and four variants of an integer set.

The LFUCache benchmark uses a large array-based index and a small priority queue to track frequently accessed pages in a simulated web cache. When the queue is reheapified, we introduce hysteresis by swapping value-one nodes with value-one children. This helps more pages to accumulate hits. A Zipf distribution determines the likelihood that a page is accessed, with probability of an access to page i given as $p_c(i) \propto \sum_{0 \leq j \leq i} j^{-2}$.

In the RandomGraph benchmark, there is an even mix of inserts and deletes. When a node is inserted, it is given four randomly chosen neighbors. As nodes insert and leave the graph, the vertex set changes, as does the degree of each node. The graph is implemented as a sorted list of nodes, with each node owning a sorted list of neighbors. Every transaction entails traversal of multiple lists; transactions tend to be quite complex. Transactions also tend to overlap significantly; it is rare to have an empty intersection of one transaction's read set with another transaction's write set.

In the integer set benchmarks, we consider an *equal* ratio, consisting of one-third each of lookup, insert, and remove operations, and a *read-heavy* mix with 80% lookups and 10% each inserts and removes.

The integer set benchmarks are a red-black tree, a hash table, and two sorted linked lists. Transactions in the hash table insert or remove one of 256 keys from a 256 bucket hash table with overflow chains. This implementation affords high concurrency with very rare conflicts. The red-black tree is a balanced binary tree of values in the range $0..65535$. The linked lists hold values from $0..255$; one list uses *early release* [11] to avoid false conflicts; the other does not.

4.3 Discussion of Results

In LFUCache (Figure 2), transactions usually do only a small amount of work, accessing one or two objects. Furthermore, the work done by all transactions tends to be on the same object or small set of objects. As a result, there is no significant parallelism in the benchmark. Lazy acquire performs best in this setting, because it shrinks the window of contention between two transactions, decreasing the chance that a transaction that successfully acquires an object will be aborted. Furthermore, since the read and write sets are small, the global commit counter saves little validation effort. The only benefit of our heuristic is slightly better performance in the single-threaded case.

RandomGraph (Figure 3), by contrast, benefits greatly from a global commit counter. Its transactions' read sets typically contain hundreds of objects. Avoiding incremental validation consequently enables orders of magnitude improvement. We observe real scalability with all three heuristic policies. This scalability is directly related to relaxing the detection of read-write conflicts: reading and acquiring are heavily interleaved in the benchmark, and detecting read-write conflicts early leads to near-livelock, as shown by the Invis/Eager line. Mixed invalidation, moreover, outperforms the best lazy conflict detection strategy. This is a direct consequence of avoiding concurrent execution of two transactions that want to modify the same object, a scenario we have previously identified as dangerous. In the interest of full disclosure, we note that the lack of true concurrency still gives coarse-grain locks a dramatic performance advantage, ranging

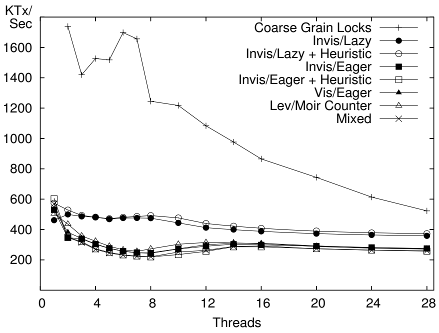


Fig. 2. LFUCache. Single-thread performance with coarse-grain locks is 4491 KTx/sec.

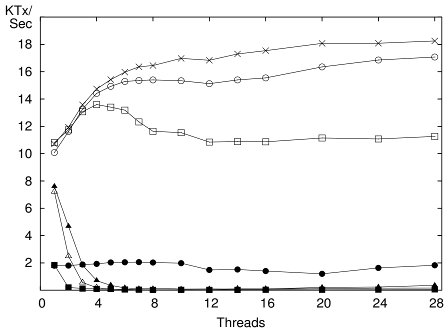


Fig. 3. RandomGraph. For readability, coarse-grain locks are omitted; the curve descends smoothly from 250 KTx/sec at 1 thread to 52 KTx/sec at 28.

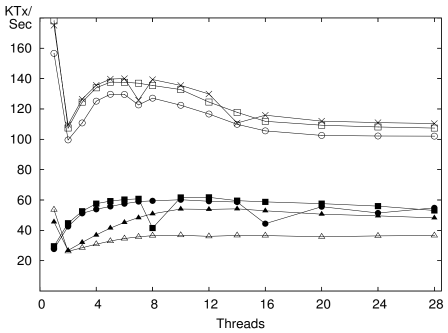


Fig. 4. Sorted List – 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend smoothly from 1540 KTx/sec at 1 thread to 176 KTx/sec at 28.

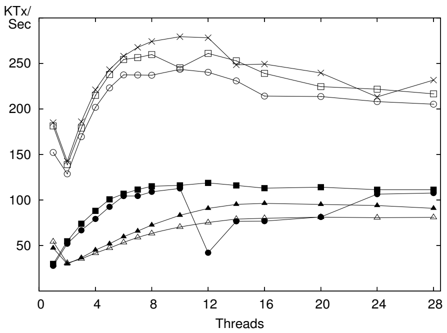


Fig. 5. Sorted List – 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend smoothly from 1900 KTx/sec at 1 thread to 328 KTx/sec at 28.

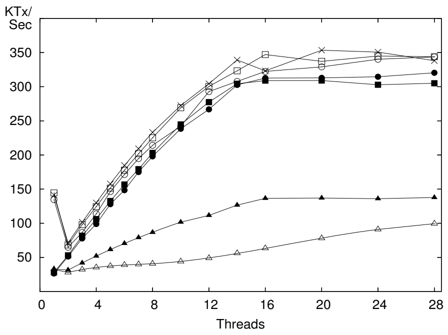


Fig. 6. Sorted List with Early Release – 33% lookup, 33% insert, 33% remove. Coarse-grain locks descend smoothly from 1492 KTx/sec at 1 thread to 171 KTx/sec at 28.

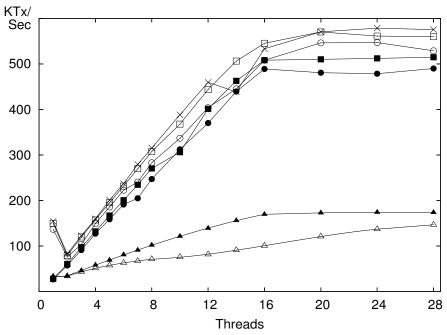


Fig. 7. Sorted List with Early Release – 80% lookup, 10% insert, 10% remove. Coarse-grain locks descend smoothly from 1994 KTx/sec at 1 thread to 354 KTx/sec at 28.

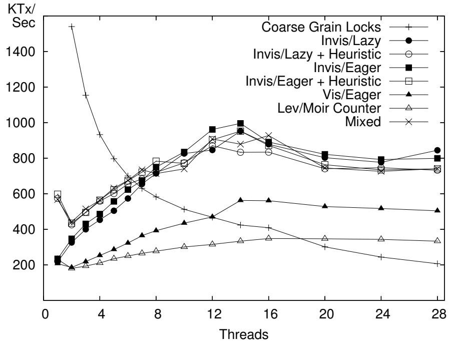


Fig. 8. Red-Black Tree – 33% lookup, 33% insert, 33% remove. Single-threaded performance with coarse-grain locks is 2508 KTx/sec.

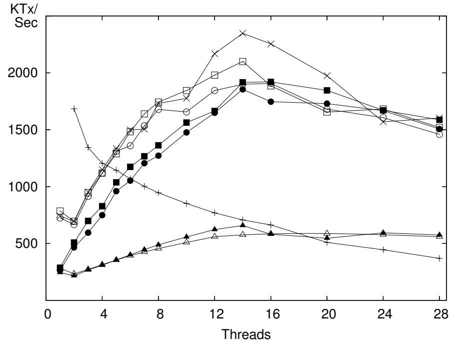


Fig. 9. Red-Black Tree – 80% lookup, 10% insert, 10% remove. Single-threaded performance with coarse-grain locks is 3052 KTx/sec.

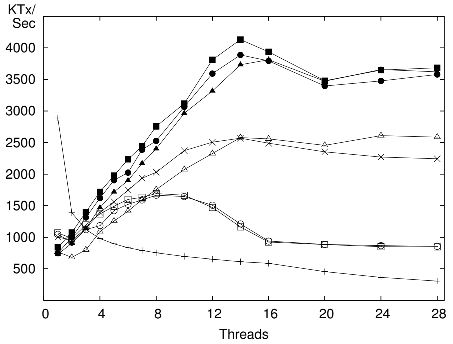


Fig. 10. Hash Table – 33% lookup, 33% insert, 33% remove

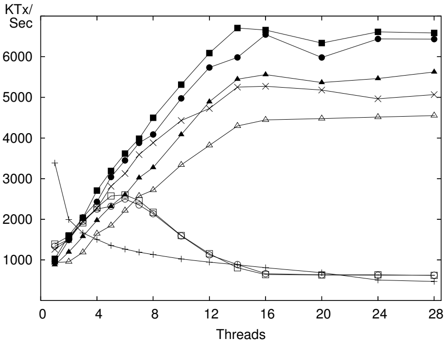


Fig. 11. Hash Table – 80% lookup, 10% insert, 10% remove

from more than two orders of magnitude at low thread counts to a factor of almost 3 with 28 active threads.

The LinkedList benchmarks (Figures 4–7) show a tremendous benefit from the global commit counter when early release is not used, and a small constant improvement with early release. The difference stems from the fact that without early release this benchmark is largely serial: the average reader opens 64 nodes to reach the middle of the list; any concurrent transaction that modifies an early node will force the reader to abort. With early release the programmer effectively certifies that modifications to early nodes are irrelevant once the reader has moved past them. No transaction keeps more than 3 nodes open at any given time, greatly increasing potential concurrency. Since transactions that modify the list do so with an acquire at the end of their transaction, there is little benefit to a relaxation of read-write conflict detection. The commit counter effectively reduces the frequency of incremental validation, however, and also significantly improves the single-threaded case.

In the RBTree benchmark (Figures 8–9), transactions tend to be small (fewer than 16 objects in the read set), with limited conflict. As a result, decreasing the cost of validation does not significantly improve performance, nor does relaxing read-write conflict detection. However, the heuristic significantly improves the single-threaded case. The value of the heuristic also increases noticeably with the fraction of read-only transactions, as the cost of validation becomes a larger portion of overall execution time.

Unlike the other benchmarks, HashTable (Figures 10–11) is hurt by the global commit counter. Since the table is only 50% loaded on average, the likelihood of two transactions conflicting is negligible. Furthermore, non-conflicting transactions do not read any common data objects. As a result, the benchmark is "embarrassingly concurrent." The introduction of a global counter serializes all acquiring transactions at a single memory location, and thus decreases opportunities for parallelism. Some of this cost is regained with mixed invalidation, especially when there is a high percentage of read-only transactions.

5 Related Work

In previous work, we reviewed several STM systems [16, 18] and ultimately designed both ASTM [17] and RSTM [19] to decrease overhead on the critical path of transactions. In ASTM, we adaptively switch from DSTM-style eager acquire [11] to OSTM-style lazy acquire [2, 3]. This permits some dynamic determination of how and when transactions should validate, but it is not as nuanced as mixed invalidation and does not avoid unnecessary validation.

In RSTM, we add the ability to switch between visible and invisible readers on a per-object basis, though we have not yet implemented automatic adaptation. RSTM thus subsumes the flexibility of Herlihy's SXM [5], which uses a *factory* to set visibility for entire classes of objects. While visible readers offer potential gains in fairness by allowing contention management for writes following uncommitted reads, we have found the cost in terms of reduced cache line sharing and reduced scalability to be unacceptably high; visible readers generally scale far worse than invisible readers when more than 4 threads are active.

Intel's McRT-STM [21] uses locks to avoid the need for object cloning, thereby improving performance. The McRT compiler inserts periodic validation checks in transactions with internal loops, to avoid the performance risk of long-running doomed transactions. As in OSTM, the programmer must insert any validation checks that are needed for correctness.

Recent proposals from Microsoft Research [9, 10] focus on word-based STM using Haskell and C#. The C# STM uses aggressive compiler optimization to reduce overheads, while the Haskell TM focuses on rich semantics for composability. Like previous word-based STMs [2, 8, 26], these systems avoid the cost of copying unmodified portions of objects, but incur bookkeeping costs on every load and store (or at least on every one that the compiler cannot prove is redundant). These differences complicate direct comparisons between word-based and object-based STM systems. Nonetheless, we believe that our heuristic mixed invalidation would be a useful addition to word-based STM, and might assist developers in further reducing the overheads of those systems.

Several proposals [1, 7, 12, 20] seek to leverage cache coherence protocols to achieve lightweight hardware transactions. However, these hardware TMs generally fix the conflict detection policy at design time, with eager read-write conflict detection more common than lazy [20]. We have recently proposed hardware assists to improve STM performance [27]. We believe this approach is more pragmatic: software dictates conflict detection and resolution policies, but special hardware instructions and cache states permit the small transactions in the common case to run as fast as coarse-grained locks.

The only other heuristic validation proposal we are aware of is the Lev and Moir conflict counter described in Section 3 [15]. While this heuristic removes unnecessary validation, it does not delay the detection of read-write conflicts. Inserting atomic operations into the critical path of every read shares lower-bound complexity with our visible reader implementation; we have shown that this strategy suffers the same costs (less cache line sharing, more processor stalls) as our visible reader implementation, and thus does not scale as well as invisible readers.

6 Conclusions

We have presented a comprehensive and detailed analysis of conflict detection strategies in RSTM. We assess existing policies for managing read-write and write-write conflicts using reader visibility and acquire time, and discuss the utility of *mixed invalidation* in avoiding conservative aborts of transactions that may be able to succeed.

We approximate mixed invalidation in RSTM using a global commit counter heuristic. Our implementation demonstrates that the resulting gain in concurrency can lead to significant performance improvements in workloads with long, highly contended transactions. We also demonstrate that a global commit counter can be used to detect the case when a system contains only one transactional thread, which can then opportunistically avoid the overhead of bookkeeping and contention management.

Our heuristics are still insufficient to close the performance gap between STM and locks in all cases. In fact, the global commit counter serves to *decrease* performance in highly concurrent workloads (such as hash tables) by forcing all transactions to serialize on a single memory location when they otherwise would access disjoint memory sets. Nonetheless, mixed invalidation appears to be a valuable step toward maximizing STM performance.

The fact that no one conflict detection or validation mechanism performs best across all workloads—and that the differences between mechanisms are both large and bidirectional—suggests that a production quality STM system should adapt its policy to match the offered workload. Our ASTM system [17] adapted in some cases between eager and lazy acquire; further forms of adaptation are the subject of future work.

Acknowledgments

The ideas in this paper benefited greatly from discussions with other members of the Rochester synchronization group: Sandhya Dwarkadas, Hemayet Hossain, Arrvindh Shriraman, Vinod Sivasankaran, Athul Acharya, David Eisenstat, and Chris Heriot. Our thanks as well to the anonymous referees.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] K. Fraser. *Practical Lock-Freedom.* Ph.D. Dissertation, UCAM-CL-TR-579. Cambridge Univ. Computer Laboratory, Feb. 2004.

[3] K. Fraser and T. Harris. Concurrent Programming without Locks. Submitted for publication, 2004.

[4] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proc. of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005. Held in conjunction with OOPSLA '05.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[6] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, page 102. IEEE Computer Society, June 2004.

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. of the 18th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct. 2003.

[9] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, pages 48–60, June 2005.

[10] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2006.

[11] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd Annual ACM Symp. on Principles of Distributed Computing*, July 2003.

[12] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*, pages 289–300. ACM Press, May 1993.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[14] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proc. of the 2006 Intl. Symp. on Memory Management*, June 2006.

[15] Y. Lev and M. Moir. Fast Read Sharing Mechanism for Software Transactional Memory (POSTER). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, St. Johns, NL, Canada, July 2004.

[16] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proc. of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Oct. 2004.

[17] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[18] V. J. Marathe and M. L. Scott. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report TR 839, Dept. of Computer Science, Univ. of Rochester, June 2004.

[19] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. Earlier, extended version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2006.

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proc. of the 11th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, pages 187–197, Mar. 2006.

[22] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[23] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[24] W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (POSTER). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[25] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[26] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[27] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006. Earlier, extended version available as TR 887, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

Transactional Locking II

Dave Dice[1], Ori Shalev[2,1], and Nir Shavit[1]

[1] Sun Microsystems Laboratories, 1 Network Drive, Burlington MA 01803-0903
{dice, shanir}@sun.com
[2] Tel-Aviv University, Tel-Aviv 69978, Israel
orish@post.tau.ac.il

Abstract. The transactional memory programming paradigm is gaining momentum as the approach of choice for replacing locks in concurrent programming. This paper introduces the transactional locking II (TL2) algorithm, a software transactional memory (STM) algorithm based on a combination of commit-time locking and a novel global version-clock based validation technique. TL2 improves on state-of-the-art STMs in the following ways: (1) unlike all other STMs it fits seamlessly with any systems memory life-cycle, including those using malloc/free (2) unlike all other lock-based STMs it efficiently avoids periods of unsafe execution, that is, using its novel version-clock validation, user code is guaranteed to operate only on consistent memory states, and (3) in a sequence of high performance benchmarks, while providing these new properties, it delivered overall performance comparable to (and in many cases better than) that of all former STM algorithms, both lock-based and non-blocking. Perhaps more importantly, on various benchmarks, TL2 delivers performance that is competitive with the best hand-crafted fine-grained concurrent structures. Specifically, it is ten-fold faster than a single lock. We believe these characteristics make TL2 a viable candidate for deployment of transactional memory today, long before hardware transactional support is available.

1 Introduction

A goal of current multiprocessor software design is to introduce parallelism into software applications by allowing operations that do not conflict in accessing memory to proceed concurrently. The key tool in designing concurrent data structures has been the use of locks. Coarse-grained locking is easy to program, but unfortunately provides very poor performance because of limited parallelism. Fine-grained lock-based concurrent data structures perform exceptionally well, but designing them has long been recognized as a difficult task better left to experts. If concurrent programming is to become ubiquitous, researchers agree that alternative approaches that simplify code design and verification must be developed. This paper is interested in "mechanical" methods for transforming sequential code or coarse-grained lock-based code into concurrent code. By mechanical we mean that the transformation, whether done by hand, by a preprocessor, or by a compiler, does not require any program specific information (such as

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 194–208, 2006.

the programmer's understanding of the data flow relationships). Moreover, we wish to focus on techniques that can be deployed to deliver reasonable performance across a wide range of systems today, yet combine easily with specialized hardware support as it becomes available.

1.1 Transactional Programming

The *transactional memory* programming paradigm of Herlihy and Moss [1] is gaining momentum as the approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification – by making parts of the code appear to be sequential without the need to program fine-grained locks. Transactions will hopefully remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to conflict with each other. Non-conflicting Transactions will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlock. There are currently proposals for hardware implementations of transactional memory (HTM) [1,2,3,4], purely software based ones, i.e. software transactional memories (STM) [5,6,7,8,9,10,11,12,13], and hybrid schemes (HyTM) that combine hardware and software [14,10].[1]

The dominant trend among transactional memory designs seems to be that the transactions provided to the programmer, in either hardware or software, should be "large scale", that is, unbounded, and dynamic. *Unbounded* means that there is no limit on the number of locations accessed by the transaction. *Dynamic* (as opposed to *static*) means that the set of locations accessed by the transaction is not known in advance and is determined during its execution.

Providing large scale transactions in hardware tends to introduce large degrees of complexity into the design [1,2,3,4]. Providing them efficiently in software is a difficult task, and there seem to be numerous design parameters and approaches in the literature [5,6,7,8,9,10,11]. as well as requirements to combine well with hardware transactions once those become available [14,10].

1.2 Lock-Based Software Transactional Memory

STM design has come a long way since the first STM algorithm by Shavit and Touitou [12], which provided a non-blocking implementation of static transactions (see [5,6,7,8,15,9,10,11,12,13]). A recent paper by Ennals [5] suggested that on modern operating systems deadlock prevention is the only compelling reason for making transactions non-blocking, and that there is no reason to provide it for transactions at the user level. We second this claim, noting that mechanisms already exist whereby threads might yield their quanta to other threads and that Solaris' *schedctl* allows threads to transiently defer preemption while holding locks. Ennals [5] proposed an all-software lock-based implementation

[1] A broad survey of prior art can be found in [6,15,16].

of software transactional memory using the object-based approach of [17]. His idea was to run through the transaction possibly operating on an inconsistent memory state, acquiring write locks as locations to be written are encountered, writing the new values in place and having pointers to an undo set that is not shared with other threads. The use of locks eliminates the need for indirection and shared transaction records as in the non-blocking STMs, it still requires however a closed memory system. Deadlocks and livelocks are dealt with using timeouts and the ability of transactions to request other transactions to abort.

Another recent paper by Saha et al. [11], uses a version of Ennals' lock-based algorithm within a run-time system. The scheme described by Saha et al. acquires locks as they are encountered, but also keeps shared undo sets to allow transactions to actively abort others.

A workshop presentation by two of the authors [18] shows that lock-based STMs tend to outperform non-blocking ones due to simpler algorithms that result in lower overheads. However, two limitations remain, limitations that must be overcome if STMs are to be commercially deployed:

Closed Memory Systems. Memory used transactionally must be recyclable to be used non-transactionally and vice versa. This is relatively easy in garbage collected languages, but must also be supported in languages like C with standard malloc() and free() operations. Unfortunately, all non-blocking STM designs require closed memory systems, and the lock-based STMs [5,11] either use closed systems or require specialized malloc() and free() operations.

Specialized Managed Runtime Environments. Current efficient STMs [5,11] require special environments capable of containing irregular effects in order to avoid unsafe behavior resulting from their operating on inconsistent states.

The TL2 algorithm presented in this paper is the first STM that overcomes both of these limitations: it works with an open memory system, essentially with any type of malloc() and free(), and it runs user code only on consistent states, eliminating the need for specialized managed runtime environments[2].

1.3 Vulnerabilities of STMs

Let us explain the above vulnerabilities in more detail. Current efficient STM implementations [18,17,5,11] require closed memory systems as well as managed runtime environments capable of containing irregular effects. These closed systems and managed environments are necessary for efficient execution. Within these environments, they allow the execution of "zombies": transactions that have observed an inconsistent read-set but have yet to abort. The reliance on an accumulated read-set that is not a valid snapshot [19] of the shared memory locations accessed can cause unexpected behavior such as infinite loops, illegal memory accesses, and other run-time misbehavior.

[2] The TL algorithm [18], a precursor of TL2, works with an open memory system but runs on inconsistent states.

The specialized runtime environment absorbs traps, converting them to transaction retries. Handling infinite loops in zombies is usually done by validating transactions while in progress. Validating the read-set on every transactional load would guarantee safety, but would also significantly impact performance. Another option is to perform periodic validations, for example, once every number of transactional loads or when looping in the user code [11]. Ennals [5] attempts to detect infinite loops by having every n-th transactional object "open" operation validate part of the accumulated read-set. Unfortunately, this policy admits infinite loops (as it is possible for a transaction to read less than n inconsistent memory locations and cause the thread to enter an infinite loop containing no subsequent transactional loads). In general, infinite loop detection mechanisms require extending the compiler or translator to insert validation checks into potential loops.

The second issue with existing STM implementations is their need for a closed memory allocation system. For type-safe garbage collected managed runtime environments such as that of the Java programming language, the collector assures that transactionally accessed memory will only be released once no references remain to the object. However, in C or C++, an object may be freed and depart the transactional space while concurrently executing threads continue to access it. The object's associated lock, if used properly, can offer a way around this problem, allowing memory to be recycled using standard malloc/free style operations. The recycled locations might still be read by a concurrent transaction, but will never be written by one.

1.4 Our New Results

This paper introduces the *transactional locking II* (TL2) algorithm. TL2 overcomes the drawbacks of all state-of-the-art lock-based algorithms, including our earlier TL algorithm [18]. The new idea in our new TL2 algorithm is to have, perhaps counter-intuitively, a global version-clock that is incremented once by each transaction that writes to memory, and is read by all transactions. We show how this shared clock can be constructed so that for all but the shortest transactions, the effects of contention are minimal. We note that the technique of time-stamping transactions is well known in the database community [20]. A global-clock based STM is also proposed by Riegel et al. [21]. Our global-clock based algorithm differs from the database work in that it is tailored to be highly efficient as required by small STM transactions as opposed to large database ones. It differs from the "snapshot isolation" algorithm of Riegel et al. as TL2 is lock-based and very simple, while Riegel et al. is non-blocking but costly as it uses time-stamps to choose between multiple concurrent copies of a transaction based on their associated execution intervals.

In TL2, all memory locations are augmented with a lock that contains a version number. Transactions start by reading the global version-clock and validating every location read against this clock. As we prove, this allows us to guarantee at a very low cost that only consistent memory views are ever read. Writing transactions need to collect a read-set but read-only ones do not. Once read- and write-sets are collected, transactions acquire locks on locations to be written, increment the global version-clock and attempt to commit by validating the read-set. Once committed, transactions update the memory locations with the new global version-clock value and release the associated locks.

We believe TL2 is revolutionary in that it overcomes most of the safety and performance issues that have plagued high-performance lock-based STM implementations:

- Unlike all former lock-based STMs it efficiently avoids vulnerabilities related to reading inconsistent memory states, not to mention the fact that former lock-based STMs must use compiler assist or manual programmer intervention to perform validity tests in user code to try and avoid as many of these zombie behaviors as possible. The need to overcome these safety vulnerabilities will be a major factor when going from experimental algorithms to actual production quality STMs. Moreover, as Saha et al. [11] explain, validation introduced to limit the effects of these safety issues can have a significant impact on overall STM performance.
- Unlike any former STM, TL2 allows transactional memory to be recycled into non-transactional memory and back using malloc and free style operations. This is done seamlessly and with no added complexity.
- As we show in Section 3, rather encouragingly, concurrent red-black trees derived in a mechanical fashion from sequential code using the TL2 algorithm and providing the above software engineering benefits, tend to perform as well as prior algorithms, exhibiting performance comparable to that of hand-crafted fine-grained lock-based algorithms. Overall TL2 is an order of magnitude faster than sequential code made concurrent using a single lock.

In summary, TL2's superior performance together with the fact that it combines seamlessly with hardware transactions and with any system's memory life-cycle, make it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

2 Transactional Locking II

The TL2 algorithm we describe here is a global version-clock based variant of the transactional locking algorithm of Dice and Shavit (TL) [18]. As we will explain, based on this global versioning approach, and in contrast with prior local versioning approaches, we are able to eliminate several key safety issues afflicting other lock-based STM systems and simplify the process of mechanical code transformation. In addition, the use of global versioning will hopefully improve the performance of read-only transactions.

Our TL2 algorithm is a two-phase locking scheme that employs *commit-time* lock acquisition mode like the TL algorithm, differing from *encounter-time* algorithms such as those by Ennals [5] and Saha et al. [11].

For each implemented transactional system (i.e. per application or data structure) we have a shared global version-clock variable. We describe it below using an implementation in which the counter is incremented using an increment-and-fetch implemented with a compare-and-swap (CAS) operation. Alternative implementation exist however that offer improved performance. The global version-clock will be read and incremented by each *writing transaction* and will be read by every read-only transaction.

We associate a special versioned write-lock with every transacted memory location. In its simplest form, the *versioned write-lock* is a single word spinlock that uses a CAS operation to acquire the lock and a store to release it. Since one only needs a single bit to indicate that the lock is taken, we use the rest of the lock word to hold a version number. This number is advanced by every successful lock-release. Unlike the TL algorithm or Ennals [5] and Saha et al. [11], in TL2 the new value written into each versioned write-lock location will be a property which will provide us with several performance and correctness benefits.

To implement a given data structure we allocate a collection of *versioned write-locks*. We can use various schemes for associating locks with shared data: *per object* (PO), where a lock is assigned per shared object, or *per stripe* (PS), where we allocate a separate large array of locks and memory is striped (partitioned) using some hash function to map each transactable location to a stripe. Other mappings between transactional shared variables and locks are possible. The PO scheme requires either manual or compiler-assisted automatic insertion of lock fields whereas PS can be used with unmodified data structures. PO might be implemented, for instance, by leveraging the header words of objects in the Java programming language [22,23]. A single PS stripe-lock array may be shared and used for different TL2 data structures within a single address-space. For instance an application with two distinct TL2 red-black trees and three TL2 hash-tables could use a single PS array for all TL2 locks. As our default mapping we chose an array of 2^{20} entries of 32-bit lock words with the mapping function masking the variable address with "0x3FFFFC" and then adding in the base address of the lock array to derive the lock address.

In the following we describe the PS version of the TL2 algorithm although most of the details carry through verbatim for PO as well. We maintain thread local read- and write-sets as linked lists. Each read-set entries contains the address of the lock that "covers" the variable being read, and unlike former algorithms, does not need to contain the observed version number of the lock. The write-set entries contain the address of the variable, the value to be written to the variable, and the address of its associated lock. In many cases the lock and location address are related and so we need to keep only one of them in the read-set. The write-set is kept in chronological order to avoid write-after-write hazards.

2.1 The Basic TL2 Algorithm

We now describe how TL2 executes a sequential code fragment that was placed within a TL2 transaction. As we explain, TL2 does not require traps or the insertion of validation tests within user code, and in this mode does not require type-stable garbage collection, working seamlessly with the memory life-cycle of languages like C and C++.

Write Transactions. The following sequence of operations is performed by a writing transaction, one that performs writes to the shared memory. We will assume that a transaction is a writing transaction. If it is a *read-only transaction* this can be denoted by the programmer, determined at compile time or heuristically at runtime.

1. **Sample global version-clock:** Load the current value of the *global version clock* and store it in a thread local variable called the *read-version* number (rv). This value is later used for detection of recent changes to data fields by comparing it to the *version* fields of their versioned write-locks.
2. **Run through a speculative execution:** Execute the transaction code (load and store instructions are mechanically augmented and replaced so that speculative execution does not change the shared memory's state, hence the term "speculative".) Locally maintain a *read-set* of addresses loaded and a *write-set* address/value pairs stored. This logging functionality is implemented by augmenting loads with instructions that record the read address and replacing stores with code recording the address and value to-be-written. The transactional load first checks (using a Bloom filter [24]) to see if the load address already appears in the write-set. If so, the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids read-after-write hazards.
   A load instruction sampling the associated lock is inserted before each original load, which is then followed by *post-validation* code checking that the location's versioned write-lock is free and has not changed. Additionally, we make sure that the lock's version field is $\leq rv$ and the lock bit is clear. If it is greater than rv it suggests that the memory location has been modified after the current thread performed step 1, and the transaction is aborted.
3. **Lock the write-set:** Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.
4. **Increment global version-clock:** Upon successful completion of lock acquisition of all locks in the write-set perform an increment-and-fetch (using a CAS operation for example) of the global version-clock recording the returned value in a local *write-version number* variable wv.
5. **Validate the read-set:** validate for each location in the read-set that the version number associated with the versioned-write-lock is $\leq rv$. We also verify that these memory locations have not been locked by other threads. In case the validation fails, the transaction is aborted. By re-validating the read-set, we guarantee that its memory locations have not been modified while steps 3 and 4 were being executed. In the special case where $rv + 1 = wv$ it is not necessary to validate the read-set, as it is guaranteed that no concurrently executing transaction could have modified it.

6. **Commit and release the locks:** For each location in the write-set, store to the location the new value from the write-set and release the locations lock by setting the version value to the write-version *wv* and clearing the write-lock bit (this is done using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improve performance under high contention. The Bloom filter allows us to determine if a value is not in the write-set and need not be searched for by reading the single filter word. Though locks could have been acquired in ascending address order to avoid deadlock, we found that sorting the addresses in the write-set was not worth the effort.

Low-Cost Read-Only Transactions. One of the goals of the proposed methodology's design is an efficient execution of *read-only transactions*, as they dominate usage patterns in many applications. To execute a read-only transaction:

1. **Sample the global version-clock:** Load the current value of the *global version-clock* and store it in a local variable called *read-version* (*rv*).
2. **Run through a speculative execution:** Execute the transaction code. Each load instruction is *post-validated* by checking that the location's versioned write-lock is free and making sure that the lock's version field is $\leq$ *rv*. If it is greater than *rv* the transaction is aborted, otherwise commits.

As can be seen, the read-only implementation is highly efficient because it does not construct or validate a read-set. Detection of read-only behavior can be done at the level of of each specific transaction site (e.g., method or atomic block). This can be done at compile time or by simply running all methods first as read-only, and upon detecting the first transactional write, abort and set a flag to indicate that this method should henceforth be executed in write mode.

2.2 A Low Contention Global Version-Clock Implementation

There are various ways in which one could implement the global version-clock used in the algorithm. The key difficulty with the global clock implementation is that it may introduce increased contention and costly cache coherent sharing. One approach to reducing this overhead is based on splitting the global version-clock variable so it includes a version number and a thread id. Based on this split, a thread will not need to change the version number if it is different than the version number it used when it last wrote. In such a case all it will need to do is write its own version number in any given memory location. This can lead to an overall reduction by a factor of *n* in the number of version clock increments.

1. Each version number will include the thread id of the thread that last modified it.
2. Each thread, when performing the load/CAS to increment the global version-clock, checks after the load to see if the global version-clock differs from the thread's previous *wv* (note that if it fails on the CAS and retries the

load/CAS then it knows the number was changed). If it differs, then the thread does not perform the CAS, and writes the version number it loaded and its id into all locations it modifies. If the global version number has not changed, the thread must CAS a new global version number greater by one and its id into the global version and use this in each location.

3. To read, a thread loads the global version-clock, and any location with a version number $> rv$ or $= rv$ and having an id different than that of the transaction who last changed the global version will cause a transaction failure.

This has the potential to cut the number of CAS operations on the global version-clock by a linear factor. It does however introduce the possibility of "false positive" failures. In the simple global version-clock which is always incremented, a read of some location that saw, say, value $v + n$, would not fail on things less than $v + n$, but with the new scheme, it could be that threads 1..n-1 all perform non-modifying increments by changing only the id part of a version-clock, leaving the value unchanged at v, and the reader also reads v for the version-clock (instead of $v + n$ as he would have in the regular scheme). It can thus fail on account of each of the writes even though in the regular scheme it would have seen most of them with values $v ... v + n - 1$.

2.3 Mixed Transactional and Non-transactional Memory Management

The current implementation of TL2 views memory as being clearly divided into transactional and non-transactional (heap) space where mixed-mode transactional and non-transactional accesses are proscribed. As long as a memory location can be accessed by transactional load or store operations it must not be accessible to non-transactional load and store operations and vice versa. We do however wish to allow memory recycled from one space to be reusable in the other. For type-safe garbage collected managed runtime environments such as that of the Java programming language, any of the TL2 lock-mapping policies (PS or PO) provide this property as the GC assures that memory will only be released once no references remain to an object. However, in languages such as C or C++ that provide the programmer with explicit memory management operations such as malloc and free, we must take care never to free objects while they are accessible. The pitfalls of finding a solution for such languages are explained in detail in [18].

There is a simple solution for the *per-stripe* (PS) variation of TL2 (and in the the earlier TL [18] scheme) that works with any malloc/free or similar style pair of operations. In the transactional space, a thread executing a transaction can only reach an object by following a sequence of references that are included in the transaction's read-set. By validating the transaction before writing the locations we can make sure that the read set is consistent, guaranteeing that the object is accessible and has not been reclaimed. Transacted memory locations are modified after the transaction is validated and before their associated locks are released. This leaves a short period in which the objects in the transaction's write-set must not be freed. To prevent objects from being freed in that period, threads let objects *quiesce* before freeing them. By *quiescing* we mean letting any activity on the transactional locations complete by making sure that all locks on an object's associated memory locations are released by their owners before. Once an object is quiesced it can be freed. This scheme works because any transaction that may acquire the lock and reach the disconnected location will fail its read-set validation.

Unfortunately, we have not found an efficient scheme for using the PO mode of TL2 in C or C++ because locks reside inside the object header, and the act of acquiring a lock cannot guaranteed to take place while the object is alive. As can be seen in the performance section, on the benchmarks/machine we tested there is a penalty, though not an unbearable one, for using PS mode instead of PO.

In STMs that use encounter-time lock acquisition and undo-logs [5,11] it is significantly harder to protect objects from being modified after they are reclaimed, as memory locations are modified one at a time, replacing old values with the new values written by the transaction. Even with quiescing, to protect from illegal memory modifications, one would have to repeatedly validate the entire transaction before updating each location in the write-set. This *repeated* validation is inefficient in its simplest form and complex (if at all possible) if one attempts to use the compiler to eliminate unnecessary validations.

2.4 Mechanical Transformation of Sequential Code

As we discussed earlier, the algorithm we describe can be added to code in a mechanical fashion, that is, without understanding anything about how the code works or what the program itself does. In our benchmarks, we performed the transformation by hand. We do however believe that it may be feasible to automate this process and allow a compiler to perform the transformation given a few rather simple limitations on the code structure within a transaction.

We note that hand-crafted data structures can always have an advantage over TL2, as TL2 has no way of knowing that prior loads executed within a transaction might no longer have any bearing on results produced by transaction.

2.5 Software-Hardware Interoperability

Though we have described TL2 as a software based scheme, it can be made inter-operable with HTM systems. On a machine supporting dynamic hardware transactions, transactions need only verify for each location read or written that the associated versioned write-lock is free. There is no need for the hardware transaction to store an intermediate locked state into the lock word(s). For every write they also need to update the version number of the associated lock upon completion. This suffices to provide interoperability between hardware and software transactions. Any software read will detect concurrent modifications of locations by a hardware writes because the version number of the associated lock will have changed. Any hardware transaction will fail if a concurrent software transaction is holding the lock to write. Software transactions attempting to write will also fail in acquiring a lock on a location since lock acquisition is done using an atomic hardware synchronization operation (such as CAS or a single location transaction) which will fail if the version number of the location was modified by the hardware transaction.

3 Empirical Performance Evaluation

We present here a set of microbenchmarks that have become standard in the community [25], comparing a sequential red-black tree made concurrent using various algorithms representing state-of-the-art non-blocking [6] and lock-based [5,18] STMs. For lack of space we can only present the red-black tree data structure and only four performance graphs.

The sequential red-black tree made concurrent using our transactional locking algorithm was derived from the `java.util.TreeMap` implementation found in the Java programming language JDK 6.0. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al. [26]. We would have preferred to use the exact Fraser-Harris red-black tree [6] but that code was written to their specific transactional interface and could not readily be converted to a simple form.

The sequential red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. The put operation installs a key-value pair. If the key is not present in the data structure put will insert a new element describing the key-value pair. If the key is already present in the data structure put will simply update the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls put, get and delete to operate on the underlying data structure. The harness allows for the proportion of put, get and delete operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range describes the maximum possible size (capacity) of the data structure.

For our experiments we used a 16-processor Sun Fire^{TM} V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris^{TM} 10. As claimed in the introduction, modern operating systems handle locking well, even when the number of threads is larger than the number of CPUs. In our benchmarks, our of STMs used the Solaris *schedctl* mechanism to allow threads to request short-term preemption deferral by storing to a thread-specific location which is read by the kernel-level scheduler. Preemption deferral is advisory - the kernel will try to avoid preempting a thread that has requested deferral. We note that unfortunately we could not introduce the use of schedctl-based preemption deferral into the hand crafted lock-based *hanke* code, the lock-based *stm_ennals* code described below, or *stm_fraser*. This affected their relative performance beyond 16 threads but not in the range below 16 threads.

Our benchmarked algorithms included:

Mutex. We used the Solaris POSIX threads library mutex as a coarse-grained locking mechanism.

stm_fraser. This is the state-of-the-art non-blocking STM of Harris and Fraser [6]. We use the name originally given to the program by its authors. It has a special record per object with a pointer to a transaction record. The transformation of sequential to transactional code is not mechanical: the programmer specifies when objects are transactionally opened and closed to improve performance.

stm_ennals. This is the lock-based encounter-time object-based STM algorithm of Ennals taken from [5] and provided in LibLTX [6]. Note that LibLTX includes the original Fraser and Harris lockfree-lib package. It uses a lock per object and a non-mechanical object-based interface of [6]. Though we did not have access to code for the Saha et al. algorithm [11], we believe the Ennals algorithm to be a good representative this class of algorithms, with the possible benefit that the Ennals structures were written using the non-mechanical object-based interface of [6] and because unlike Saha et al., Ennals write-set is not shared among threads.

TL/PO. A version of our algorithm [18] which does not use a global version clock, instead it collects read and write-sets and validates the read-set after acquiring the locks on the memory locations. Unlike TL2, it thus requires a safe running environment. We bring here the per-object locking variation of the TL algorithm.

hanke. This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [27] as coded by Fraser [6]. The idea of relaxed balancing is to uncouple the re-balancing from the updating in order to speed up the update operations and to allow a high degree of concurrency.

TL2. Our new transactional locking algorithm. We use the notation TL2/PO and TL2/PS to denote the per-object and per-stripe variations. The PO variation consistently performed better than PS, but PS is compatible with open memory systems.

In Figure 1 we present four red-black tree benchmarks performed using two different key ranges and two set operation distributions. The key range of $[100, 200]$ generates a small size tree while the range $[10000, 20000]$ creates a larger tree, imposing larger transaction size for the set operations. The different operation distributions represent two type of workloads, one dominated by reads (5% puts, 5% deletes, and 90% gets) and the other (30% puts, 30% deletes, and 40% gets) dominated by writes.

In all four graphs, all algorithms scale quite well to 16 processors, with the exception of the mutual exclusion based one. Ennals's algorithm performs badly on the contended write-dominated benchmark, apparently suffering from frequent transaction collisions, which are more likely to occur in encounter-time

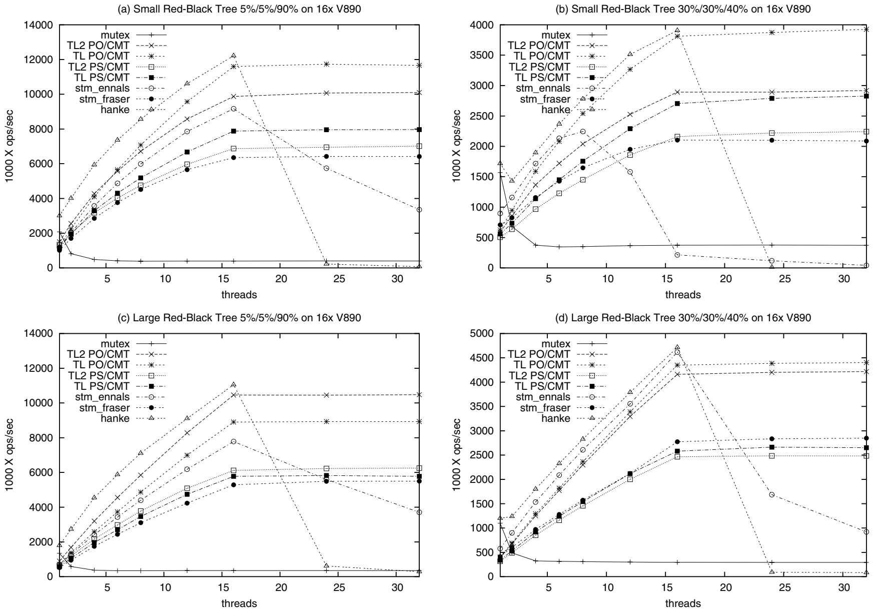


Fig. 1. Throughput of Red-Black Tree with 5% puts and 5% deletes and 30% puts, 30% deletes

locking based solutions. Beyond 16 threads, the Hanke and Ennals algorithms deteriorate because we could not introduce the *schedctl* mechanism to allow threads to request short-term preemption deferral. It is interesting to note that the Fraser-Harris STM continues to perform well beyond 16 threads even without this mechanism because it is non-blocking. As expected, object based algorithms (PO) do better than stripe-based (PS) ones because of the improved locality in accessing the locks and the data.

The performance of all the STM implementations usually differs by a constant factor, most of which we associate with the overheads of the algorithmic mechanisms employed (as seen in the single thread performance). The hand-crafted algorithm of Hanke provides the highest throughput in most cases because its single thread performance (a measure of overhead) is superior to all STM algorithms. On the smaller data structures TL/PO (or TL/PS) performs better than TL2/PO (respectively TL2/PS) because of lower overheads, part of which can be associated with invalidation traffic caused by updates of the version clock (this is not traffic caused by CAS operations on the shared location. It is due to the fact that the location is being updated). This is reversed and TL2 becomes superior when the data structure is large because the read-set is larger and read-only transactions incur less overhead in TL2. The TL and TL2 algorithms are in most cases superior to Ennals's STM and Fraser and Harris's STM. In all benchmarks they are an order of magnitude faster than the single lock Mutex implementation.

4 Conclusion

The TL2 algorithm presented in this paper provides a safe and easy to integrate STM implementation with reasonable performance, providing a programming environment similar to that attained using global locks, but with a ten-fold improvement in performance. TL2 will easily combine with hardware transactional mechanisms once these become available. It provides a strong indication that we should continue to devise lock-based STMs.

There is however still much work to be done to improve TL2's performance. A lot of these improvements may require hardware support, for example, in implementing the global version clock and in speeding up the collection of the read-set. The full TL2 code will be publicly available shortly.

References

1. Herlihy, M., Moss, E.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture. (1993)
2. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture, Washington, DC, USA, IEEE Computer Society (2005) 494–505
3. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Washington, DC, USA, IEEE Computer Society (2005) 316–327
4. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture, Washington, DC, USA, IEEE Computer Society (2004) 102
5. Ennals, R.: Software transactional memory should not be obstruction-free. www.cambridge.intel-research.net/ rennals/notlockfree.pdf (2005)
6. Harris, T., Fraser, K.: Concurrent programming without locks. www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-cpwl-submission.pdf (2004)
7. Herlihy, M.: The SXM software package. `http://www.cs.brown.edu/~mph/SXM/README.doc` (2005)
8. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.: Software transactional memory for dynamic data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing. (2003)
9. Marathe, V.J., Scherer III, W.N., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of the 19th International Symposium on Distributed Computing, Cracow, Poland (2005)
10. Moir, M.: HybridTM: Integrating hardware and software transactional memory. Technical Report Archivist 2004-0661, Sun Microsystems Research (2004)
11. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: A high performance software transactional memory system for a multi-core runtime. In: To appear in PPoPP 2006. (2006)
12. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2) (1997) 99–116

13. Welc, A., Jagannathan, S., Hosking, A.L.: Transactional monitors for concurrent objects. In: Proceedings of the European Conference on Object-Oriented Programming. Volume 3086 of Lecture Notes in Computer Science., Springer-Verlag (2004) 519–542
14. Ananian, C.S., Rinard, M.: Efficient software transactions for object-oriented languages. In: Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL), ACM (2005)
15. Marathe, V.J., Scherer, W.N., Scott, M.L.: Design tradeoffs in modern software transactional memory systems. In: LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems, New York, NY, USA, ACM Press (2004) 1–7
16. Rajwar, R., Hill, M.: Transactional memory online. `http://www.cs.wisc.edu/trans-memory` (2006)
17. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. **38**(11) (2003) 388–402
18. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT06 ACM Workshop. (2006)
19. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4) (1993) 873–890
20. Thomasian, A.: Concurrency control: methods, performance, and analysis. ACM Comput. Surv. **30**(1) (1998) 70–119
21. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: 20th International Symposium on Distributed Computing (DISC). (2006)
22. Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S., White, D.: An efficient meta-lock for implementing ubiquitous synchronization. ACM SIGPLAN Notices **34**(10) (1999) 207–222
23. Dice, D.: Implementing fast java monitors with relaxed-locks. In: Java Virtual Machine Research and Technology Symposium, USENIX (2001) 79–90
24. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970) 422–426
25. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing, ACM Press (2003) 92–101
26. Cormen, T.H., Leiserson, Charles, E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990) COR th 01:1 1.Ex.
27. Hanke, S.: The performance of concurrent red-black tree algorithms. Lecture Notes in Computer Science **1668** (1999) 286–300

Less Is More: Consensus Gaps Between Restricted and Unrestricted Objects

Yehuda Afek and Eran Shalom

School of Computer Science
Tel-Aviv University
afek@post.tau.ac.il, eransha@post.tau.ac.il

Abstract. What characteristics of an object determine its consensus number? Here we analyze how the consensus power of various objects changes without changing their functionality, but by placing certain restrictions on the object usage. For example it is shown that the consensus number of either a bounded-use **queue** or **stack** is 3 while the consensus number of the long-lived bounded-size and unbounded-size versions of either is 2. Similarly we show that the consensus number of restricted versions of **Fetch&Add**, **Swap** and **Set** are infinite (n) while for the unrestricted counterparts it is 2. This paper thus underlines the fact that the consensus number of an object reflects the amount of coordination required in the object implementation and not by its capacity. That is, the more corners, broken edges, and other hard limitations placed on an object, the higher its consensus number tends to be.

Keywords: Consensus hierarchy, Common2, Wait-free, Queues, Stacks, Bounded-use, Bounded-size, Long-lived, Fetch&Add, Swap, Set.

1 Introduction

As defined by Herlihy [2], an object has consensus number k if any number of copies of this object and of read/write registers can be used to implement a wait-free k-consensus protocol, but cannot be used to implement a wait-free $k + 1$-consensus protocol (in a shared memory multi processor system). Thus objects with higher consensus number are not deterministically implementable from objects with lower consensus numbers.

Most of the objects analyzed in [2] are unbounded in their size and are unrestricted in the number of operations that may be applied on them. While in [10,11] it has been shown that the more values a single *compare-and-swap* object can hold, the higher its consensus number, herein we analyze what was believed to be the case, that for most objects with consensus number 2, their consensus number increases when various restrictions are placed on the object usage, i.e., on the number of operations of a certain type that may be successfully applied on the object, or on its size.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 209–223, 2006.

We use the term *f-bounded-use* to describe an object that limits the number of times (f times) a certain operation from its set of operations may be successfully applied (further invocations of that operation respond with a **fail** indication, or with no indication). We use the term *bounded-size* to describe an object whose space (memory) is finite. Mostly we consider restricting operations that change the object state.

While it is known that placing restrictions on shared memory objects is likely to increase their consensus number, here we systematically and thoroughly analyze the effect of restricting the number of operations that can be applied to an object. Specifically we show that when restricting the total number of successful enqueue operations on a queue, its consensus number increases from 2 to 3. Similarly, bounding the total number of successful push operations on a *stack*, its consensus number increases from 2 to 3. These results are surprising when compared to the following; When limiting the number of successful Fetch&Add (Swap) operations on a F&A (*Swap*) object that supports read operations, its consensus number increases to n.

In addition we prove that *bounded-sizeSet*'s consensus number is n, whereas in [2] Herlihy shows that unbounded size *set*'s consensus number is 2; Most of the proofs in this paper use the standard valency arguments as in [2,7].

In the next section we present the definitions and model used inhere. Related work is presented in Sect. 3. In Sect. 4 we prove that *f-bounded-useQ*'s consensus number is exactly 3 then we claim (proof eliminated) that *bounded-sizeQ*'s consensus number is 2. The proofs of the same results for *f-bounded-useStack* and *bounded-sizeStack* are similar to those of queue and are eliminated from this paper. In Sect. 5 we show that both *f-bounded-useF&A* and *f-bounded-useSwap* have consensus number n. In Sect. 6 we discuss the consensus gap that might exist between bounded-size and unbounded objects; we use *bounded-sizeSet* as an example. In Sect.7 we show an implementation of an *f-bounded-useQ* using *1-bounded-useSwap* registers. A concluding discussion is given in Sect. 8.

2 Definitions and Model

Our model follows [2]. The system consists of a number of asynchronous sequential threads of execution called *processes*, which communicate only by using shared objects. Each object is manipulated by a set of *operation types*. All objects are wait-free and linearizable (as defined in [5,6]). Our proof techniques are mostly standard FLP arguments as in [2,7].

2.1 Bounded-Use Objects

Definition 1. *An* ***f-bounded-use object*** *is an object that one of its operation types, Op, is* ***restricted****, that is only the first f executions of Op are completed successfully. Once the f^{th} execution of Op completed, the object becomes* ***exhausted****, and any further application of Op on the object fails (by either returning a fail signal, or not, both variations are studied). Operations other than Op execute correctly regardless to whether the object is exhausted or not.*

Definition 2. *An **f-bounded-useQ,** is a queue in which only the first f enqueue operations execute successfully. If the f-bounded-useQ is not empty, a dequeue operation returns the oldest element residing in the queue and removes it from the queue; otherwise it returns empty.*

Definition 3. *An **f-bounded-useStack,** is a stack in which only the first f push operations execute successfully. If the f-bounded-useStack is not empty, a pop operation returns the youngest element residing in the stack and removes it from the stack; otherwise it returns empty.*

Definition 4. *An **f-bounded-useF&A,** is a fetch&add object in which only the first f fetch&add operations successfully execute. In any state, a read operation returns the f-bounded-useF&A's value.*

Definition 5. *An **f-bounded-useSwap,** is a swap object in which only the first f swap operations successfully execute. In any state, a read operation returns the f-bounded-useSwap's value.*

An object with a bounded space (memory) is a ***bounded-size object***. Note that in many bounded-size objects, after enough operations that add data to such an object, the operations may return failure. Note that some bounded-use objects are also bounded-size. For example an *f-bounded-useQ* is bounded to f elements.

3 Related Work

In [10,11] it is shown that when restricting the range of values a *compare&swap* register may have, its consensus number is decreased; moreover it is conjectured that their result might be extended to any *read-modify-write* object. The different values a register may hold is related to its size, thus the register size might change the consensus number of the register. The result presented in [10,11] impacts strong synchronization objects such as *read-modify-write* objects that have infinite consensus number (when their size is unbounded), but this result does not apply to lower consensus number objects. Restricting the size of objects such as *Swap*, *F&A* and *Test&Set* , does not impact their consensus number, which is 2. In contrast the results presented in this paper apply mostly to such objects, whose consensus number is 2.

In [12] Plotkin presented the *Sticky-bit*, which is essentially a read/write atomic bit in which only one write is allowed. In Plotkin's definition the Sticky-bit returns true the first time its value is updated (using the JAM operation), or when if the JAM operation does not change its value, otherwise the operation fails. The Sticky-bit supports a read operation. The consensus number of the sticky-bit is 2, which is greater than an ordinary read/write bit's consensus number. The consensus number of any sticky n-value read/write register is n.

The question whether the queue object is in common2 class of objects as defined in [1], was addressed by some researches over the years but remains open. Li [3] had shown an implementation for a wait-free linearizable queue where the number of dequeuers was limited to be 2. The implementation was based on an helping mechanism. David [4] gave an implementation using a single enqueuer with unbounded number of dequeuers. Both Li and David as well as [9] conjecture that wait-free linearizable queue is not in common2.

Jayanti and Toueg in [8] presented the $f-$*bounded peek queue* that supports up to f enqueue operations and no dequeue operation. The $f+1^{th}$ enqueue operation in [8] fails and the queue becomes faulty, which means any operation of any kind fails. This specification although vastly different than the *f-bounded-useQ* has some resemblance to it. In [8] it is shown that $f-$*bounded* queue can solve consensus between at most f processes. While the consensus number of a long-lived version of a queue that supports a peek operation is n.

4 Bounded-Use Queue's and Stack's Consensus Numbers

Theorem 6. *f-bounded-useQ's consensus number is at least 3.*

Proof. Let p_1, p_2, p_3 be three processes. The protocol given in algorithm 1. solves three process consensus.

Algorithm 1. Decide method of process i (3-process consensus using *f-bounded-useQ*)

```
shared objects
  Q : f-bounded-useQ. Initialized[1] with f-1$\perp$ elements
  R : 3 swmr registers array. R[i] keeps the input of process i
  loser$[1-3]$ : swmr registers initialized to false
local variables
  winner : register initialized to i

decide$_i$(v : inputvalue)
d1:  R[i] = v;
d2:  if (Q.enqueue(i) == fail)
d3:     loser$[i]$=true;
d4:     do
d5:       winner = Q.Dequeue();
d6:     until ((winner == empty) OR (winner$\neq\perp$));
d7:     if (winner == empty)
d8:       winner = (process whose loser$[\cdot]$ == false);
d7e:    fi;
d2e: fi;
d9:  return R[winner];
end;
```

[1] initialization is done using f-1 enqueue operations

Wait Freedom. The Q object is wait-free, thus all operations applied on Q are wait-free. There is a single loop in the algorithm in which the only operation applied on Q is dequeue. There is at most one **successful** enqueue applied on Q, thus it is guaranteed that the loop in d_4 ends within a finite number of iterations.

Validity. Let p_1, p_2, p_3 be three processes executing the protocol. Clearly exactly one process successfully enqueues and after this enqueue the queue is exhausted. Without loss of generality let p_1 succeeds in enqueueing, thus p_1 is the winner. A process that fails to enqueue declares itself *loser* and finds out who the winner is by repeatedly dequeuing from the queue until either getting *empty* or dequeuing a non $\perp$ value. Clearly only one of the two *losers* dequeues the winner id in line $d5$. Moreover, at the time that the winner id has been dequeued the dequeuing process has already declared itself *loser.* Thus when a process fails to dequeue a non $\perp$ value, it knows who the winner is by watching which other process has declared itself *loser.* □

Corollary 7. *f-bounded-useQ is not in common2.*

While algorithm 1. uses an initialized *f-bounded-useQ*, a generalization of this algorithm results in an algorithm in which no initialization of the *f-bounded-useQ* is required. The following algorithm was given by Israel Nir from Tel-Aviv Univ.: Every process enqueues its id until it fails. Then it records the number of successful enqueues it made to a register that is initialized to -1. Next the process dequeues until it dequeues empty. The lowest process id that was ever enqueued is the elected winner. To determine the winner the following procedure is used; either a process dequeues all f elements (f is known), and can easily deduce the winner, or at least one other process has successfully dequeued. Such other process announced the number of times it successfully enqueued before its first dequeue. Thus a process that dequeued empty knows the number of elements it enqueued (if any), and the number of elements another process enqueued, from which it deduces the number of elements the third process has enqueued; and thus it can choose the winner.

Theorem 8. *f-bounded-useQ's consensus number is exactly 3.*

Proof. In Sect. A.1 we prove that *f-bounded-useQ* can not solve 4-process consensus. Along with theorem 6 the proof follows. □

Theorem 9. *bounded-sizeQ's consensus number is at least 2.*

Proof. Follows from [2]. □

Theorem 10. *bounded-sizeQ's consensus number is exactly 2.*

Proof follows standard arguments as in [2], thus omitted.

A similar 3-process consensus protocol exists for *f-bounded-useStack*. With minor changes the above theorems are proved for *f-bounded-useStack*'s and *bounded-sizeStack*. We summarize here the theorems.

Theorem 11. *f-bounded-useStack's consensus number is exactly 3.*

Theorem 12. *bounded-sizeStack's consensus number is exactly 2.*

5 *f*-Bounded-Use F&A's & Swap's Consensus Number

5.1 *f-bounded-useF&A*

Theorem 13. *f-bounded-useF&A's consensus number is n.*

Binary consensus among n processes is easy using one or two instances of *f-bounded-useF&A*. For example if the *f-bounded-useF&A* object supports failure notifications, we let each process fetch&add its own input (0 or 1) until it fails, then it reads the *f-bounded-useF&A*'s value; if the value is 0 it decides 0, otherwise it decides 1. This algorithm can be twisted to work also if f is unknown, and requires no special initialization, as follows; Use two fetch&add registers, so that every process fetch&add its input to the first register then fetch&add 1 to the second, the process repeats updating the two registers until the value of the second fetch&add register stops changing, at which point the the fetch&add register is exhausted, and hence also the first fetch&add register is exhausted. The winner is then decided as above.

We present here an n-process n-value consensus protocol that does not use the protocols we described above. The algorithm (Algorithm 2.) demonstrates the essence of the bounded-use property of the *f-bounded-useF&A* object.

Algorithm 2. Decide method of process i (n-process consensus using **uninitialized** *f-bounded-useF&A*)

```
shared objects
  F : f-bounded-useF&A. Initialized to 0
  R : n swmr registers array. R[i] keeps the input of process i
local variables
  winner : register

decide$_i$(v : inputvalue)
d1:  R[i] = v;
d2:  do $f$ times
d3:    F.Fetch&Add($(f+1)^i$);
d2e: od;
d4:  winner = process with minimal id
              in coefficients of F.Read();
d5:  return R[winner];
end;
```

Proof. (of Theorem 13) Let p_1, $p_2 ... p_n$ be n processes. Algorithm 2. solves n-process consensus.

Wait Freedom. Follows immediately from the fact that f is finite and the loop is executed at most f times.

Validity. In algorithm 2. there is no special initialization using fetch&add operations on the *f-bounded-useF&A*. All processes share an instance of an *f-bounded-useF&A*, named F. Every process applies f fetch&add $((f+1)^i)$ operations on F, followed by a read of F. The fact that we add different powers of $(f+1)$ (we assume $f > 0$ thus $f+1 > 1$) by different processes, assures us that there is exactly one combination of total of f F&A operations, that leads to each possible value of F. That is, after a total of f fetch&add operations, F is exhausted, and $F = \sum_{i=1}^{n} a_i(f+1)^i$, a_i denoting the number of times process i successfully increments F. Notice that $\sum_{i=1}^{n} a_i = f$, thus there is a **single** combination of a_i values that results in the value of F.[1] Since F is exhausted all processes read the same value of F once each of them performed f fetch&add operations. Every process calculates the coefficients (a_i) combination and decides the process with the lowest id ever succeeded incrementing F as the winner. Notice that the computation of $F's$ coefficients is finite since finite domain of process ids is assumed (if this was not the case we can always create pseudo-ids using simple F&A register that provides each process an id to be used in the protocol instead of its original id). □

***f-bounded-useF&A*, Unknown *f* and no Failure Response.** Algorithm 2. assumes f is known. In case f is unknown we can use a single instance of shared *f-bounded-useF&A* (call it F_c, initialized to 0). Before performing the loop in d_2, every process repeatedly performs fetch&add on F_c until it notices no change in F_c's value (F_c is exhausted). Then it reads F_c's value and use it in d_2 and d_3 (to represent f).

Claim. If the *f-bounded-useF&A* returns normally when exhausted (i.e. the register's value is returned instead of failure notification), its consensus number remains n.

Proof. Algorithm 2. ignores any result value in d_3. It is easy to see that when f is unknown and we use F_c, a process can detect when F_c stops changing by reading its value prior and after the fetch&add is applied. □

5.2 *f-bounded-useSwap*

Protocols and proofs similar to the above are used (not all repeated herein) to show that *f-bounded-useSwap*'s consensus number is n. There is no need to know f in advance in order to perform special initialization prior execution of the protocol, nor is there a need for the *f-bounded-useSwap* to return failure notification to a swap operation when exhausted. Algorithm 3. is an example to an n-process consensus protocol using **uninitialized** *f-bounded-useSwap* registers that return failure notifications in case swap is applied on when exhausted. Theorem 14 follows.

[1] For any i, j *such that* $j > i$, for p_i to add the same value to F as p_j, p_i must perform at at least f fetch&add operations, but then F becomes exhausted.

Theorem 14. *f-bounded-useSwap's consensus number is n.*

Algorithm 3. Decide method of process i (n-process consensus using **uninitialized** *f-bounded-useSwap*)

```
shared objects
  Swp : f-bounded-useSwap. Initialized to 0
  R   : n swmr registers array. R[i] keeps the input of process i
local variables
  winner : register

decide_i(v : inputvalue)
d1:  R[i] = v;
d2:  while(Swp.Swap(i)≠fail);
d3:  winner = Swp.Read();
d4:  return R[winner];
end;
```

6 *bounded-sizeSet*

In the previous sections we claimed that there might be a difference between a bounded-size and unbounded versions of the same object. The *bounded-sizeSet* is an example for such a case. The unbounded set's consensus number is 2 as claimed in [2]; we show here that *bounded-sizeSet*'s consensus number is n.

A set is a data structure to which one can execute either: *insert, delete, find, min, member*. Other operations that involve more than one set are: *union, merge, intersection, equal, assign*, etc. all can be found in the literature.

The basic set operations however, are: insert, delete and find. We use only this subset of operations to show the consensus number of a *bounded-sizeSet*. Notice that in a *bounded-sizeSet* an insert might fail due to the fact that the set is fully occupied.

Theorem 15. *bounded-sizeSet's consensus number is n.*

Proof. Let p_1, $p_2 ... p_n$ be n processes. The protocol given in algorithm 4. solves n process consensus. We prove correctness by showing wait-freedom and validity.

Wait Freedom. Follows immediately since the loop iterates over a finite number of processes, and all object are considered wait free.

Validity. A winner is a process that succeeds on the insert operation(d_2). First observe that the insertions are done by the id of the processes, thus no collisions of values are possible, and the reason for an insert operation to fail has only to do with the fact that the size of the set is bounded. Moreover notice that

Algorithm 4. Decide method of process i (n-process consensus using *bounded-sizeSet*)

```
shared objects
  S : bounded-sizeSet of size f. Initialized with $\bot_1 ... \bot_{f-1}$
  R : n swmr registers array. R[i] keeps the input of process i
local variables
  winner : register initialized to i

decide$_i$(v : inputvalue)
d1:  R[i] = v;
d2:  if (S.insert(i) == fail)
d3:     for each process p do
d4:       if (S.find(p)) then winner = p;
d3e:    rof;
d2e: fi;
d6:  return R[winner];
end;
```

S is already initialized[2] with $f-1$ different elements($\bot_1 ... \bot_{f-1}$), hence there is only one free place left in S. Thus there is exactly one process that might succeed inserting its id into S. Without loss of generality let this process be p_1. Since p_1 succeeds in its insert it returns its own value. Now by negation assume there is another winner, without loss of generality let it be p_2. If p_2 is a winner then p_2's insert succeeds. However notice that there is no use of delete in the protocol, thus either p_1 failed on its insert(leaving the single free place free) or the set object is incorrect. But neither of these is the case, since we assumed p_1's insert succeeded, and we assume that the set object is implemented correctly. Hence only p_1 is the winner. It is left to show that other processes return p_1's value. As it was shown every process other than p_1 fails to insert. Without loss of generality let p_2 be such a process. Notice that S contains exactly one process id (p_1's id), thus once p_2 executes the loop in $d_3 - d_{3e}$, all find(p) operations where $p \neq p_1$ fail but for $p = p_1$ the operation succeeds; thus eventually p_2 returns the value of p_1. Notice that p_1's id must already be in S, since otherwise the set object is incorrect (if $p_1's$ id is not in S, then another process must succeed on its insert). Hence there is exactly one winner. □

Notice that this result has to do with the fact the we only use a subset of the bounded-sizeSet operations (insert and find), that imposes a bounded-use set be-

[2] Notice that the initialization of the set S can be avoided, simply by causing the value inserted to the set to be unique among the processes. i.e. every process inserts at most f distinct values, that are constructed from its id and a local counter ($p_{id}1..p_{id}f$); thus all values inserted to the set by all processes are guaranteed to be distinct. A process fails to insert a value to the set, only when the set is full (and not due to duplicated values). Once failed to insert, a process goes over all initial values possibly inserted by the processes(i.e. $p_{id}1$ for all processes) and chooses the winner to be the process with the lowest id, that succeeded inserting at least a single value to the set.

havior; since if no one deletes a value from the set, then the number of insert operations is in fact limited.

7 *f-bounded-useQ*'s Implementation Using *1-bounded-useSwap*

In this section we show that one can implement an *f-bounded-useQ* using *1-bounded-useSwap* registers. This result is interesting, since it implies that the functionality of a queue,[3] for any finite number of enqueuers and dequeuers, is implementable using bounded-use objects whose long-lived counterparts are members of the common2. This result should not be surprising, since the *1-bounded-useSwap* object has higher consensus number than the *f-bounded-useQ*; nevertheless it implies that the problem of implementing a queue out of common2 members, or proving that such an implementation does not exist, might not rely only on the core FIFO functionality, even though in [9] it is shown that LIFO (stack) is in common2.

An implementation of *f-bounded-useQ* based on Test*&*Set[4] objects and *1-bounded-useSwap* registers, is given in Algorithm 5.. We prove its correctness in Sect. A.2.

Algorithm 5. *f-bounded-useQ*'s implementation

shared objects

```
  s : array of f, ⊥ initialized, 1-bounded-useSwap registers
  t : array of f Test&Set registers
```

local variables

```
  j : loop integer, local for each function
```

```
Dequeue() : outputvalue
d1:  for j = 0 to f-1 do
d2:    if (s[j].Read()≠ ⊥)[1]
d3:      if (t[j].Test&Set())[2]
d4:         return s[j].Read();
d3e:     fi;
d2i:   else
d5:      return empty;
d2e:   fi;
d1e: rof;
d6:  return empty;
end;
1. did someone enqueue to this place
2. did someone return or plan to return
   the value in s[j]
```

```
Enqueue(v : inputvalue)
e1:  for j = 0 to f-1 do
e2:    if (s[j].Swap(v)==⊥)[1]
e3:      return true;
e2e:   fi;
e1e: rof;
e4:  return fail;
end;
1. first to enqueue to this place
```

[3] i.e. enabling removal of elements in the order of their entrance.

[4] Interestingly Test*&*Set are bounded-use by definition.

8 Concluding Remarks

This paper presents the bounded-use property that when imposed on some objects with consensus number 2, increases their consensus number. We pointed out that the consensus number of bounded-size objects might also be higher than their unbounded counterparts; we relate this result with the bounded-use property. These observations underline additional properties which impact the consensus number of objects. Moreover these observations emphasize the fact that complex object behavior sometimes has higher consensus number, as it is for the well known relation between Compare*&*Swap and Swap.

In Sect. 7 we provided a simple implementation of an *f-bounded-useQ* using *Test&Set* and *1-bounded-useSwap* registers. The existence of such implementation should not be surprising. Nevertheless its simplicity and existence prove that queue's FIFO behavior, is implementable using swap functionality registers, thus an impossibility proof for showing that queue is not in common2, might not rely on the core FIFO functionality, despite of the fact that in [9] it is shown that LIFO (stack) is in common2.

References

1. Y. Afek , E. Weisberger and H. Weisman. A completeness theorem for a class of synchronization objects. Proceedings of the twelfth annual ACM symposium on Principles of distributed computing, p.159-170, August 15-18, 1993, Ithaca, New York, United States
2. M. Herlihy. Wait-Free Synchronization, ACM Transactions on Programming Languages and Systems, Jan. 1991, pages 124-149.
3. Z. Li. Non-blocking implementation of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001
4. M. David. A Single-Enqueuer Wait-Free Queue Implementation. Lecture Notes in Computer Science (Proceedings of DISC 2004), Volume 3274, Jan. 2004, pages 132-143.
5. M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In Proceedings of the 14th ACM Symposium on Principles of Programmmg Languages, Jan. 1987, pages 13-26.
6. M. P. Herlihy and J. M. Wing. Linearizability:A Correctness Condition for Concurrent Objects. ACM Transactions of Programming Languages and Systemsj Vol. 12, No. 3 pages 463-492, 1990.
7. M. Fisher, N. Lynch and M. Patterson. Impossibility of distributed consensus with one faulty process. J. ACM 32(2), Apr. 1985, pages 274–382,
8. P. Jayanti and S. Toueg. Some Results on the Impossibility,Universality, and Decidability of Consensus. In Proceedings of the 6th International Workshop on Distributed Algorithms, pages 69-84. Springer Verlag, Nov. 1992.
9. Y. Afek, E. Gafni and A. Morrison. Common2 extended to stacks and unbounded concurrency. In Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC 2006)
10. Y. Afek and G. Stupp. Delimiting the power of bounded size synchronization objects. In Proceedings 13th ACM Symp. on Principles of Distributed Computing, pages 42–51, August 1994.

11. Y. Afek and G. Stupp. Synchronization power depends on the register size. In Proceedings of the 34th IEEE Ann. Symp. on Foundation of Computer Science, pages 196–205. IEEE Computer Society Press, November 1993.
12. S. A. Plotkin. Sticky bits and universality of consensus. In Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, pages 159-175, August 1989.

A Appendix

A.1 *f-bounded-useQ* Can Not Solve 4-Process Consensus

In this section we show that the *f-bounded-useQ* can not solve 4-process consensus. We follow the proof technique used in [2].

Notations. We use s to indicate a critical state as part of a consensus protocol, s' or s'_i to indicate x-valent states, and s'' or s''_i indicate y-valent states, $x \neq y$. In addition the following notations are used:

- d_i^v, d_i^{ep}, d_i^{dc} - indicate a dequeue operation by process p_i that returns either v or empty or we don't care what its output is, respectively.
- $e(v)_i^{ok}$, $e(v)_i^{fd}$ - indicate an enqueue operation by process p_i that respectively either succeeds or fails to enqueue the value v.

Example: Let s be a critical state then $s' = s \;\; e(v)_1^{fd} \;\; d_2^{dc}$ - indicates that a state s' is reached from state s in case operations are done in the following order: p_1 fails to enqueue a value v, then p_2 dequeues an element.

The Proof. Let $p_1,\ p_2,\ p_3,\ p_4$ be four processes that execute a binary consensus protocol. By [2] and [7] for any consensus algorithm there is a bivalent initial state, from which a critical state s is reachable. Following [2], the pending steps of all the processes in the critical state s can only be applied on the same *f-bounded-useQ* object (otherwise one can easily show that s is not a critical state). By manipulating the scheduler we bring the system into a critical state s, and we focus on the case in which the next step by any of the processes is on the same *f-bounded-useQ* object. Let Q be the *f-bounded-useQ* accessed by the pending operations. Several cases must be considered: whether Q is exhausted or not, and the different combinations of the pending steps. Let us examine two pending operations each leads to a different valent state, without loss of generality these operations are done by processes p_1 and p_2. The rest is case analysis.

Case 1: The pending operation of both p_1 and p_2 is a dequeue

Case 2: p_1's pending operation is enqueue, and p_2's pending operation is dequeue
Both these cases follow standard arguments shown in [2],[5] thus are omitted from this paper.

[5] Showing that a third process can not decide different values depending on the order of execution.

Case 3: The pending operation of both p_1 and p_2 is an enqueue
3.a: Q is exhausted. Let $s' = s \;\; e(v)_1^{fd}$ and $s'' = s \;\; e(w)_2^{fd}$ each a univalent state with a different decision value. Notice that in both s' and s'' the enqueue has no effect on Q's state. Let β be a p_4 only schedule from either s' or s''. Since s' and s'' are indistinguishable by p_4, the same decision value is reached by p_4 in $s'\beta$ and $s''\beta$, a contradiction.

3.b: Q is not exhausted. In this case Q might have either one or more free places at state s. Let $s_1' = s \;\; e(v)_1^{ok}$ and $s_1'' = s \;\; e(w)_2^{ok}$ each a univalent state with a different decision value. Notice that in case that Q has only one free place at state s, as soon as the first successful enqueue had taken place by either of the processes, Q becomes exhausted. However this happens in both s_1' and s_1'', thus the exhaustion state of Q does not help either p_3 or p_4 to distinguish between s_1' and s_1''. If p_3 was to run solo from either s_1' or s_1'' then in order for p_3 to be able to distinguish between s_1' and s_1'', p_3 must eventually perform a dequeue that returns either v or w. Let ξ_3^x be a p_3 only schedule from either s_1' or s_1'', such that the last step executed by p_3 in ξ_3^x is a dequeue that results in x (either v or w). Since s_1' and s_1'' are indistinguishable by p_3, it issues the same operations on the same objects, during the run segments ξ_3^v and ξ_3^w. We now stop p_3 from executing any other operation on any object. Let $s_2' = s_1' \;\; \xi_3^v$ and $s_2'' = s_1'' \;\; \xi_3^w$ each a univalent state with a different decision value. Notice that in both states, s_2' and s_2'', Q's exhaustion state is the same (either exhausted or not) and Q contains the same elements (if any). Let β be a p_4 only schedule from either s_2' or s_2''. Since s_2' and s_2'' are indistinguishable by p_4, the same decision value is reached by p_4 in $s_2'\beta$ and $s_2''\beta$, a contradiction.

We have finished the case analysis, and provided a contradiction to the different valency, hence to the fact that s is a critical state in all cases. □

A.2 *f-bounded-useQ*'s Implementation Using *1-bounded-useSwap*

Algorithm 5.'s proof of correctness
For the implementation to be correct it has to be wait-free, linearizable and valid, that is perform as a *f-bounded-useQ*, i.e., preserving a FIFO order among at most f successful enqueue operations; each element might be dequeued at most once. We use the notations from Sect. A.1. In addition we use $Op1 \rightarrow Op2$ to denote that $Op1$ precedes $Op2$.

Linearizability & Validity. When successful, an enqueue operation is linearized at e_2, otherwise it is linearized at e_4. The dequeue operation is linearized at d_3 when returning a value, otherwise it is linearized at either d_5 or d_6. The following lemmas prove validity and linearizability.

Lemma 16. *Algorithm 5. implements a FIFO order.*

Proof. Notice that the place at which a new value is stored is determined according to the first *1-bounded-useSwap* register the enqueuer succeeds swapping its value with. Since we use *1-bounded-useSwap* registers, it is clear that exactly one process succeeds to swap to each of s' registers, thus elements already stored in a *1-bounded-useSwap* are not overridden by later swap operations. The combination of lemmas 17 and 18 proves that FIFO order is indeed preserved. Proofs of these lemmas follow immediately from the code, thus omitted. □

Lemma 17. *Elements of non-overlapping enqueue operations are stored in an incremental order (i.e. first* $s[0]$ *then* $s[1]$ *and so on).*

Lemma 18. *Elements of non-overlapping dequeue operations are removed in a FIFO order (i.e. first* $s[0]$ *then* $s[1]$ *and so on).*

Lemma 19. *At most* f *enqueue operations can ever succeed.*

Proof. In order for an enqueue executed by a process p to succeed, p has to succeed in swapping the value into one of s' *1-bounded-useSwap* registers. We use *1-bounded-useSwap* registers that are initialized with $\perp$, hence exactly one process can successfully swap to each of these registers. Since there are only f such registers, there can be at most f successful enqueues. □

Lemma 20. *An element can be dequeued at most once.*

Proof. Assume not. Let b be an element that is dequeued twice once by process p_1 and once by process p_2, both execute a dequeue operation (operations might be overlapping). Without loss of generality assume b is stored in $s[0]$. Notice that d_1^b implies that: $p_1.t[0].Test\&Set() = true$, but d_2^b implies that: $p_2.t[0].Test\&Set() = true$. A contradiction to the *Test&Set* specification. □

Lemma 21. *A false fail on enqueue scenario does not exist.*

Proof. Let p be a process that tries to enqueue a value v and returns fail (thus the operation is linearized to e_4). Notice that p's search for a storing place is a brute force search hence in order for it to get to e_4 it goes over all f *1-bounded-useSwap* registers in s, and tries to swap them. In case it fails it means that all are taken, hence the fail result is not false. □

Lemma 22. *A false empty on dequeue scenario does not exist.*

Proof. By negation assume there exists a false empty scenario. Let p_1 be the process which dequeues and returns false empty. At the beginning of the dequeue execution there is at least one element in the queue, assume there are z elements and that the **youngest**[6] element resides in $s[x]$ $(0 \leq x < f - 1)$. The dequeue operation is a brute force search for an element to dequeue. In order for p_1 to return empty, p_1 goes over all swap registers including $s[x]$, thus p_1 must see that $s[x]$ is not empty. Next p_1 tries to Test&Set $t[x]$. Since p_1 returns empty,

[6] Assume that p_1 is the slowest process and meanwhile $z - 1$ elements were dequeued.

then it failed to Test*&*Set $t[x]$, however if it failed then some other process, let it be p_2, succeeded (right after p_1 started its dequeue otherwise the element is not considered to be inside the queue in the first place). However if p_2 dequeues the element from $s[x]$ then since $s[x]$ stored the youngest element then the queue is indeed empty. Thus we are interested in the case in which before p_2 removes the element from $s[x]$ (by successful Test*&*Set of $t[x]$), another process, let it be p_3, enqueues a new element, and only then p_2 completes its dequeue, thus the queue is not empty after p_2's dequeue. Notice that the new value is inserted at $s[x+1]$. However when p_1 reads $s[x+1]$ it must see that it contains a value, and it tries to Test*&*Set $t[x+1]$. Again if it fails to Test*&*Set then another process must have succeeded, and we continue using the previous argument until either the queue is empty, or until we reach the f^{th} element ever enqueued, meaning that the queue is exhausted, and no other enqueues could occur, which means that the queue is indeed empty. □

Wait Freedom. All operations used by either of the functions are wait free. The for-loops on both enqueue and dequeue operations are finite. Thus every invocation of either of these functions is to be ended within a finite number of steps. Hence the implementation is wait free. □

One-Step Consensus Solvability

Taisuke Izumi and Toshimitsu Masuzawa

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, 560-8531, Japan
{t-izumi, masuzawa}@ist.osaka-u.ac.jp

Abstract. While any fault-tolerant asynchronous consensus algorithm requires two communication steps even in failure-free executions, it is known that we can construct an algorithm terminating in one step for some good inputs (e.g. all processes propose a same value). In this paper, we present the necessary and sufficient constraint for the set of inputs for which we can construct an asynchronous consensus algorithm terminating in one step. Our investigation is based on the notion of the condition-based approach: it introduces conditions on input vectors to specify subsets of all possible input vectors and condition-based algorithms can circumvent some impossibility if the actual input vector satisfy a particular condition. More interestingly, conditions treated in this paper are adaptive. That is, we consider hierarchical sequences of conditions whose k-th condition is the set of input vectors for which the consensus can be solved in one step if at most k processes crash. The necessary and sufficient constraint we propose in this paper is one for such condition sequences. In addition, we present an instance of the sufficient condition sequences. Compared with existing constraints for inputs this instance is more relaxed.

1 Introduction

The *consensus* problem is one of fundamental and important problems for designing fault-tolerant distributed systems. In the consensus problem, each process proposes a value, and all non-faulty processes have to agree on a common value that is proposed by a process. The *uniform consensus*, a stronger variant of the consensus, further requires that faulty processes are disallowed to disagree (Uniform Agreement). The (uniform) consensus problem has many practical applications, e.g., atomic broadcast [3,10], shared object [1,11], weak atomic commitment [8] and so on. While it is a very important task to build an efficient consensus primitive on the system because of such applications, it has no deterministic solution in asynchronous systems subject to only a single crash fault [5]. Thus, to circumvent this impossibility, several approaches, such as *eventual synchrony*, *unreliable failure detectors*, and so on, have been proposed. However, even using such approach, it is not an easy task to solve the consensus problem "efficiently". One of commonly used measurements to evaluate efficiency of algorithms is *communication steps*, one of which is an execution period where each pair of processes can concurrently exchange messages at most once. In asynchronous systems with some assumptions to solve the consensus problem, it is proved

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 224–237, 2006.

that any fault-tolerant consensus algorithm requires at least two communication steps for decisions even in the run where no crash fault occurs [13].

To circumvent this two-step lower bound, some papers investigate consensus algorithms that achieve one-step decision in some good input cases. The first result of such investigations is published by Brasileiro et al. [2]. On the assumption of the underlying non-one-step consensus primitive, this paper proposes a simple algorithm that correctly solves the consensus problem for any input and that especially achieves the one-step decision if all processes propose a same value. In other results [4][9], the one-step decision scheme is also considered in the context of efficient combination with other schemes such as randomization and failure detectors. However, these results leave an interesting and important question as follows: for what input can the consensus problem be solved in one step?

In this paper, we address this question based on the notion of the *condition-based approach.* The principle of the condition-based approach is to restrict inputs so that the generally-unsolvable problem can become solvable. A *condition* represents some restriction to inputs. In the case of the consensus problem, it is defined as a subset of all possible *input vectors* whose entries correspond to the proposal of each process. The first result of the condition-based approach clarifies the conditions for which the uniform consensus can be solved in asynchronous systems subject to crash faults [14]. More precisely, this result presented a class of conditions, called *d-legal conditions*, and proved that the d-legal conditions are the class of necessary and sufficient conditions that make the (uniform) consensus solvable in asynchronous systems where at most d processes can crash. In previous results, the condition-based approach is used to overcome several impossibility results in distributed agreement problems [6,12,15,16,17,18,19,20]. We also use the notion of the condition-based approach to overcome the two-step lower bound of asynchronous consensus. In the same way as [2], this paper assumes the underlying non-one-step consensus primitive. On this assumption, the main objective of our study is to clarify the class of conditions such that we can construct the algorithm that terminates in one step for the inputs belonging to the condition and that even terminates (but not in one step) for any input out of the condition.

The contribution of this paper is to characterize such class of the necessary and sufficient conditions that make the uniform consensus terminate in one step. More interestingly, the condition we consider in this paper is *adaptive* in the sense of our previous result [12]. In the adaptive condition-based approach, a restriction for inputs is not represented by a single subset of all possible inputs, but represented by a hierarchical sequence of conditions called *condition sequence.* An adaptive condition-based algorithm is instantiated by a condition sequence, and guarantees some property according to the rank of the input vector in the hierarchy of the given condition sequence. For example, the first result for the adaptive condition-based approach [12] considers time complexity lower bound in synchronous consensus. In this result, all input vectors are classified into some hierarchical condition sequence whose k-th condition is the set of input vectors that reduce the worst-case time complexity of synchronous consensus by k, and we construct the algorithm achieving time reduction according to the rank of the actual input vector in the hierarchy. This paper considers the consensus algorithm instantiated by an *one-step condition sequence*, whose k-th condition is the set of input vectors for which the algorithm can terminate in one-step even if at most k processes crash. We present a property of condition sequences called *one-step legality*, and prove that a condition sequence can become the one-step condition sequence of some algorithm if and only if it is one-step legal. We introduce *root adjacency graphs*, which is an analysis tool for specifying the property of one-step legality. The notion of root adjacency graphs is based on the idea of graph representation of conditions proposed in [14]. An root adjacency graph is also a graph representation of a condition sequence, and the one-step legality property for a condition sequence is defined as the characterization of its root adjacency graph. Additionally, we also propose an instance of one-step legal condition sequences. Compared with existing constraints (i.e., all processes propose a same value), this instance is more relaxed.

The paper is organized as follows: In section 2, we introduce the system model, the definition of the consensus problem, and other necessary formalizations. Sections 3 and 4 provide the characterization theorem of one-step consensus solvability and its correctness proof. In Section 5, we present an example of one-step legal condition sequences. We conclude this paper in Section 6.

2 Preliminaries

2.1 Asynchronous Distributed Systems

A distributed system consists of n processes $\mathcal{P} = \{p_0, p_1, p_2, \cdots, p_{n-1}\}$, in which any pair of processes can communicate with each other by exchanging messages. All channels are reliable; neither message creation, alteration, nor loss occurs. The system is asynchronous in the sense that there is no bound on communication delay.

Each process can crash. If a process crashes, it prematurely stops its execution and makes no operation subsequently. Each process can crash at any timing. A process that does not crash (even in the future) is called a *correct* process. We assume that there is some upper bound t on the number of processes that can crash in the whole system. Every process knows the value of t a priori. The actual number of crash faults is denoted by f. The value of f is unknown to each process.

Formally, a process is modeled as a state machine. The communication is defined by two events, $\mathsf{Send}_i(m, p_k)$, and $\mathsf{Receive}_i(m, p_k)$. The event $\mathsf{Send}_i(m, p_k)$ is one that p_i sends message m to the process p_k. The event $\mathsf{Receive}_i(m, p_k)$ is one that p_i receives the message m from p_k. A process crash is also defined as an event. An event Crash_i means the crash of process p_i. We also assume that algorithms we consider in this paper are *deterministic*, i.e., the state after a transition is uniquely determined by the triggering event and the state immediately before the transition.

2.2 Uniform Consensus

A (uniform) consensus algorithm provides each process p_i with two events, $\mathsf{Propose}_i(v)$ and $\mathsf{Decide}_i(v)$, as the interface to the upper application layer. In a consensus algorithm, each correct process p_i initially proposes a value v by $\mathsf{Propose}_i(v)$, and eventually chooses a decision value v' by $\mathsf{Decide}_i(v')$. Then, the decision value must be chosen from the values proposed by processes so that all processes decide a same value. More precisely, the consensus problem is specified by the following three properties:

Termination: Every process p_i eventually invokes $\mathsf{Decide}_i(v)$ unless it crashes.

Uniform Agreement: If two events $\mathsf{Decide}_i(v_1)$ and $\mathsf{Decide}_j(v_2)$ are invoked, $v_1 = v_2$ holds.

Validity: If $\mathsf{Decide}_i(v)$ is invoked, then $\mathsf{Propose}_j(v)$ is invoked by some process p_j.

We define $\mathcal{V}$ as the set of all possible proposal values. Throughout this paper, we assume that $\mathcal{V}$ is a finite ordered set. An input to consensus algorithms is represented by a vector whose i-th entry is the value of p_i's proposal value. We call it an *input vector*.

2.3 Uniform Consensus Primitive

This paper investigates the inputs for which consensus algorithms can decide in one step. However, it is well-known that the consensus problem cannot be solved in asynchronous systems subject to only one crash fault. Thus, we need some assumption to guarantee correct termination for arbitrary inputs. In this paper, same as the previous result [2], we assume that a uniform consensus primitive is equipped to the system. This assumption can be regarded as an higher abstraction of other standard assumptions, such as *unreliable failure detector* or *eventual synchrony*, which are sufficient ones to solve the consensus problem. On this assumption, our aim is to provide an algorithm that decides in one step for good inputs and that solves consensus (but not in one step) for any input with support of the underlying uniform consensus primitive. In the following discussion, two events, $\mathsf{UC\_propose}(v)$ and $\mathsf{UC\_decide}(w)$, are provided by the underlying consensus, which mean the proposal of value v and the decision with value w respectively.

2.4 Configurations and Executions

A system configuration c is represented by all processes' states and the set of messages under transmission. An execution of a distributed system is an alternative sequence of configurations and events $E = c_0, e_0, c_1, e_1, c_2 \cdots$. In this paper, we deal with *admissible* executions, where occurrences of send, receive, propose and decide of the underlying consensus, and crash events satisfy those semantics.

2.5 Nonessential Assumptions

The asynchronous system model introduced in this section is the standard one as defined in [3]. In this subsection, for ease of presentation, we introduce some additional assumptions into the model. Notice that the introduced assumptions do not essentially differentiate our model from standard ones. Throughout this paper, we assume the followings:

- There exists a discrete global clock, and that each event has one time when it occurs. This global clock is a fictional device. That is, each process does not have access to the global clock (thus, it adds no additional power to the model).
- Local processing delay is negligible (i.e., any local computation is instantaneously processed).
- Each process p_i invokes $\mathsf{Propose}_i(v)$ at time zero unless it initially crashes.
- Any message has at least one time unit delay.

2.6 One-Step Decision of Consensus Problem

In this subsection, we introduce the definition of one-step decision in the consensus problem.

An *initial message* is one that is sent at time zero. Intuitively, an initial message sent by a process p_i is one whose sending event is triggered by p_i's activation of the consensus algorithm. We say that a message m is *over* at time u if the receiver of m has received m or crashed at u. Let $ot(E)$ be the minimum time when all initial messages are over in execution E. Then, the prefix of the execution E by time $ot(E)$ (including transitions occurring at time $ot(E)$) is called the *one-step prefix* of E, and is denoted by *pref*(E). We also define $E(\mathcal{A}, I)$ as the set of all admissible executions of a consensus algorithm $\mathcal{A}$ whose input vectors are I.

Using the above definitions, we define one-step decision as follows:

Definition 1 (One-step Decision). A consensus algorithm $\mathcal{A}$ decides in one step for an input vector I if for any execution $E \in E(\mathcal{A}, I)$ all processes decide or crash in *pref*(E).

3 Characterization of One-Step Consensus Solvability

3.1 Notations

For an input vector I, we define a *view* J of I to be a vector in $(\mathcal{V} \cup \{\perp\})^n$ obtained by replacing several entries in I by $\perp$ ($\perp$ is a default value such that $\perp \notin \mathcal{V}$). Let $\perp^n$ be the view such that all entries are $\perp$. For views J_1 and J_2, the containment relation $J_1 \leq J_2$ is defined as $\forall k (0 \leq k \leq n-1) : J_1[k] \neq \perp \Rightarrow J_1[k] = J_2[k]$. We also describe $J_1 < J_2$ if $J_1 \leq J_2$ and $J_1 \neq J_2$ hold. For a view J $(\in (\mathcal{V} \cup \{\perp\})^n)$ and a value $a (\in \mathcal{V} \cup \{\perp\})$, $\#_a(J)$ denotes the number of entries of value a in the view J, that is, $\#_a(J) = |\{k \in \{0, 1, \cdots, n-1\} | J[k] = a\}|$. For a view J and a value a, we often describe $a \in J$ if there exists a value k such that $J[k] = a$. For an input vector $I \in \mathcal{V}^n$, For two views J_1 and J_2, let $\text{dist}(J_1, J_2)$ be the Hamming distance between J_1 and J_2, that is $\text{dist}(J_1, J_2) = |\{k \in \{0, 1, \cdots, n-1\} | J_1[k] \neq J_2[k]\}|$. A *condition* is a subset of all possible input vectors $\mathcal{V}^n$.

Let $[V^n]_k$ be the set of all possible views where $\perp$ appears at most k times.

3.2 One-Step Condition Sequence

The objective of this paper is not only to clarify the static conditions that enable the consensus problem to terminate in one step, but also to provide such conditions in an adaptive fashion: the content of a condition varies according to the number of actual faults. To handle such adaptiveness, we introduce *condition sequences*. Formally, a condition sequence S is a hierarchical sequence of $t+1$ conditions $(C_0, C_1, C_2, \cdots, C_t)$ satisfying $C_k \supseteq C_{k+1}$ for any $k (0 \leq k \leq t-1)$. Then, we define *one-step condition sequences* as follows.

Definition 2 (One-Step Condition Sequence). The *one-step condition sequence* of a consensus algorithm $\mathcal{A}$ is the condition sequence whose k-th condition $(0 \leq k \leq t)$ is the set of input vectors for which the algorithm $\mathcal{A}$ decides in one step when at most k processes crash.

The one-step condition sequence of an algorithm $\mathcal{A}$ is denoted by $Sol^{\mathcal{A}}$.

3.3 Characterization Theorem

This subsection presents the characterization theorem for one-step consensus solvability. The key idea of the characterization theorem derives from the notion of *legality* in [14]: we consider a graph representation of condition sequences, and the characterization is given as a property of such graphs. To provide the theorem, we first introduce *root adjacency graphs* and their legality. Root adjacency graphs are a variant of the graph representation of legal conditions proposed in [14] so that it can handle the one-step solvability and condition sequences.

Definition 3 (Decidable/Undecidable views). For a condition sequence $S = (C_0, C_1, \cdots, C_t)$, the *decidable views* $DV(S)$ and *undecidable views* $UV(S)$ are respectively the set of views defined as follows:

$$DV(S) = \bigcup_{k=1}^{t} \{J | dist(J, I) \leq k, I \in C_k\}$$

$$UV(S) = \bigcup_{J \in DV(S)} \{J' | J' \in [V^n]_t, J' \leq J\}$$

Notice that $DV(S) \subseteq UV(S)$ holds.

Definition 4 (Root Adjacency Graphs). Given a condition sequence S, its root adjacency graph $RAG(S)$ is the graph such that

- The vertex set consists of all views in $UV(S)$.
- The two views J_1 and J_2 are connected if $J_1 \leq J_2$ holds and J_2 belongs to $DV(S)$.

The legality of root adjacency graphs is defined as follows:

Definition 5 (Legality). A root adjacency graph G is *legal* if, for each connected component Com of G, at least one common value appears at all views belonging to Com.

We say a condition sequence S is *one-step legal* if $RAG(S)$ is legal. Using the above definitions, we state the characterization theorem.

Theorem 1 (One-step Consensus Solvability Theorem). There exists a consensus algorithm whose one-step condition sequence is S if and only if S is one-step legal.

The intuitive meaning of root adjacency graphs is explained as follows: In general, the information that a process can gathers in one-step prefixes can be represented by a view because the information each process can send in one-step prefixes is only its proposal. In this sense, the decidable views can be interpreted as the set of views J such that if a process gathers the information corresponding J in one step , it can immediately decides (i.e., one-step decision). The undecidable views can also be interpreted as the set of views J such that if a process gathers the information corresponding J in one step, it must consider the possibility that other processes may decide in one step (but it does not have to decide immediately). Then, a root adjacency graph can be regarded as one obtained by connecting two views J_1 and J_2 such that if two processes gathers J_1 and J_2 respectively, they must reach a same decision. Thus, the sentence "root adjacency graphs is legal" implies that there exists at least one possible decision value.

For any view $J \in DV(S)$, there exists an input vector I satisfying $\mathsf{dist}(J, I) \leq k$ and $I \in C_k$ for some k. In such vectors, we call one that maximizes k the *master vector* of J (if two or more vectors maximize k, one chosen by some (arbitrary) deterministic rule is the master vector of J). Then, we also call the value of k *legality level* of the master vector. For any view $J \in DV(S)$ and its master vector I with legality level k, there exists a view J' satisfying $J' \leq J$ and $J \leq I$. The view J' minimizing $\#_\perp(J')$ is called the root view of J, and denoted by $Rv(J)$. Notice that $Rv(J) \in DV(S)$ and $\#_\perp(J') \leq k$ necessarily hold because of $I \in C_K$ and $\mathsf{dist}(J, I) \leq k$.

4 Proof of the Characterization Theorem

4.1 Proof of Sufficiency

This subsection presents the sufficiency proof of the theorem, i.e., we propose a generic one-step consensus algorithm for any one-step legal condition sequence S and prove its correctness.

Figure 1 presents the pseudo-code description of a generic consensus algorithm OneStep that is instantiated by any one-step legal condition sequence S. In the description, we use the function h, that is the mapping from a view in $UV(S)$ to a value in $\mathcal{V}$. The mapped value $h(J)$ for a view J is one that appears in common at any view in the connected component to which J belongs (such value necessarily exists from the fact that S is one-step legal). If two or more values appear in common, the largest one is chosen. The algorithmic principle of OneStep is as follows: first, each process p_i exchanges its proposal with each other, and constructs a view J_i. The view J_i is maintained incrementally. That is, it is updated on each reception of a message. When at least $n-t$ messages are received by p_i, process p_i tests whether J_i belongs to the undecidable views $UV(S)$ or not. If it belongs to $UV(S)$, process p_i activates the underlying consensus with proposal $h(J_i)$. Otherwise, p_i activates the underlying consensus with proposal $J_i[i]$. In addition, when the view J_i is updated, each process p_i tests whether its view J_i belongs to the decidable views $DV(S)$ or not. If J_i belongs to $DV(S)$, process p_i immediately decides $h(J_i)$, that is, it decides in one step. When the underlying consensus reaches decision, each process simply borrows the decision of the underlying consensus unless it has already decided in one step. Intuitively, the correctness of the algorithm OneStep relies on two facts: one is that if two processes p_i and p_j decide in one-step, the views J_i and J_j at the time of their decisions are necessarily connected in $RAG(S)$, and another one is that if a process p_i decides v in one-step, each process p_j activates the underlying consensus with the proposal value v. From the former fact, we can show that two processes p_i and p_j, both of which decide in one step, have a same decision. The latter fact implies that if a process p_i decides v in one step, any other process p_j (that may not decide in one step) propose v. The detailed explanation of correctness is given in the following proof.

Correctness. We prove the correctness of the algorithm Onestep. In the following proofs, let vector J_i^{pro} and J_i^{dec} be the the value of J_i at the time when p_i execute the lines 13 and 10 respectively (Notice that both values are uniquely defined because lines 13 and 10 are respectively executed at most once). If p_i does not execute line 13 (10), J_i^{pro} (J_i^{dec}) is undefined.

Lemma 1 (Termination). Each process p_i eventually decides unless it crashes.

Proof. Since at most t processes can crash, each process p_i receives at least $n-t$ messages. Then, p_i necessarily activates the underlying consensus, and thus the decision of underlying consensus eventually occurs on p_i unless p_i crashes. This implies that p_i eventually decides unless it crashes. □

Lemma 2 (Uniform Agreement). No two processes decide differently.

Proof. Let p_i and p_j be the processes that decide, and v_i and v_j be the decision values of p_i and p_j respectively. The input vector is denoted by I. Then, we prove $v_i = v_j$. We consider the following three cases.

```
Algorithm OneStep for one-step legal condition sequence $S$
Code for $p$ :

1:  variable:
2:      $proposed$ , $decided$ : FALSE
3:      $J$ : init $\perp$

4:  Upon Propose ($v$ ) do:
5:      $J$ $[i] \leftarrow v$
6:      Send $v$ to all processes (excluding $p$ );

7:  Upon Receive ($v$) from $p$ do:
8:      $J$ $[j] \leftarrow v$
9:      if $J$ $\in DV(S)$ and $decided$ $\neq$ TRUE then
10:         $decided$ $\leftarrow$ TRUE ; Decide ($h(J$ $))$
11:     endif
12:     if $\#_\perp(J$ $) \leq t$ and $proposed$ $=$ FALSE then
13:         if $J$ $\in UV(S)$ then UC_propose ($h(J$ $))$    /* Starting the Underlying Consensus */
14:         else UC_propose($J$ $[i]$) endif
15:         $proposed$ $\leftarrow$ TRUE
16:     endif

17: Upon UC_decide ($v$) do:                       /* Decision of the Underlying Consensus */
18:     if $decided$ $\neq$ TRUE then
19:         $decided$ $\leftarrow$ TRUE ; Decide ($v$)
20:     endif
```

Fig. 1. Algorithm Onestep: An One-Step Consensus Algorithm for a One-Step Legal Condition Sequence S

- (**Case1**) When both p_i and p_j decide at line 10: For short, let $g = \#_\perp(J_i^{dec})$. Both J_i^{dec} and J_j^{dec} appear in $DV(S)$. Let I' be the master vector of J_i^{dec}, and k be its legality level. Then, for any vector I'' that is obtained from J_i^{dec} by replacing $\perp$ by any value, $\mathsf{dist}(I', I'') \leq k$ holds. Thus, letting I be the input vector, $\mathsf{dist}(I', I) \leq k$ holds, and thus we obtain $I \in DV(S)$. In addition, since $J_i^{dec} \leq I$, and $J_j^{dec} \leq I$ holds, I and J_i^{dec}, and I and J_j^{dec} are respectively adjacent to each other in $RAG(S)$. This implies that $v_i = h(J_i^{dec}) = h(J_j^{dec}) = v_j$ holds.
- (**Case2**) When p_i and p_j respectively decide at lines 10 and 19: Since p_j's decision is borrowed from the decision of the underlying consensus primitive, it is a value proposed by some process at line 13 or 14. Thus, it is sufficient to show that every process p_k proposes v_i at line 13 or 14 unless it crashes. Let v_k be p_k's proposal. J_i^{dec} appears in $DV(S)$. Then, by the same argument as Case 1, we can show $I \in DV(S)$. Since $J_k^{pro} \leq I$ and $J_i^{dec} \leq I$ holds, we can conclude $J_k^{pro} \in UV(S)$, that is, p_k propose the value $h(J_k^{pro})$ at line 13. Since J_k^{pro} and J_i^{dec} is connected in $RAG(S)$, $v_k = h(J_k^{pro}) = h(J_i^{pro}) = v_i$ holds. It follows that every process p_k proposes v_i.
- (**Case3**) When both p_i and p_j decide at line 19: Then, from the uniform agreement property of the underlying consensus, $v_i = v_j$ clearly holds.

Consequently, the lemma holds. □

Lemma 3 (Validity). If a process decides a value v, then, v is a value proposed by a process.

Proof. If a process p_i decides at line 15, its decision value is $h(J_i^{dec})$, which is a value in J_i^{dec}. On the other hand, if a process p_i decides at line 19, then, its decision value is a proposal of the underlying consensus. That is, the decision value is $h(J_k^{pro})$ or $J_k^{pro}[k]$ for some p_k, which is also a value in J_k^{pro}. In both cases, the decision value is one of proposals, and thus the validity holds. □

Lemma 4 (One-Step decision). The algorithm OneStep decides in one step for any input vector I belonging to $S_k (k \geq f)$ if at most k processes crash.

Proof. Since each process p_i receives the messages from all correct processes unless it crashes, $\#_{\perp}(J_i) \leq k$ holds eventually. Then, J_i is included in $[S_k]_k$. This implies that p_i decides in one step. □

From Lemma 1, 2, 3, and 4, we can show the following lemma that implies the sufficiency of the characterization theorem.

Lemma 5. For any one-step legal condition sequence S, there exists one-step consensus algorithm whose one-step condition sequence is S.

4.2 Proof of Necessity

This subsection presents the necessity proof of the characterization theorem. In the proof we consider a subclass of admissible executions called *P-block synchronous executions*, which are defined as follows:

Definition 6 (P-Block Synchronous Executions). For a set of processes P, P-block synchronous executions are defined as ones satisfying the following properties:

1. No $\mathsf{UC\_decide}_i(v')$ occurs at time zero or one.
2. All message transferred between two processes in P have one time unit delay, and others have two time unit delay.

For a view, J, let $P(J)$ be a set of processes p_i such that $J[i] \neq \perp$ holds. For a consensus algorithm $\mathcal{A}$, a view J, and a set of processe P, let $E_{Sync}(\mathcal{A}, J, P)$ be the set of all possible P-block synchronous executions where process $p_i \in P(J)$ proposes $J[i]$ and never crashes, and $p_i \notin P(J)$ initially crashes. Here, we define *representative executions* of a view J as follows:

Definition 7 (Representative Executions). For a consensus algorithm $\mathcal{A}$ and a view J, its representative executions $E_{Rep}(\mathcal{A}, J)$ is a set of executions defined as follows:

$$E_{Rep}(\mathcal{A}, J) = \begin{cases} E_{Sync}(\mathcal{A}, J, \mathcal{P}) & \text{if } J \notin DV(Sol^{\mathcal{A}}) \\ E_{Sync}(\mathcal{A}, J, P(Rv(J))) & \text{if } J \in DV(Sol^{\mathcal{A}}) \end{cases}$$

For a consensus algorithm $\mathcal{A}$ and a view J, let $val(\mathcal{A}, J)$ be the set of all decision values that appear at executions in $E_{Rep}(\mathcal{A}, J)$.

Using the above notations, we prove the following lemma that implies the necessity of the characterization theorem (for lack of space, we only give a part of all proofs).

Lemma 6. Let $\mathcal{A}$ be an consensus algorithm, and J be a view in $DV(Sol^{\mathcal{A}})$. Then, in any execution $E \in E_{Rep}(\mathcal{A}, J)$, each process in $P(Rv(J))$ decides in one step.

Proof. Let I be the master vector of J, and k be its legality level. From the definition of root views, $Rv(J) \leq I$ holds. Since $\#_{\perp}(Rv(J)) \leq k$ holds, in any execution $E \in E_{Rep}(\mathcal{A}, J)$, at most k processes crash. Thus, any execution $E \in E_{Rep}(\mathcal{A}, J)$ can be regarded as one where input vector is $I \in C_k$ and the number of crash processes is at most k. This implies that each process achieves one-step decision in E.

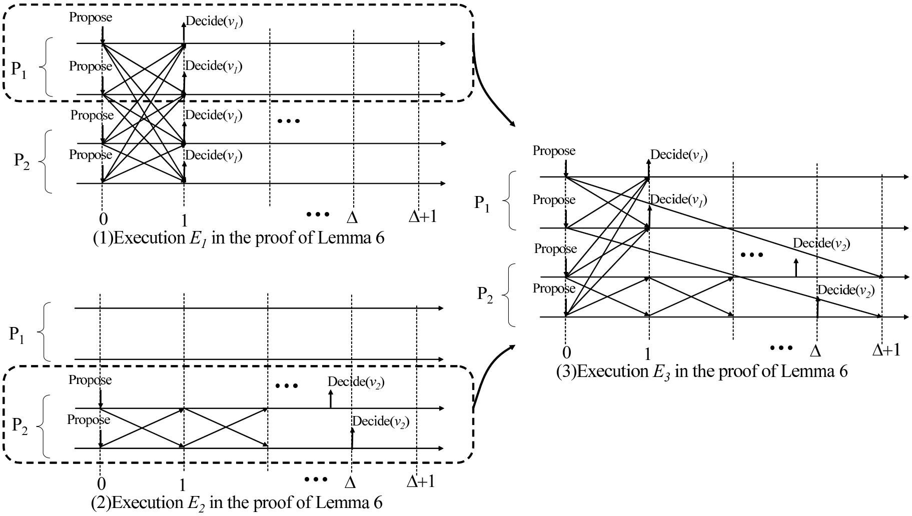
Lemma 7. Let $\mathcal{A}$ be an consensus algorithm. Then, $RAG(Sol^{\mathcal{A}})$ is one-step legal.

Proof. Clearly for any view J, all values in $val(J)$ must be appeared in J from the validity property of the consensus problem. Thus, we prove this lemma by showing $val(J_1) = val(J_2)$ for any two different views J_1 and J_2 that are adjacent to each other in $RAG(Sol^{\mathcal{A}})$. Then, a value in $val(J)$ appears in any view of the connected component to which J belongs, i.e. each connected component has a common value.

Suppose for contradiction that $val(J_1) \neq val(J_2)$ holds for two different views J_1 and J_2 that are adjacent to each other. Without loss of generality, we assume $J_2 < J_1$. Then, from the definition of the $RAG(Sol^{\mathcal{A}})$, J_1 belongs to $DV(Sol^{\mathcal{A}})$. Since we assume $val(J_1) \neq val(J_2)$, there exist two synchronous executions $E_1 \in E_{Rep}(\mathcal{A}, J_1)$ and $E_2 \in E_{Rep}(\mathcal{A}, J_2)$ where processes reach different decisions v_1 and v_2 respectively. In execution E_2, all correct processes eventually reach to the decision. Let Δ be the time when all correct processes decides in E_2. From the fact of $J_2 < J_1$ and $J_1 \leq Rv(J_1)$, there exists at least one process in $P(Rv(J_1))$ that does not crash in E_1 but crashes in E_2. Let P_1 be the set of such processes. We also define P_2 as $\mathcal{P} - P_1$. Then, we consider the execution E_3 obtained by modifying the execution E_2 as follows:

- The behavior of each process in P_2 by time Δ is identical to E_2.
- Each process p_i in P_1 proposes a value $J_1[i]$.
- All messages transferred from a process in P_1 to one in P_2 have $\Delta + 1$ time unit delay, and all other messages have exactly one time unit delay.

The construction of the execution E_3 is illustrated in Figure 2. Since each process in P_1 does not affect ones in P_2 by time $Delta + 1$, the execution E_3 is possible admissible execution of the algorithm $\mathcal{A}$. In execution E_3, each non-faulty process in P_2 decides v_2 at Δ or earlier because it cannot distinguish the execution E_3 from E_2 by time $\Delta + 1$. In both E_1 and E_3, each process in P_1 receives a same set of initial messages at time one because initial messages sent by a process p_i is uniquely determined by p_i's proposal. This implies that each process in P_1 cannot distinguish E_3 from E_1 by time two. Thus, by Lemma 6, in E_3 each process in P_1 decides v_1 at time one. However, since we assume $v_1 \neq v_2$, the execution E_3 has two different decision values (processes in P_1 decides v,



(1)Execution E_1 in the proof of Lemma 6

(2)Execution E_2 in the proof of Lemma 6

(3)Execution E_3 in the proof of Lemma 6

Fig. 2. Executions E_1, E_2 and E_3

and others decides v_2). It contradicts the uniform agreement property of the consensus. □

5 An Example of One-Step Legal Condition Sequence

In this section, we propose an example of one-step legal condition sequences. First, we introduce a condition that are basis of the example.

Definition 8 (Frequency-Based Condition C_d^{freq}). Let $1\text{st}(J)$ be the non-$\perp$ value that appears most often in view J (if two more values appears most often, the largest one is chosen), and $\hat{J}$ be the vector obtained from J by replacing $1\text{st}(J)$ by $\perp$. Letting $2\text{nd}(J) = 1\text{st}(\hat{J})$, the frequency-based condition C_d^{freq} is defined as follows:

$$C_d^{\text{freq}} = \{I \in \mathcal{V}^n | \#_{1\text{st}(I)}(I) - \#_{2\text{nd}(I)}(I) > d\}$$

It is known that C_d^{freq} belongs to *d-legal* conditions, which is the class of conditions that are necessary and sufficient to solve the consensus problem in failure-prone asynchronous systems where at most d processes can crash.

Using this condition, we can construct a one-step condition sequence.

Theorem 2. Letting $3t < n$, a condition sequence, $Sa^{\text{freq}} = (C_t^{\text{freq}}, C_{t+2}^{\text{freq}}, \cdots, C_{t+2k}^{\text{freq}}, \cdots, C_{3t}^{\text{freq}})$ is one-step legal.

Proof. We prove the one-step legality of Sa^{freq} by showing that $1\text{st}(J_1) = 1\text{st}(J_2)$ holds for any two views J_1 and J_2 that are adjacent to each other in

$RAG(Sa^{\mathrm{freq}})$. Suppose $1\mathrm{st}(J_1) \neq 1\mathrm{st}(J_2)$ for contradiction. Since either J_1 or J_2 belongs to $DV(Sa^{\mathrm{freq}})$, we assume $J_1 \in DV(Sa^{\mathrm{freq}})$ without loss of generality. Let I_1 be the master vector of J_1, and k be its legality level. From the definition of C^{freq}_{t+2k}, $\#_{1\mathrm{st}(I_1)}(I_1) - \#_v(I_1) > t + 2k$ holds for any value $v \neq 1\mathrm{st}(I_1)$. Then, since $dist(J_1, I_1) \leq k$ holds, $\#_{1\mathrm{st}(I_1)}(J_1) - \#_v(J_1) > t$ also holds for any value v. This implies $1\mathrm{st}(I_1) = 1\mathrm{st}(J_1)$, and thus we obtain $\#_{1\mathrm{st}(J_1)}(J_1) - \#_v(J_1) > t$ for any v. It follows that $\#_{1\mathrm{st}(J_1)}(J_2) - \#_{1\mathrm{st}(J_2)}(I_1) > 0$ holds because of $J_1 \geq J_2$ and $\#_\perp(J_2) \leq t$. However, it contradicts to the fact that $1\mathrm{st}(J_2)$ is the most often value in J_2. Thus, $1\mathrm{st}(J_2) = 1\mathrm{st}(J_1)$ holds and the lemma is proved. □

Notice that the assumption $3t < n$ is necessary is to achieve one-step decision [2]. That is, if $3t \geq n$, no condition sequence is one-step legal.

Compared with existing constraints (e.g., one that all processes propose a same value), the adaptive condition sequence Sa^{freq} is more relaxed. Thus, the algorithm OneStep instantiated by Sa^{freq} is a strict improvement of existing one-step consensus algorithms.

6 Concluding Remarks

This paper investigated the one-step solvability of consensus problem. While any consensus algorithm require at least two steps even in failure-free executions, we can construct an algorithm that terminates in one step for several good inputs. In this paper, we proposed the necessary and sufficient condition sequences for which we can construct one-step consensus algorithms. We also presented an instance of sufficient condition sequences. Compared with existing constraints for inputs (e.g., all processes propose a same value), this instance is more relaxed.

Acknowledgment. This work is supported in part by MEXT: "The 21st Century Center of Excellence Program", JSPS: Grant-in-Aid for Scientific Research ((B)15300017), MEXT: Grant-in-Aid for Scientific Research on Priority Areas (16092215and MIC: Strategic Information and Communications R&D Promotion Programme (SCOPE).

References

1. H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
2. F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *Proc. of 6th International Conference on Parallel Computing Techn (PACT)*, volume 2127 of *LNCS*, pages 42–50. Springer-Verlag, 2001.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
4. P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *Proc. of the 4th European Dependable Computing Conference on Dependable Computing(EDCC)*, volume 2485 of *LNCS*, pages 191–208. Springer-Verlag, 2002.

5. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
6. R. Friedman, A. Mostéfaoui, S. Rajsbaum, and M. Raynal. Distributed agreement and its relation with error-correcting codes. In *Proc. of 17th International Conference on Ditributed Computing (DISC)*, volume 2508 of *LNCS*, pages 63–87. Springer-Verlag, 2002.
7. E. Gafni. Round-by-round faults detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the 17th annual ACM symposium on Principles of distributed computing*, pages 143–152, 1998.
8. R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proc. of 9th International Workshop on Distributed Algorithms(WDAG)*, volume 972 of *LNCS*, Sep 1995.
9. R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Computers*, 53(4):453–466, 2004.
10. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.
11. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.
12. T. Izumi and T. Masuzawa. Synchronous condition-based consensus adapting to input-vector legality. In *Proc. of 18th International Conference on Distributed Computing(DISC)*, volume 3274 of *LNCS*, pages 16–29. Springer-Verlag, Oct 2004.
13. I. Keider and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2):45–63, 2001.
14. A. Mostéfaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *Journal of the ACM*, 50(6):922–954, 2003.
15. A. Mostefaoui, S. Rajsbaum, and M. Raynal. Using conditions to exppedite consensus in synchronous distributed systems. In *Proc. of 17th International Conference on Ditributed Computing(DISC)*, volume 2848 of *LNCS*, pages 249–263, Oct 2003.
16. A. Mostéfaoui, S. Rajsbaum, and M. Raynal. The synchronous condition-based consensus hierarchy. In *Proc. of 18th International Conference on Distributed Computing(DISC)*, volume 3274 of *LNCS*, pages 1–15. Springer-Verlag, Oct 2004.
17. A. Mostéfaoui, S. Rajsbaum, and M. Raynal. The combined power of conditions and failure detectors to solve asynchronous set agreement. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, 2005. (to appear).
18. A. Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition-based protocols for set agreement problems. In *Proc. of 17th International Conference on Ditributed Computing (DISC)*, volume 2508 of *LNCS*, pages 48–62. Springer-Verlag, 2002.
19. A. Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing*, 17(1):1–20, 2004.
20. Y. Zibin. Condition-based consensus in synchronous systems. In *Proc. of 17th International Conference on Ditributed Computing(DISC)*, volume 2848 of *LNCS*, pages 239–248, Oct 2003.

Time-Bounded Task-PIOAs: A Framework for Analyzing Security Protocols[*]

Ran Canetti[1,3], Ling Cheung[2,3], Dilsun Kaynar[3], Moses Liskov[4],
Nancy Lynch[3], Olivier Pereira[5], and Roberto Segala[6]

[1] IBM T.J. Watson Center and Massachusetts Institute of Technology
[2] Radboud University of Nijmegen
[3] Massachusetts Institute of Technology
lynch@csail.mit.edu
[4] The College of William and Mary
[5] Université Catholique de Louvain
[6] Università di Verona

Abstract. We present the *Time-Bounded Task-PIOA* modeling framework, an extension of the Probabilistic I/O Automata (PIOA) framework that is intended to support modeling and verification of security protocols. Time-Bounded Task-PIOAs directly model probabilistic and nondeterministic behavior, partial-information adversarial scheduling, and time-bounded computation. Together, these features are adequate to support modeling of key aspects of security protocols, including secrecy requirements and limitations on the knowledge and computational power of adversarial parties. They also support security protocol verification, using methods that are compatible with informal approaches used in the computational cryptography research community. We illustrate the use of our framework by outlining a proof of functional correctness and security properties for a well-known Oblivious Transfer protocol.

1 Introduction

Interacting abstract state machine modeling frameworks such as I/O Automata, and proof techniques based on invariant assertions, levels of abstraction, and composition, have long been used successfully for proving correctness of distributed algorithms. Security protocols are special cases of distributed algorithms—ones that use cryptographic primitives such as encryption and trap-door functions, and guarantee properties such as secrecy and authentication. Thus, one would expect the same kinds of models and methods to be useful for analyzing security protocols. However, making this work requires additions to the traditional frameworks, including mechanisms for modeling secrecy requirements, and for describing limitations on knowledge and computational power of adversarial parties.

[*] Canetti is supported by NSF CyberTrust Grant #430450; Cheung by DFG/NWO bilateral cooperation project 600.050.011.01 Validation of Stochastic Systems (VOSS); Kaynar and Lynch by DARPA/AFOSR MURI Award #F49620-02-1-0325, MURI AFOSR Award #SA2796PO 1-0000243658, NSF Awards #CCR-0326277 and #CCR-0121277, and USAF, AFRL Award #FA9550-04-1-0121; Pereira by the Belgian National Fund for Scientific Research (FNRS); and Segala by MURST project Constraint-based Verification of reactive systems (CoVer).

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 238–253, 2006.

In this paper, we describe a modeling framework that extends Segala's Probabilistic I/O Automata (PIOA) framework [Seg95, SL95] and supports description of security-related features. Our extension, which we call the *Time-Bounded Task-PIOA* framework, directly models probabilistic and nondeterministic behavior, partial-information adversarial scheduling, and time-bounded computation. We define an *approximate implementation relation* for Time-Bounded Task-PIOAs, $\leq_{neg,pt}$, which captures the notion of *computational indistinguishability*—the idea that a polynomial-time-bounded observer cannot, with non-negligible probability, distinguish the behavior of one automaton from that of another. We prove that $\leq_{neg,pt}$ is transitive and compositional, and we define a type of probabilistic simulation relation that can be used to prove $\leq_{neg,pt}$. We believe that these features are adequate to support formal modeling and verification of typical security protocols, using methods that are compatible with the informal approaches used in the computational cryptography research community.

We illustrate the use of our framework by outlining a proof of functional correctness and security properties for the Oblivious Transfer (OT) protocol of [EGL85, GMW87]. Together, these properties are expressed by a statement, formulated using $\leq_{neg,pt}$, that every possible behavior of the OT protocol can also be realized by an abstract system representing the required functionality. The protocol model consists of the protocol parties, plus an adversary that acts as a message delivery system and hence has access to dynamic information such as ciphertexts. The abstract system includes an *ideal functionality*, i.e., a trusted third party whose security is assumed, together with a simulator.

Between the protocol and abstract system, we define intermediate systems at different levels of abstraction, and prove that each consecutive pair of levels satisfies $\leq_{neg,pt}$. This decomposes the security proof into several stages, each of which addresses some particular aspect of the protocol. In particular, computational reasoning is isolated to a single stage in the proof, where a system using hard-core bits of trap-door functions is shown to implement a system using random bits. For this interesting step, we reformulate the notion of hard-core bits using $\leq_{neg,pt}$, and prove that our reformulation is equivalent to a standard definition in the literature. The proof of $\leq_{neg,pt}$ for this stage is essentially a reduction to the security of hard-core bits, but the reduction is reformulated in terms of $\leq_{neg,pt}$ and composition results for Time-Bounded Task-PIOAs. Other stages are proved using probabilistic simulation relations.

Background and Prior Work: Traditionally, security protocols have been analyzed using one of two approaches: *formal* or *computational*. In the formal approach, cryptographic operations are modeled purely symbolically, and security of a protocol is expressed in terms of absolute guarantees when the protocol is run against a Dolev-Yao adversary [DY83], which is incapable of breaking the cryptographic primitives. This approach lends itself to rigorous proofs using familiar methods; however, it neglects important computational issues that could render protocols invalid in practice. In contrast, in the computational approach, cryptographic operations are modeled as algorithms operating on bit strings, and security is expressed in terms of probabilistic guarantees when protocols are run against resource-bounded adversaries. This approach treats computational issues realistically, but it does not easily support rigorous proofs. For example, resource-bounded protocol components are often modeled as Interactive Turing Machines (ITMs) [GMR89, Can01]. But rigorous proofs in terms of ITMs are infeasible, because they represent components at too fine a level of detail.

A recent trend in security verification is to combine formal and computational analysis in one framework [LMMS98, PW00, BPW04, BCT04, RMST04, Bla05], by defining computational restrictions for abstract machines. This work provides formal syntax for specifying probabilistic polynomial-time (PPT) processes, and formulates computational security requirements in terms of semantic properties of these processes. Our work follows the same general approach, though with its own unique set of modeling choices.

Our starting point was the *Probabilistic I/O Automata (PIOA)* modeling framework [Seg95, SL95], which is based on abstract machines that allow both probabilistic and nondeterministic choices. In order to resolve nondeterministic choices (a prerequisite for stating probabilistic properties), PIOAs are combined with perfect-information schedulers, which can use full knowledge about the past execution in selecting the next action. This scheduling mechanism is too powerful for analyzing security protocols; e.g., a scheduler's choice of the next action may depend on "secret" information hidden in the states of honest protocol participants, and thus may reveal information about secrets to dishonest participants. Therefore, we augmented the PIOA framework to obtain the *Task-PIOA* framework [CCK$^+$06a, CCK$^+$06b], in which nondeterministic choices are resolved by oblivious *task schedules*, which schedule sets of actions (*tasks*) instead of individual actions. Task-PIOAs support familiar methods of abstraction and composition. They include an implementation relation, $\leq_0$, between automata, based on trace distributions, and probabilistic simulation relations that can be used to prove $\leq_0$.

In this paper, we define *Time-Bounded Task-PIOAs* by imposing time bounds on Task-PIOAs, expressed in terms of bit string encodings of automata constituents. This allows us to define polynomial-time Task-PIOAs and our approximate implementation relation $\leq_{neg,pt}$. We adapt the probabilistic simulation relations of [CCK$^+$06a, CCK$^+$06b] so that they can be used to prove $\leq_{neg,pt}$. Our analysis of Oblivious Transfer follows the style of *Universally Composable (UC) Security* [Can01] and *universal reactive simulatability* [PW01].

In [LMMS98, MMS03, RMST04], a process-algebraic syntax is used to specify protocols, and security properties are specified using an asymptotic observational equivalence on process terms. Nondeterministic choices are resolved by several types of probabilistic schedulers, e.g., special Markov chains or probability distributions on the set of actions. Various restrictions, such as environment- and history-independence, are imposed on these schedulers to enable them to support computational security arguments. In [PW00, PW01, BPW04], protocols are specified as interrupt-driven state machines that interact via a system of ports and buffers, and security is specified in terms of *reactive simulatability*, which expresses computational indistinguishability between the environment's views of the protocol and the abstract functional specification. A distributed protocol, in which machines activate each other by generating explicit "clock" signals, is used for scheduling among the (purely probabilistic) machines. This mechanism is similar to the ones typically used for ITMs [GMR89, Can01].

Our work differs from that of these two schools in our choice of underlying machine model and scheduling mechanism. Also, we follow a different modeling and proof methodology, based on the one typically used for distributed algorithms: we use nondeterminism extensively as a means of abstraction, and organize our proofs using invariants, levels of abstraction, and composition.

In other related work, security analysis is sometimes carried out, informally, in terms of a sequence of *games*, which are similar to our levels of abstraction [Sho04, BR04, Bla05, Hal05].

Overview: Sections 2 and 3 review the PIOA and Task-PIOA frameworks, respectively. Section 4 defines Time-Bounded Task-PIOAs, and the approximate implementation relation $\leq_{neg,pt}$. Section 5 presents our definition of hard-core predicates for trapdoor permutations, in terms of $\leq_{neg,pt}$. Section 6 explains how we model cryptographic protocols and their requirements, illustrating this method with the OT protocol. Section 7 outlines our proofs for OT. Conclusions follow in Section 8. Complete details appear in [CCK$^+$06c].

2 PIOAs

In this section, we summarize basic definitions and results for PIOAs; full definitions, results, and proofs appear in [CCK$^+$06a, CCK$^+$06b].

We let $\mathsf{R}^{\geq 0}$ denote the set of nonnegative reals. We assume that the reader is comfortable with basic notions of probability, such as σ-fields and discrete probability measures. For a discrete probability measure μ on a set X, $supp(\mu)$ denotes the support of μ, that is, the set of elements $x \in X$ such that $\mu(x) \neq 0$. Given set X and element $x \in X$, the *Dirac* measure $\delta(x)$ is the discrete probability measure on X that assigns probability 1 to x.

A *Probabilistic I/O Automaton (PIOA)* $\mathcal{P}$ is a tuple $(Q, \bar{q}, I, O, H, D)$, where: (i) Q is a countable set of *states*, with *start state* $\bar{q} \in Q$; (ii) I, O and H are countable, pairwise disjoint sets of actions, referred to as *input, output and internal actions*, respectively; and (iii) $D \subseteq Q \times (I \cup O \cup H) \times Disc(Q)$ is a *transition relation*, where $Disc(Q)$ is the set of discrete probability measures on Q. An action a is *enabled* in a state q if $(q, a, \mu) \in D$ for some μ. The set $A := I \cup O \cup H$ is called the *action alphabet* of $\mathcal{P}$. If $I = \emptyset$, then $\mathcal{P}$ is *closed.* The set of *external* actions of $\mathcal{P}$ is $I \cup O$ and the set of *locally controlled* actions is $O \cup H$. We assume that $\mathcal{P}$ satisfies:

- **Input enabling:** For every $q \in Q$ and $a \in I$, a is enabled in q.
- **Transition determinism:** For every $q \in Q$ and $a \in A$, there is at most one $\mu \in Disc(Q)$ such that $(q, a, \mu) \in D$.

An *execution fragment* of $\mathcal{P}$ is a finite or infinite sequence $\alpha = q_0\, a_1\, q_1\, a_2 \ldots$ of alternating states and actions, such that (i) if α is finite, it ends with a state; and (ii) for every non-final i, there is a transition $(q_i, a_{i+1}, \mu) \in D$ with $q_{i+1} \in supp(\mu)$. We write $fstate(\alpha)$ for q_0, and if α is finite, we write $lstate(\alpha)$ for its last state. $\mathsf{Frags}(\mathcal{P})$ (resp., $\mathsf{Frags}^*(\mathcal{P})$) denotes the set of all (resp., all finite) execution fragments of $\mathcal{P}$. An *execution* of $\mathcal{P}$ is an execution fragment α with $fstate(\alpha) = \bar{q}$. $\mathsf{Execs}(\mathcal{P})$ (resp., $\mathsf{Execs}^*(\mathcal{P})$) denotes the set of all (resp., all finite) executions of $\mathcal{P}$. The *trace* of an execution fragment α, written $\mathsf{trace}(\alpha)$, is the restriction of α to the external actions of $\mathcal{P}$.

A PIOA, together with a *scheduler* that chooses the sequence of actions to be performed, gives rise to a unique *probabilistic execution*, and thereby, to a unique probability distribution on traces. Traditionally, the schedulers used for PIOAs have been perfect-information schedulers, which can use full knowledge about the past execution in selecting the next action.

Two PIOAs $\mathcal{P}_i = (Q_i, \bar{q}_i, I_i, O_i, H_i, D_i)$, $i \in \{1, 2\}$, are said to be *compatible* if $A_i \cap H_j = O_i \cap O_j = \emptyset$ whenever $i \neq j$. In that case, we define their *composition* $\mathcal{P}_1 \| \mathcal{P}_2$ to be the PIOA $(Q_1 \times Q_2,\ (\bar{q}_1, \bar{q}_2),\ (I_1 \cup I_2) \setminus (O_1 \cup O_2),\ O_1 \cup O_2,\ H_1 \cup H_2, D)$, where D is the set of triples $((q_1, q_2), a, \mu_1 \times \mu_2)$ such that (i) a is enabled in some q_i; and (ii) for every i, if $a \in A_i$ then $(q_i, a, \mu_i) \in D_i$, and otherwise $\mu_i = \delta(q_i)$. A *hiding* operator is also available for PIOAs: given $\mathcal{P} = (Q, \bar{q}, I, O, H, D)$ and $S \subseteq O$, $hide(\mathcal{P}, S)$ is defined to be $(Q, \bar{q}, I, O', H', D)$, where $O' = O \setminus S$ and $H' = H \cup S$.

3 Task-PIOAs

The perfect-information schedulers that have been used to resolve nondeterministic choices in PIOAs are too powerful for computational analysis of security protocols; e.g., a scheduler's choice of the next action may depend on "secret" information hidden in the states of honest protocol participants, and thus may reveal information about secrets to dishonest participants. To avoid this problem, we resolve nondeterminism using a more restrictive, oblivious task mechanism. Again, full definitions, results, and proofs appear in [CCK$^+$06a, CCK$^+$06b].

Basic Definitions: A *Task-PIOA* $\mathcal{T}$ is a pair $(\mathcal{P}, R)$, where $\mathcal{P} = (Q, \bar{q}, I, O, H, D)$ is a PIOA (satisfying transition determinism and input enabling), and R is an equivalence relation on the locally-controlled actions $O \cup H$. The equivalence classes of R are called *tasks*. A task T is an *output task* if $T \subseteq O$, and similarly for *internal tasks*. Unless otherwise stated, we will use terminology inherited from the PIOA setting. We require the following axiom for task-PIOAs:

- **Action determinism:** For every state $q \in Q$ and every task $T \in R$, there is at most one action $a \in T$ that is enabled in q.

In case some $a \in T$ is enabled in q, we say that T is *enabled* in q. If T is enabled in every state from a set S, then T is *enabled* in S.

A *task schedule* for $\mathcal{T} = (\mathcal{P}, R)$ is a finite or infinite sequence $\rho = T_1\, T_2\, \ldots$ of tasks in R. A task schedule is *oblivious*, in the sense that it does not depend on dynamic information generated during execution. Because of the action-determinism assumption for task-PIOAs and the transition-determinism assumption for PIOAs, ρ can be used to generate a unique probabilistic execution, and hence, a unique trace distribution, of $\mathcal{P}$. One can do this by repeatedly scheduling tasks, each of which determines at most one transition of $\mathcal{P}$.

Formally, we define an operation *apply* that "applies" a task schedule to a finite execution fragment, by applying tasks one at a time. To apply a task T

```
Automaton $Src(D, \mu)$:
Signature:
Input:
    none
Output:
    $rand(d), d \in D$
Internal:
    $chooserand$

State:
    $chosenval \in D \cup \{\bot\}$, initially $\bot$

Transitions:
$chooserand$
Pre: $chosenval = \bot$
Eff:
    $chosenval :=$ choose-random$(D, \mu)$

$rand(d)$
Pre: $d = chosenval \neq \bot$
Eff:
    none

Tasks: $\{chooserand\}$, $\{rand(*)\}$
```

Fig. 1. Code for $Src(D, \mu)$

to an execution fragment α: (i) if T is not enabled in $lstate(\alpha)$, then the result is α itself; (ii) otherwise, due to action- and transition-determinism, there is a unique transition from $lstate(\alpha)$ with a label in T, and the result is α extended with that transition. We generalize this construction from a single fragment α to a discrete probability measure η on execution fragments.

Now consider η of the form $\delta(\bar{q})$. For every task schedule ρ, $apply(\delta(\bar{q}), \rho)$, the results of applying ρ to $\bar{q}$, is said to be a *probabilistic execution* of $\mathcal{T}$. This can be viewed as a probabilistic tree generated by running $\mathcal{P}$ from its start state, resolving nondeterministic choices according to ρ. The *trace distribution* induced by ρ, $tdist(\rho)$, is the image measure of $apply(\delta(\bar{q}), \rho)$ under the measurable function trace. A *trace distribution* of $\mathcal{T}$ is $tdist(\rho)$ for any ρ, and we define $tdists(\mathcal{T}) := \{tdist(\rho) \mid \rho \text{ is a task schedule for } \mathcal{T}\}$.

Figure 1 contains a specification of a *random source* task-PIOA $Src(D, \mu)$, which we use in our OT models. *Src* draws an element d from the distribution μ using the action *chooserand* and outputs that element using the action $rand(d)$. It has two tasks: internal task $\{chooserand\}$ and output task $\{rand(d) \mid d \in D\}$. Notice, since all $rand(d)$ actions are grouped into a single task, a task scheduler decides whether or not to output the random element without "knowing" which element has been drawn.

Given compatible task-PIOAs $\mathcal{T}_i = (\mathcal{P}_i, R_i)$, $i \in \{1, 2\}$, we define their *composition* $\mathcal{T}_1 \| \mathcal{T}_2$ to be the task-PIOA $(\mathcal{P}_1 \| \mathcal{P}_2, R_1 \cup R_2)$. Note that $R_1 \cup R_2$ is an equivalence relation because compatibility requires sets of locally controlled actions to be disjoint. It is also easy to check that action determinism is preserved under composition. The hiding operation for task-PIOAs hides all actions in the specified tasks: given task-PIOA $\mathcal{T} = (\mathcal{P}, R)$ and a set $U \subseteq R$ of output tasks, $hide(\mathcal{T}, U)$ is defined to be $(hide(\mathcal{P}, \bigcup U), R)$.

Implementation: An implementation relation expresses the idea that every behavior of one automaton is also a behavior of a second automaton. In that case, it is "safe" to replace the second automaton with the first in a larger system. This notion mades sense only if the two automata interact with the environment via the same interface: thus, two task-PIOAs $\mathcal{T}_1$ and $\mathcal{T}_2$ are *comparable* if $I_1 = I_2$ and $O_1 = O_2$.

If $\mathcal{T}$ and $\mathcal{E}$ are task-PIOAs, then $\mathcal{E}$ is said to be an *environment* for $\mathcal{T}$ if $\mathcal{T}$ and $\mathcal{E}$ are compatible and $\mathcal{T}\|\mathcal{E}$ is closed. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are comparable task-PIOAs, then $\mathcal{T}_1$ *implements* $\mathcal{T}_2$, written $\mathcal{T}_1 \leq_0 \mathcal{T}_2$, if $\mathit{tdists}(\mathcal{T}_1\|\mathcal{E}) \subseteq \mathit{tdists}(\mathcal{T}_2\|\mathcal{E})$ for every environment $\mathcal{E}$ for both $\mathcal{T}_1$ and $\mathcal{T}_2$. Equivalently, given any task schedule ρ_1 for $\mathcal{T}_1\|\mathcal{E}$, there is a task schedule ρ_2 for $\mathcal{T}_2\|\mathcal{E}$ such that $\mathit{tdist}(\rho_1) = \mathit{tdist}(\rho_2)$. Since we require equality of corresponding trace distributions, this is also referred to as *perfect implementation.* Transitivity of $\leq_0$ is trivial and compositionality is proved in [CCK$^+$06a, CCK$^+$06b].

Simulation Relations: In [CCK$^+$06a, CCK$^+$06b], we define a new type of probabilistic simulation relation for task-PIOAs, and prove that such simulation relations are sound for proving $\leq_0$. Here we outline the definition. Let $\mathcal{T}_1 = (\mathcal{P}_1, R_1)$ and $\mathcal{T}_2 = (\mathcal{P}_2, R_2)$ be two comparable closed task-PIOAs. A simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ is a relation R from discrete probability measures on $\mathsf{Frags}^*(\mathcal{P}_1)$ to discrete probability measures on $\mathsf{Frags}^*(\mathcal{P}_2)$ satisfying a number of conditions: (i) If $(\eta_1, \eta_2) \in R$, then η_1 and η_2 induce the same trace distribution; (ii) The Dirac measures $\delta(\bar{q}_1)$ and $\delta(\bar{q}_2)$ are related by R; and (iii) There is a function $\mathit{corr} : ({R_1}^* \times R_1) \rightarrow {R_2}^*$ such that, given $(\eta_1, \eta_2) \in R$, a task schedule ρ for $\mathcal{T}_1$ and a task T of $\mathcal{T}_1$, the measures $\mathit{apply}(\eta_1, T)$ and $\mathit{apply}(\eta_2, \mathit{corr}(\rho, T))$ are related by the expansion of R. (Expansion is a standard operation on relations between discrete measures (cf. [CCK$^+$06b]).) That is, given a task schedule ρ for $\mathcal{T}_1$ that has already been matched in $\mathcal{T}_2$, and a new task T, corr matches T with a finite sequence of tasks of $\mathcal{T}_2$, and the result of scheduling T after η_1 is again related to the result of scheduling the sequence $\mathit{corr}(\rho, T)$ after η_2.

Adversarial Scheduling: The standard scheduling mechanism in the security protocol community is an *adversarial scheduler*—a resource-bounded algorithmic entity that determines the next move adaptively, based on its own view of the computation so far. Our oblivious task schedules do not directly capture the adaptivity of adversarial schedulers. To address this issue, we separate scheduling concerns into two parts: We model the adaptive adversarial scheduler as a system component, e.g., a message delivery service that can eavesdrop on the communications and control the order of message delivery. Such a service has access to partial information about the execution: it sees information that other components communicate to it during execution, but not "secret information" that these components hide. Its choices may be essential to the analysis of the protocol. On the other hand, basic scheduling choices are resolved by a task schedule sequence, chosen nondeterministically in advance. These choices are less important; for example, in the OT protocol, both the transmitter and receiver make random choices, but it is inconsequential which does so first.

4 Time-Bounded Task-PIOAs

A key assumption of computational cryptography is that certain problems cannot be solved with non-negligible probability by resource-bounded entities. In particular, adversaries are assumed to be resource-bounded. To express such bounds formally, we introduce the notion of a *time-bounded task-PIOA*, which assumes bit-string representations of automata constituents and imposes time bounds on Turing machines that decode the representations and compute the next action and next state. Complete details appear in [CCK$^+$06c].

Basic Definitions: We assume a standard bit-string representation for actions and tasks of task-PIOAs. A task-PIOA $\mathcal{T}$ is said to be *b-time-bounded*, where $b \in \mathsf{R}^{\geq 0}$, provided: (i) every state and transition has a bit-string representation, and the length of the representation of every automaton part is at most b; (ii) there is a deterministic Turing machine that decides whether a given representation of a candidate automaton part is indeed such an automaton part, and this machine runs in time at most b; (iii) there is a deterministic Turing machine that, given a state and a task of $\mathcal{T}$, determines the next action (or indicates "no action"), in time at most b; and (iv) there is a probabilistic Turing machine that, given a state and an action of $\mathcal{T}$, determines the next state of $\mathcal{T}$, in time at most b. Furthermore, each of these Turing machines can be described using a bit string of length at most b, according to some standard encoding of Turing machines.

Composing two compatible time-bounded task-PIOAs yields a time-bounded task-PIOA with a bound that is linear in the sum of the original bounds. Similarly, hiding changes the time bound by a linear factor. We say that task schedule ρ is *b-bounded* if $|\rho| \leq b$, that is, ρ is finite and contains at most b tasks.

Task-PIOA Families: Typically, a computational hardness assumption states that, as the size of a problem grows, the success probability of a resource-bounded entity trying to solve the problem diminishes quickly. The size of a problem is expressed in terms of a *security parameter* $k \in \mathsf{N}$, e.g., the key length for an encryption scheme. Accordingly, we define families of task-PIOAs indexed by a security parameter: a *task-PIOA family* $\overline{\mathcal{T}}$ is an indexed set $\{\mathcal{T}_k\}_{k\in\mathsf{N}}$ of task-PIOAs. The notion of time bound is also expressed in terms of the security parameter; namely, given $b : \mathsf{N} \to \mathsf{R}^{\geq 0}$, we say that $\overline{\mathcal{T}}$ is *b-time-bounded* if every $\mathcal{T}_k$ is $b(k)$ time-bounded. Task-PIOA family $\overline{\mathcal{T}}$ is said to be *polynomial-time-bounded* provided that $\overline{\mathcal{T}}$ is p-time-bounded for some polynomial p. Compatibility and parallel composition for task-PIOA families are defined pointwise. Results for composition and hiding carry over easily from those for time-bounded task-PIOAs.

Approximate Implementation: Our notion of approximate implementation allows errors in the emulation and takes into account time bounds of various automata involved. Let $\mathcal{T}$ be a closed task-PIOA with a special output action *accept* and let ρ be a task schedule for $\mathcal{T}$. The *acceptance probability* with respect to $\mathcal{T}$ and ρ is defined to be:

$$Paccept(\mathcal{T}, \rho) := \Pr[\beta \leftarrow tdist(\mathcal{T}, \rho) : \beta \text{ contains } accept],$$

where $\beta \leftarrow tdist(\mathcal{T}, \rho)$ means β is drawn randomly from $tdist(\mathcal{T}, \rho)$.

From now on, we assume that every environment has *accept* as an output. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be comparable task-PIOAs, $\epsilon, b \in \mathsf{R}^{\geq 0}$, and $b_1, b_2 \in \mathsf{N}$. Then we define $\mathcal{T}_1 \leq_{\epsilon,b,b_1,b_2} \mathcal{T}_2$ as follows: Given any b-time-bounded environment $\mathcal{E}$ for both $\mathcal{T}_1$

and $\mathcal{T}_2$, and any b_1-bounded task schedule ρ_1 for $\mathcal{T}_1 \| \mathcal{E}$, there is a b_2-bounded task schedule ρ_2 for $\mathcal{T}_2 \| \mathcal{E}$ such that $|\mathit{Paccept}(\mathcal{T}_1 \| \mathcal{E}, \rho_1) - \mathit{Paccept}(\mathcal{T}_2 \| \mathcal{E}, \rho_2)| \leq \epsilon$. In other words, from the perspective of a b-time-bounded environment, $\mathcal{T}_1$ and $\mathcal{T}_2$ "look almost the same" in the sense that $\mathcal{T}_2$ can use at most b_2 steps to emulate at most b_1 steps of $\mathcal{T}_1$. The relation $\leq_{\epsilon,b,b_1,b_2}$ is transitive and preserved under composition and hiding, with certain adjustments to errors and time bounds.

We extend the relation $\leq_{\epsilon,b,b_1,b_2}$ to task-PIOA families in the obvious way: Let $\overline{\mathcal{T}}_1 = \{(\mathcal{T}_1)_k\}_{k \in \mathsf{N}}$ and $\overline{\mathcal{T}}_2 = \{(\mathcal{T}_2)_k\}_{k \in \mathsf{N}}$ be (pointwise) *comparable* task-PIOA families, $\epsilon, b : \mathsf{N} \rightarrow \mathsf{R}^{\geq 0}$, and $b_1, b_2 : \mathsf{N} \rightarrow \mathsf{N}$. Then we say that $\overline{\mathcal{T}}_1 \leq_{\epsilon,b,b_1,b_2} \overline{\mathcal{T}}_2$ provided that $(\mathcal{T}_1)_k \leq_{\epsilon(k),b(k),b_1(k),b_2(k)} (\mathcal{T}_2)_k$ for every k.

Restricting attention to negligible error and polynomial time bounds, we obtain a generic version of approximate implementation, $\leq_{neg,pt}$; we will used this relation throughout our analysis to express computational security properties. A function $\epsilon : \mathsf{N} \rightarrow \mathsf{R}^{\geq 0}$ is said to be *negligible* if, for every constant $c \in \mathsf{R}^{\geq 0}$, there exists $k_0 \in \mathsf{N}$ such that $\epsilon(k) < \frac{1}{k^c}$ for all $k \geq k_0$. (In other words, ϵ diminishes more quickly than the reciprocal of any polynomial.) We say that $\overline{\mathcal{T}}_1 \leq_{neg,pt} \overline{\mathcal{T}}_2$ if, for all polynomials p and p_1, there is a polynomial p_2 and a negligible function ϵ such that $\overline{\mathcal{T}}_1 \leq_{\epsilon,p,p_1,p_2} \overline{\mathcal{T}}_2$. We show that $\leq_{neg,pt}$ is transitive and preserved under composition; for composition, we need to assume polynomial time bounds for one of the task-PIOA families.

Theorem 1. *Suppose $\overline{\mathcal{T}}_1$, $\overline{\mathcal{T}}_2$ and $\overline{\mathcal{T}}_3$ are three comparable task-PIOA families such that $\overline{\mathcal{T}}_1 \leq_{neg,pt} \overline{\mathcal{T}}_2$ and $\overline{\mathcal{T}}_2 \leq_{neg,pt} \overline{\mathcal{T}}_3$. Then $\overline{\mathcal{T}}_1 \leq_{neg,pt} \overline{\mathcal{T}}_3$.*

Theorem 2. *Suppose $\overline{\mathcal{T}}_1$, $\overline{\mathcal{T}}_2$ are comparable families of task-PIOAs such that $\overline{\mathcal{T}}_1 \leq_{neg,pt} \overline{\mathcal{T}}_2$, and suppose $\overline{\mathcal{T}}_3$ is a polynomial time-bounded task-PIOA family, compatible with both $\overline{\mathcal{T}}_1$ and $\overline{\mathcal{T}}_2$. Then $\overline{\mathcal{T}}_1 \| \overline{\mathcal{T}}_3 \leq_{neg,pt} \overline{\mathcal{T}}_2 \| \overline{\mathcal{T}}_3$.*

Simulation Relations: In order to use simulation relations in a setting with time bounds, we impose an additional assumption on the length of matching task schedules: Given $c \in \mathsf{N}$, a simulation R is said to be *c-bounded* if $|\mathit{corr}(\rho_1, T)| \leq c$ for all ρ_1 and T. We have the following theorem:

Theorem 3. *Let $\overline{\mathcal{T}}_1$ and $\overline{\mathcal{T}}_2$ be comparable task-PIOA families, $c \in \mathsf{N}$. Suppose that for every polynomial p, every k, and every $p(k)$-bounded environment $\mathcal{E}_k$ for $(\overline{\mathcal{T}}_1)_k$ and $(\overline{\mathcal{T}}_2)_k$, there exists a c-bounded simulation R_k from $(\overline{\mathcal{T}}_1)_k \| \mathcal{E}_k$ to $(\overline{\mathcal{T}}_2)_k \| \mathcal{E}_k$. Then $\overline{\mathcal{T}}_1 \leq_{neg,pt} \overline{\mathcal{T}}_2$.*

Proof. In [CCK$^+$06b], soundness of simulation is proved as follows: Given a simulation R between closed task-PIOAs $\mathcal{T}_1$ and $\mathcal{T}_2$, and a task schedule $\rho_1 = T_1, T_2, \ldots$ for $\mathcal{T}_1$, we construct a task schedule ρ_2 for $\mathcal{T}_2$ by concatenating sequences returned by *corr*: $\rho_2 := \mathit{corr}(\lambda, T_1) \ldots \mathit{corr}(T_1 \ldots T_n, T_{n+1}) \ldots$ (λ denotes the empty sequence.) We then prove that $\mathit{tdist}(\rho_1) = \mathit{tdist}(\rho_2)$. Note that, if R is c-bounded, then the length of ρ_2 is at most $c \cdot |\rho_1|$.

Now let polynomials p and p_1 be given as in the definition of $\leq_{neg,pt}$. Let p_2 be $c \cdot p_1$ and ϵ be the constant-0 function. Using the proof outlined above, it is easy to check that p_2 and ϵ satisfy the requirements for $\leq_{neg,pt}$. □

5 Hard-Core Predicates

In this section, we reformulate the standard definition of hard-core predicates using our approximate implementation relation $\leq_{neg,pt}$. This is an important step towards a fully formalized computational analysis of the OT protocol, since the security of OT relies on properties of hard-core predicates.[1] For the rest of the paper, we fix a family $\overline{D} = \{D_k\}_{k\in\mathsf{N}}$ of finite domains and a family $\overline{Tdp} = \{Tdp_k\}_{k\in\mathsf{N}}$ of sets of trap-door permutations such that D_k is the domain of every $f \in Tdp_k$.

In the traditional definition, a function $B : \bigcup_{k\in\mathsf{N}} D_k \to \{0,1\}$ is said to be a hard-core predicate for $\overline{Tdp}$ if, whenever f and z are chosen randomly from Tdp_k and D_k, respectively, the bit $B(f^{-1}(z))$ "appears random" to a probabilistic-polynomial-time observer, even if f and z are given to the observer as inputs. This captures the idea that $f^{-1}(z)$ cannot be computed efficiently from f and z. More precisely, a hard-core predicate is defined by the following slight reformulation of Definition 2.5.1 of [Gol01]:

Definition 1. *A* hard-core predicate *for $\overline{D}$ and $\overline{Tdp}$ is a predicate $B : \bigcup_{k\in\mathsf{N}} D_k \to \{0,1\}$ such that (i) B is polynomial-time computable; and (ii) for every probabilistic polynomial-time predicate $G = \{G_k\}_{k\in\mathsf{N}}$* [2]*, there is a negligible function ϵ such that, for all k,*

$$\left| \begin{array}{l} \Pr[\ f \leftarrow Tdp_k; \\ \quad z \leftarrow D_k; \\ \quad b \leftarrow B(f^{-1}(z)) : \\ \quad G_k(f,z,b) = 1 \quad] \end{array} - \begin{array}{l} \Pr[\ f \leftarrow Tdp_k; \\ \quad z \leftarrow D_k; \\ \quad b \leftarrow \{0,1\} : \\ \quad G_k(f,z,b) = 1\ ] \end{array} \right| \leq \epsilon(k).$$

Note that, when A is a finite set, the notation $x \leftarrow A$ means that x is selected randomly (according to the uniform distribution) from A.

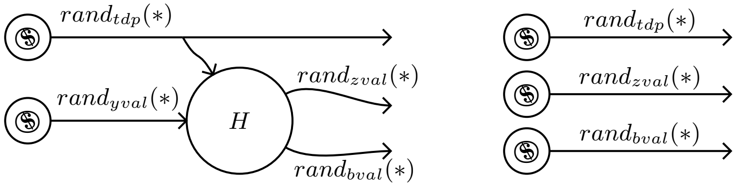
Our new definition uses $\leq_{neg,pt}$ to express the idea that $B(f^{-1}(z))$ "appears random". We define two task-PIOA families, $\overline{SH}$ and $\overline{SHR}$. The former outputs random elements f and z from Tdp_k and D_k, and the bit $B(f^{-1}(z))$. The latter does the same except $B(f^{-1}(z))$ is replaced by a random element from $\{0,1\}$. Then B is said to be a hard-core predicate for $\overline{D}$ and $\overline{Tdp}$ if $\overline{SH} \leq_{neg,pt} \overline{SHR}$.

Definition 2. *B is said to be a* hard-core predicate *for $\overline{D}$ and $\overline{Tdp}$ if $\overline{SH} \leq_{neg,pt} \overline{SHR}$, where $\overline{SH}$ and $\overline{SHR}$ are defined as follows. For each $k \in \mathsf{N}$, let μ_k and μ'_k denote the uniform distributions on Tdp_k and D_k, respectively. Let μ'' be the uniform distribution on $\{0,1\}$.*

$\overline{SH}$ is defined to be $hide(\overline{Src_{tdp}} \| \overline{Src_{yval}} \| \overline{H}, \{rand_{yval}(*)\})$, where (i) $\overline{Src_{tdp}} = \{Src_{tdp}(Tdp_k, \mu_k)\}_{k\in\mathsf{N}}$; (ii) $\overline{Src_{yval}} = \{Src_{yval}(D_k, \mu'_k)\}_{k\in\mathsf{N}}$; and (iii) each H_k obtains f from $Src_{tdp}(Tdp_k, \mu_k)$ and y from $Src_{yval}(D_k, \mu'_k)$, and outputs $z := f(y)$ via action $rand_{zval}$ and $b := B(y)$ via action $rand_{bval}$ (cf. Figure 2). Since f is a permutation, this is equivalent to choosing z randomly and computing y as $f^{-1}(z)$.

[1] In [MMS03], an OT protocol using hard-core bits is analyzed in a process-algebraic setting. However, that work does not include a formalization of hard-core predicates.

[2] This is defined to be a family of predicates that can be evaluated by a family $(M_k)_k$ of probabilistic polynomial-time Turing machines.



SH SHR

Fig. 2. *SH* and *SHR*

$\overline{SHR}$ is defined to be $\overline{Src_{tdp}} \| \overline{Src_{zval}} \| \overline{Src_{bval}}$, where (i) $\overline{Src_{tdp}}$ is as in $\overline{SH}$; (ii) $\overline{Src_{zval}} = \{Src_{zval}(D_k, \mu'_k)\}_{k \in \mathsf{N}}$; and (iii) $\overline{Src_{bval}} = \{Src_{bval}(\{0,1\}, \mu'')_k\}_{k \in \mathsf{N}}$.

These two systems are represented in Fig. 2. There, the automata labeled with $\textcircled{\$}$ represent the random source automata. We claim that these two definitions of hard-core bits are equivalent:

Theorem 4. *B is a hard-core predicate for $\overline{D}$ and $\overline{Tdp}$ according to Definition 1 if and only if B is a hard-core predicate for $\overline{D}$ and $\overline{Tdp}$ according to Definition 2.*

To illustrate how our new definition of hard-core predicates can be exploited in analyzing protocols, we show that a hard-core predicate can be applied twice, and a probabilistic polynomial-time environment still cannot distinguish the outputs from random values. We use this fact in our OT proof, in a situation where the transmitter applies the hard-core predicate to both $f^{-1}(zval(0))$ and $f^{-1}(zval(1))$, where f is the chosen trap-door function.

We show, if B is a hard-core predicate, then no probabilistic polynomial-time environment can distinguish distribution $(f, z(0), z(1), B(f^{-1}(z(0))), B(f^{-1}(z(1))))$ from distribution $(f, z(0), z(1), b(0), b(1))$, where f is a randomly-chosen trap-door permutation, $z(0)$ and $z(1)$ are randomly-chosen elements of D_k, and $b(0)$ and $b(1)$ are randomly-chosen bits. We do this by defining two task-PIOA families, $\overline{SH2}$ and $\overline{SHR2}$, that produce the two distributions, and showing that $\overline{SH2} \leq_{neg,pt} \overline{SHR2}$. Task-PIOA family $\overline{SH2}$ is defined as

$$hide(\overline{Src_{tdp}} \| \overline{Src_{yval0}} \| \overline{Src_{yval1}} \| \overline{H0} \| \overline{H1}, \{rand(*)_{yval0}, rand(*)_{yval1}\}),$$

where $\overline{Src_{tdp}}$ is as in the definition of $\overline{SH}$, $\overline{Src_{yval0}}$ and $\overline{Src_{yval1}}$ are isomorphic to $\overline{Src_{yval}}$ in $\overline{SH}$, and $\overline{H0}$ and $\overline{H1}$ are two instances of $\overline{H}$ (with appropriate renaming of actions). Task-PIOA family $\overline{SHR2}$ is defined as

$$(\overline{Src_{tdp}} \| \overline{Src_{zval0}} \| \overline{Src_{zval1}} \| \overline{Src_{bval0}} \| \overline{Src_{bval1}}),$$

where $\overline{Src_{tdp}}$ is as in $\overline{SH2}$, $\overline{Src_{zval0}}$ and $\overline{Src_{zval1}}$ are isomorphic to $\overline{Src_{zval}}$ in $\overline{SHR}$, and $\overline{Src_{bval0}}$ and $\overline{Src_{bval1}}$ are isomorphic to $\overline{Src_{bval}}$ in $\overline{SHR}$.

Theorem 5. *If B is a hard-core predicate, then $\overline{SH2} \leq_{neg,pt} \overline{SHR2}$.*

Proof. Theorem 4 implies that $\overline{SH} \leq_{neg,pt} \overline{SHR}$. To prove that $\overline{SH2} \leq_{neg,pt} \overline{SHR2}$, we introduce a new task-PIOA family $\overline{Int}$, which is intermediate between $\overline{SH2}$ and $\overline{SHR2}$. $\overline{Int}$ is defined as

$$hide(\overline{Src_{tdp}} \| \overline{Src_{yval0}} \| \overline{H0} \| \overline{Src_{zval1}} \| \overline{Src_{bval1}}, \{rand(*)_{yval0}\}),$$

where $\overline{Src_{tdp}}$ is exactly as in $\overline{SH2}$ and $\overline{SHR2}$; $\overline{Src_{yval0}}$ and $\overline{H0}$ are as in $\overline{SH2}$; and $\overline{Src_{zval1}}$ and $\overline{Src_{bval1}}$ are as in $\overline{SHR2}$. Thus, $\overline{Int}$ generates $bval0$ using the hard-core predicate B, as in $\overline{SH2}$, and generates $bval1$ randomly, as in $\overline{SHR2}$.

To see that $\overline{SH2} \leq_{neg,pt} \overline{Int}$, note that Definition 1 implies that

$$hide(\overline{Src_{tdp}} \| \overline{Src_{yval1}} \| \overline{H1}, \{rand(*)_{yval1}\}) \leq_{neg,pt} \overline{Src_{tdp}} \| \overline{Src_{zval1}} \| \overline{Src_{bval1}},$$

because these two systems are simple renamings of $\overline{SH}$ and $\overline{SHR}$. Now let $\overline{I}$ be the task-PIOA family $hide(\overline{Src_{yval0}} \| \overline{H0}, \{rand(*)_{yval0}\}$. It is easy to see, from the code for the two components of $\overline{I}$, that $\overline{I}$ is polynomial-time-bounded. Then by Theorem 2,

$$hide(\overline{Src_{tdp}} \| \overline{Src_{yval1}} \| \overline{H1}, \{rand(*)_{yval1}\}) \| \overline{I} \leq_{neg,pt} \overline{Src_{tdp}} \| \overline{Src_{zval1}} \| \overline{Src_{bval1}} \| \overline{I}.$$

Since the left-hand side of this relation is $\overline{SH2}$ and the right-hand side is $\overline{Int}$, this implies $\overline{SH2} \leq_{neg,pt} \overline{Int}$.

Similarly, $\overline{Int} \leq_{neg,pt} \overline{SHR2}$. Since $\overline{SH2} \leq_{neg,pt} \overline{Int}$ and $\overline{Int} \leq_{neg,pt} \overline{SHR2}$, transitivity of $\leq_{neg,pt}$ (Theorem 1) implies that $\overline{SH2} \leq_{neg,pt} \overline{SHR2}$. □

6 Computational Security

Here, we explain how we define the security of cryptographic protocols, illustrating with our OT example. Our method follows the general outline in [PW01, Can01], which in turn follows standard practice in the computational cryptography community. We first define a task-PIOA specifying the functionality the protocol is supposed to realize, then specify task-PIOAs describing the protocol, and finally, define what it means for a protocol to securely realize its specification.

The *functionality* task-PIOA represents a "trusted party" that receives protocol inputs and returns protocol outputs, at various locations. See the full version of [Can01] for many examples of classical cryptographic functionalities. The functionality we use for Oblivious Transfer is a task-PIOA *Funct* that behaves as follows. First, it waits for two bits (x_0, x_1) representing the inputs for the protocol's first party (the *transmitter*), and one bit i representing the input for the protocol's second party (the *receiver*). Then, it outputs the bit x_i to the receiver, and nothing to the transmitter.

Since the definitions of protocols are typically parameterized by a security parameter k, we define a protocol as a task-PIOA family $\overline{\pi} = \{\pi_k\}_{k \in \mathsf{N}}$, where π_k is the composition of the task-PIOAs specifying the roles of the different protocol participants for parameter k. Given families $\overline{D}$ and $\overline{Tdp}$, and a parameter k, the OT protocol we consider executes as follows. First the transmitter *Trans* selects a trapdoor permutation f from Tdp_k, together with its inverse f^{-1}, and sends f to the receiver *Rec*. Then, using its input bit i and two randomly selected elements (y_0, y_1) of D_k, *Rec* computes the pair $(z_0, z_1) = (f^{1-i}(y_0), f^i(y_1))$, and sends it in a second protocol message to *Trans*. Finally, using its input bits

(x_0, x_1), *Trans* computes the pair $(b_0, b_1) = (x_0 \oplus B(f^{-1}(z_0)), x_1 \oplus B(f^{-1}(z_1)))$ and sends it to *Rec*, who can now recover x_i as $B(y_i) \oplus b_i$.

In order to define the security of a protocol with respect to a functionality, we must specify a particular class of adversaries. Depending on the context, adversaries may have different capabilities: they may have passive or active access to the network, may be able to corrupt parties (either statically or dynamically), may assume partial or full control of the parties, etc. Various scenarios are discussed in [Can01]. We specify a particular class of adversaries by defining appropriate restrictions on the signature and transition relation of adversary task-PIOAs. By composing an adversary task-PIOA Adv_k with a protocol task-PIOA π_k, we obtain what we call the *real system*.

For the OT protocol, we consider polynomial-time-bounded families of adversaries. The adversaries have passive access to protocol messages: they receive and deliver messages (possibly delaying, reordering, or losing messages), but do not compose messages of their own. They may corrupt parties only statically (at the start of execution). They are "honest-but-curious": they obtain access to all internal information of the corrupted parties, but the parties continue to follow the protocol definition. In this paper, we discuss only one corruption case, in which only the receiver is corrupted. In this case, the adversary gains access to the input and output of *Rec* (that is, i and x_i), and to its internal choices (that is, y_0 and y_1). However, as we said above, *Rec* continues to follow the protocol definition, so we model it as a component distinct from the adversary.

In order to prove that the protocol realizes the functionality, we show that, for every adversary family $\overline{Adv}$ of the considered class, there is another task-PIOA family $\overline{Sim}$, called a *simulator*, that can mimic the behavior of $\overline{Adv}$ by interacting with the functionality. This proves that the protocol does not reveal to the adversary any information it does not need to reveal—that is, anything not revealed by the functionality itself.

The quality of emulation is evaluated from the viewpoint of one last task-PIOA, the *environment*. Thus, security of the protocol says that no environment can efficiently decide if it is interacting with the real system, or with the composition of the functionality and the simulator (we call this composition the *ideal system*). This indistinguishability condition is formalized as follows:

Theorem 6. *Let $\overline{RS}$ be a real-system family, in which the family $\overline{Adv}$ of adversary automata is polynomial-time-bounded. Then there exists an ideal-system family $\overline{IS}$, in which the family $\overline{Sim}$ is polynomial-time-bounded, and such that $\overline{RS} \leq_{neg,pt} \overline{IS}$.*

In the statement of Theorem 6, quantification over environments is encapsulated within the definition of $\leq_{neg,pt}$: $\overline{RS} \leq_{neg,pt} \overline{IS}$ says, for every polynomial time-bounded environment family $\overline{Env}$ and every polynomial-bounded task schedule for $\overline{RS}||\overline{Env}$, there is a polynomial-bounded task schedule for $\overline{IS}||\overline{Env}$ such that the acceptance probabilities in these two systems differ by a negligible amount.

7 Levels-of-Abstraction Proofs

In order to prove that Theorem 6 holds for the OT protocol and the adversaries of Section 6, we show the existence of an ideal system family $\overline{IS}$ with $\overline{RS} \leq_{neg,pt} \overline{IS}$.

To that end, we build a structured simulator family $\overline{SSim}$ from any adversary family $\overline{Adv}$: for every index k, $SSim_k$ is the composition of Adv_k with an abstract version of π_k based on a task-PIOA $TR(D_k, Tdp_k)$. TR works as follows. First, it selects and sends a random element f of Tdp_k, as *Trans* would. Then, when Adv_k has delivered f, TR emulates *Rec*: it chooses a random pair (y_0, y_1) of elements of D_k, and sends the second protocol message, computed as the pair $(f^{1-i}(y_0), f^i(y_1))$. Next, TR generates the third protocol message using the bit x_i obtained from the *Funct*, which TR obtains because x_i is an output at the corrupted receiver. Namely, TR computes b_i as $B(y_i) \oplus x_i$ and b_{1-i} as a random bit. Observe that, if $\overline{Adv}$ is polynomial-time-bounded, then so is $\overline{SSim}$. Also, if we define $\overline{SIS}$ by $SIS_k = Funct \| SSim_k$, then $\overline{SIS}$ is an ideal system family.

In showing that $\overline{RS} \leq_{neg,pt} \overline{SIS}$, we define two intermediate families of systems, $\overline{Int1}$ and $\overline{Int2}$, and decompose the proof into showing three subgoals: $\overline{RS} \leq_{neg,pt} \overline{Int1}$, $\overline{Int1} \leq_{neg,pt} \overline{Int2}$, and $\overline{Int2} \leq_{neg,pt} \overline{SIS}$. All arguments involving computational indistinguishability and other cryptographic issues are isolated to the middle level, namely, $\overline{Int1} \leq_{neg,pt} \overline{Int2}$.

The $Int1_k$ system is almost the same as SIS_k, except that TR is replaced by $TR1$, which differs from TR as follows. First, it has an extra input $in(x)_{Trans}$, obtaining the protocol input (x_0, x_1) intended for the transmitter. Second, it computes the third protocol message differently: the bit b_i is computed as in TR, but the bit b_{1-i} is computed using the hard-core predicate B, as $B(f^{-1}(z_{1-i})) \oplus x_{1-i}$. The $Int2_k$ system is defined to be the same as SIS_k except that it includes a random source automaton Src_{cval1} that chooses a random bit $cval1$, and TR is replaced by $TR2$, which is essentially the same as $TR1$ except that b_{1-i} is computed as $cval1 \oplus x_{1-i}$.

Our proofs that $\overline{RS} \leq_{neg,pt} \overline{Int1}$ and $\overline{Int2} \leq_{neg,pt} \overline{SIS}$ use simulation relations. To prove that $\overline{RS} \leq_{neg,pt} \overline{Int1}$, we show that, for every polynomial p, for every k, and for every $p(k)$-bounded environment Env_k for RS_k and $Int1_k$, there is a c-bounded simulation relation R_k from $RS_k \| Env_k$ to $Int1_k \| Env_k$, where c is a constant. Using Theorem 3, we obtain $\overline{RS} \leq_{neg,pt} \overline{Int1}$. Our proof of $\overline{Int2} \leq_{neg,pt} \overline{SIS}$ is similar, but with an interesting aspect. Namely, we show that computing the bit b_{1-i} as a random bit is equivalent to computing it as the XOR of a random bit and the input bit x_i.

Our proof that $\overline{Int1} \leq_{neg,pt} \overline{Int2}$ uses a computational argument, based on our definition of a hard-core predicate. The only difference between $\overline{Int1}$ and $\overline{Int2}$ is that a use of $B(f^{-1}(z_{1-i}))$ in $\overline{Int1}$ is replaced by a random bit in $\overline{Int2}$. This is precisely the difference between the $\overline{SR}$ and $\overline{SHR}$ systems discussed in Section 5. In order to exploit the fact that $\overline{SR} \leq_{neg,pt} \overline{SHR}$, we build an interface task-PIOA family $\overline{Ifc}$ which represents the common parts of $\overline{Int1}$ and $\overline{Int2}$. Then, we prove: (i) $\overline{Int1} \leq_{neg,pt} \overline{SH} \| \overline{Ifc} \| \overline{Adv}$ and $\overline{SHR} \| \overline{Ifc} \| \overline{Adv} \leq_{neg,pt} \overline{Int2}$ by exhibiting simple, constant-bounded simulation relations between these systems, and (ii) $\overline{SH} \| \overline{Ifc} \| \overline{Adv} \leq_{neg,pt} \overline{SHR} \| \overline{Ifc} \| \overline{Adv}$ by using our definition, Definition 2, of hard-core predicates, the fact that both $\overline{Ifc}$ and $\overline{Adv}$ are polynomial-time-bounded, and the composition property of $\leq_{neg,pt}$ (Theorem 2). Finally, using transitivity of $\leq_{neg,pt}$ (Theorem 1), we have $\overline{RS} \leq_{neg,pt} \overline{SIS}$, as needed.

8 Conclusions

We have introduced *time-bounded task-PIOAs* and *task-PIOA families*, building on the task-PIOA framework of [CCK$^+$06a, CCK$^+$06b]. We have adapted basic machinery, such as composition and hiding operations, and implementation and simulation relations, to time-bounded task-PIOAs. We have demonstrated the use of this framework in formulating and proving computational security properties for an Oblivious Transfer protocol [EGL85, GMW87]. Our proofs are decomposed into several stages, each showing an implementation relationship, $\leq_{neg,pt}$, between two systems. Most of these implementations are proved using simulation relations to match corresponding events and probabilities in the two systems. Others are proved using computational arguments involving reduction to the security of cryptographic primitives. Traditional reduction arguments for cryptographic primitives are reformulated in terms of $\leq_{neg,pt}$.

Our framework supports separation of scheduling concerns into two pieces: high-level scheduling, which is controlled by an algorithmic entity (e.g., the adversary component), and low-level scheduling, which is resolved nondeterministically by a task schedule. This separation allows inessential ordering of events to remain as nondeterministic choices in our system models, which increases the generality and reduces the complexity of the models and proofs.

We believe that the model and techniques presented here provide a suitable basis for analyzing a wide range of cryptographic protocols, including those that depend inherently on computational assumptions and achieve only computational security. Future plans include applying our methods to analyze more complex protocols, including protocols that use other cryptographic primitives and protocols that work against more powerful adversaries. We plan to establish general security protocol composition theorems in the style of [Can01, PW01] within our framework. We would like to formulate general patterns of adversarial behavior, in the style of [PW01], within our framework, and use this formulation to obtain general results about the security of the resulting systems.

References

[BCT04] G. Barthe, J. Cerderquist, and S. Tarento. A machine-checked formalization of the generic model and the random oracle model. In *Automated Reasoning: Second International Joint Conference (IJCAR)*, 2004.

[Bla05] B. Blanchet. A computationally sound mechanized prover for security protocols. Cryptology ePrint Archive, Report 2005/401, 2005. `http://eprint.iacr.org/`.

[BPW04] M. Backes, B. Pfitzmann, and M. Waidner. Secure asynchronous reactive systems. Cryptology ePrint Archive, Report 2004/082, 2004. `http://eprint.iacr.org/`.

[BR04] M. Bellare and P. Rogaway. The game-playing technique and its application to triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. `http://eprint.iacr.org/`.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Conference on Foundations of Computer Science (FOCS)*, 2001. Full version available at http://eprint.iacr.org/2000/067.

[CCK⁺06a] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-structured probabilistic I/O automata. In *Proceedings of the 8th International Workshop on Discrete Event Systems (WODES)*, Ann Arbor, Michigan, July 2006.

[CCK⁺06b] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-structured probabilistic I/O automata. Technical Report MIT-CSAIL-TR-2006-XXX, CSAIL, MIT, Cambridge, MA, 2006.

[CCK⁺06c] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Using task-structured probabilistic I/O automata to analyze an oblivious transfer protocol. Technical Report MIT-CSAIL-TR-2006-047, CSAIL, MIT, Cambridge, MA, June 2006.

[DY83] D. Dolev and A.C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *CACM*, 28(6):637–647, 1985.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[Gol01] O. Goldreich. *Foundations of Cryptography*, volume I Basic Tools. Cambridge Universirty Press, 2001.

[Hal05] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. `http://eprint.iacr.org/`.

[LMMS98] P.D. Lincoln, J.C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS-5)*, pages 112–121, 1998.

[MMS03] P. Mateus, J.C. Mitchell, and A. Scedrov. Composition of cryptographic protocols in a probabilistic polynomial-time calculus. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR)*, volume 2761 of *LNCS*, pages 327–349, 2003.

[PW00] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *7th ACM Conference on Computer and Communications Security*, pages 245–254, 2000.

[PW01] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *IEEE Symposium on Security and Privacy*, pages 184–200, 2001.

[RMST04] A. Ramanathan, J.C. Mitchell, A. Scedrov, and V. Teague. Probabilistic bisimulation and equivalence for security analysis of network protocols. In *Proceedings of Foundations of Sotware Science and Computation Structires (FOSSACS)*, volume 2987 of *LNCS*, pages 468–483, 2004.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, May 1995. Also, MIT/LCS/TR-676.

[Sho04] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `http://eprint.iacr.org/`.

[SL95] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, August 1995.

On Consistency of Encrypted Files

Alina Oprea[1] and Michael K. Reiter[2]

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
alina@cs.cmu.edu
[2] Electrical & Computer Engineering Department, Computer Science Department, and CyLab,
Carnegie Mellon University, Pittsburgh, PA, USA
reiter@cmu.edu

Abstract. In this paper we address the problem of consistency for cryptographic file systems. A cryptographic file system protects the users' data from the file server, which is possibly untrusted and might exhibit Byzantine behavior, by encrypting the data before sending it to the server. The consistency of the encrypted file objects that implement a cryptographic file system relies on the consistency of the two components used to implement them: the file storage protocol and the key distribution protocol.

We first define two generic classes of consistency conditions that extend and generalize existing consistency conditions. We then formally define consistency for encrypted file objects in a generic way: for any consistency conditions for the key and file objects belonging to one of the two classes of consistency conditions considered, we define a corresponding consistency condition for encrypted file objects. We finally provide, in our main result, necessary and sufficient conditions for the consistency of the key distribution and file storage protocols under which the encrypted storage is consistent. Our framework allows the composition of existing key distribution and file storage protocols to build consistent encrypted file objects and simplifies complex proofs for showing the consistency of encrypted storage.

1 Introduction

Consistency for a file system that supports data sharing specifies the semantics of multiple users accessing files simultaneously. Intuitively, the ideal model of consistency would respect the real-time ordering of file operations, i.e., a read would return the last written version of that file. This intuition is captured in the model of consistency known as linearizability [16], though in practice, such ideal consistency models can have high performance penalties. It is well known that there is a tradeoff between performance and consistency. As such, numerous consistency conditions weaker than linearizability, and that can be implemented more efficiently in various contexts, have been explored. Sequential consistency [19], causal consistency [4], PRAM consistency [22] and more recently, fork consistency [24], are several examples.

In this paper we address the problem of consistency for encrypted file objects used to implement a cryptographic file system. A cryptographic file system protects the users' data from the file server, which is possibly untrusted and might exhibit Byzantine behavior, by encrypting the data before sending it to the server. When a file can be shared, the decryption key must be made available to authorized readers, and similarly authorized writers of the file must be able to retrieve the encryption key or else create one of

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 254–268, 2006.

their own. In this sense, a key is an object that, like a file, is read and/or written in the course of implementing the abstraction of an encrypted file.

Thus, an *encrypted file object* is implemented through two main components: the *key object* that stores the encryption key, and the *file object* that stores (encrypted) file contents. We emphasize that the key and file objects may be implemented via completely different protocols and infrastructures. Our concern is the impact of the consistency of each on the encrypted file object that they are used to implement. The consistency of the file object is obviously essential to the consistency of the encrypted data retrieved. At the same time, the encryption key is used to protect the confidentiality of the data and to control access to the file. So, if consistency of the key object is violated, this could interfere with authorized users decrypting the data retrieved from the file object, or it might result in a stale key being used indefinitely, enabling revoked users to continue accessing the data. We thus argue that the consistency of both the key and file objects affects the consistency of the encrypted file object. Knowing the consistency of a key distribution and a file access protocol, our goal is to find necessary and sufficient conditions that ensure the consistency of the encrypted file that the key object and the file object are utilized to implement.

The problem that we consider is related to the *locality* problem. A consistency condition is *local* if a history of operations on multiple objects satisfies the consistency condition if the restriction of the history to each object does so. However, locality is a very restrictive condition and, to our knowledge, only very powerful consistency conditions, such as linearizability, satisfy it. In contrast, the combined history of key and file operations can satisfy weaker conditions and still yield a consistent encrypted file. We give a generic definition of consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for an encrypted file object, starting from any consistency conditions C_1 and C_2 for the key and file objects that belong to one of the two classes of generic conditions we define. Intuitively, our consistency definition requires that the key and file operations seen by each client can be arranged such that they preserve C_1-consistency for the key object and C_2-consistency for the file object, and, in addition, the latest key versions are used to encrypt file contents. The requirement that the most recent key versions are used for encrypting new file contents is important for security, as usually the encryption key for a file is changed when some users are revoked access to the file. We allow the decryption of a file content read with a previous key version (not necessarily the most recent seen by the client), as this would not affect security. Thus, a system implementing our definition guarantees both *consistency* for file contents and *security* in the sense that revoked users are restricted access to the encrypted file object.

Rather than investigate consistency for a single implementation of an encrypted file, we consider a collection of implementations that are all *key-monotonic*. Intuitively, in a key-monotonic implementation, there exists a consistent ordering of file operations such that the written file contents are encrypted with monotonically increasing key versions. We formally define this property that depends on the consistency of the key and file objects. We prove in our main result (Theorem 1) that ensuring that an implementation is key-monotonic is a necessary and sufficient condition for obtaining consistency for the encrypted file object, given several restrictions on the consistency of the key and file objects. Our main result provides a framework to analyze the consistency of a given implementation of an encrypted file object: if the key object and file object satisfy consistency conditions C_1 and C_2, respectively, and the given implementation is key-monotonic with respect to C_1 and C_2, then the encrypted file object is $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistent.

In this context, we summarize our contributions as follows:

- We define two generic classes of consistency conditions. The class of *orderable consistency conditions* includes and generalizes well-known conditions such as linearizability, causal consistency and PRAM consistency. The class of *forking consistency conditions* is particularly tailored to systems with untrusted shared storage and extends fork consistency [24] to other new, unexplored consistency conditions.
- We define consistency for encrypted files: for any consistency conditions C_1 and C_2 of the key and file objects that belong to these two classes, we define a corresponding consistency condition $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for encrypted files. To our knowledge, our paper is the first to rigorously formalize consistency conditions for encrypted files.
- Our main result provides necessary and sufficient conditions that enable an encrypted file to satisfy our definition of consistency. Given a key object that satisfies a consistency property C_1, and a file object with consistency C_2 from one of the classes we define, our main theorem states that it is enough to ensure the key-monotonicity property in order to obtain consistency for the encrypted file object. This result is subject to certain restrictions on the consistency conditions C_1 and C_2.

In addition, in the full version of this paper [26], we give an example implementation of a consistent encrypted file from a sequentially consistent key object and a fork consistent file object. The proof of consistency of the implementation follows immediately from our main theorem. This demonstrates that complex proofs for showing consistency of encrypted files are simplified using our framework.

The rest of the paper is organized as follows: we survey related work in Section 2, and give the basic definitions, notation and system model in Section 3. We define the two classes of consistency conditions in Section 4 and give the definition of consistency for encrypted files in Section 5. Our main result, a necessary and sufficient condition for constructing consistent encrypted files, is presented in Section 6.

2 Related Work

SUNDR [21] is the first file system that provides consistency guarantees (fork consistency [24]) in a model with a Byzantine storage server and benign clients. In SUNDR, the storage server keeps a signed *version structure* for each user of the file system. The version structures are modified at each read or write operation and are totally ordered as long as the server respects the protocol. A misbehaving server might conceal users' operations from each other and break the total order among version structures, with the effect that users get divided into groups that will never see the same system state again. SUNDR only provides data integrity, but not data confidentiality. In contrast, we are interested in providing consistency guarantees in encrypted storage systems in which keys may change, and so we must consider distribution of the encryption keys, as well.

For obtaining consistency conditions stronger than fork consistency (e.g., linearizability) in the face of Byzantine servers, one solution is to distribute the file server across n replicas, and use this replication to mask the behavior of faulty servers. Modern examples include BFT [9], SINTRA [8] and PASIS [1]. An example of a distributed encrypting file system that provides strong consistency guarantees for both file data and meta-data is FARSITE [2]. File meta-data in FARSITE (that also includes the encryption key for the file) is collectively managed by all users that have access to the file, using a Byzantine fault tolerant protocol. There exist distributed implementations of storage servers that guarantee weaker semantics than linearizability. Lakshmanan et al. [18] provide causal consistent implementations for a distributed storage system. While they discuss encrypted data, they do not treat the impact of encryption on the consistency of the system.

Several network encrypting file systems, such as SiRiUS [14] and Plutus [17], develop interesting ideas for access control and user revocation, but they both leave the key distribution problem to be handled by clients through out-of-band communication. Since the key distribution protocol is not specified, neither of the systems makes any claims about consistency. Other file systems address key management: e.g., SFS [23] separates key management from file system security and gives multiple schemes for key management; Cepheus [12] relies on a trusted server for key distribution; and SNAD [25] uses separate key and file objects to secure network attached storage. However, none of these systems addresses consistency. We refer the reader to the survey by Riedel et al. [27] for an extensive comparison of the security properties of various encrypting file systems.

Another area related to our work is that of consistency semantics. Different applications have different consistency and performance requirements. For this reason, many different consistency conditions for shared objects have been defined and implemented, ranging from strong conditions such as linearizability [16], sequential consistency [19], and timed consistency [28] to loose consistency guarantees such as causal consistency [4], PRAM [22], coherence [15,13], processor consistency [15,13,3], weak consistency [10], entry consistency [7], and release consistency [20]. A generic, continuous consistency model for wide-area replication that generalizes the notion of serializability [6] for transactions on replicated objects has been introduced by Yu and Vahdat [30]. We construct two generic classes of consistency conditions that include and extend some of the existing conditions for shared objects.

Different properties of generic consistency conditions for shared objects have been analyzed in previous work, such as *locality* [29] and *composability* [11]. Locality analyzes for which consistency conditions a history of operations is consistent, given that the restriction of the history to each individual object satisfies the same consistency property. Composability refers to the combination of two consistency conditions for a history into a stronger, more restrictive condition. In contrast, we are interested in the consistency of the combined history of key and file operations, given that the individual operations on keys and files satisfy possibly different consistency properties. We also define generic models of consistency for histories of operations on encrypted file objects that consist of operations on key and file objects.

Generic consistency conditions for shared objects have been restricted previously only to conditions that satisfy the *eventual propagation* property [11]. Intuitively, eventual propagation guarantees that all the write operations are eventually seen by all processes. This assumption is no longer true when the storage server is potentially faulty and we relax this requirement for the class of forking consistency conditions we define.

3 Preliminaries

3.1 Basic Definitions and System Model

Most of our definitions are taken from Herlihy and Wing [16]. We consider a system to be a set of processes $p_1, \ldots, p_n$ that invoke operations on a collection of shared objects. Each operation o consists of an *invocation* $\mathsf{inv}(o)$ and a *response* $\mathsf{res}(o)$. We only consider read and write operations on single objects. A write of value v to object X is denoted $X.\mathsf{write}(v)$ and a read of value v from object X is denoted $v \leftarrow X.\mathsf{read}()$.

A *history* H is a sequence of invocations and responses of read and write operations on the shared objects. We consider only *well-formed* histories, in which every invocation of an operation in a history has a matching response. We say that an operation belongs to a history H if its invocation and response are in H. A *sequential history* is a history in which every invocation of an operation is immediately followed by the corresponding response. A *serialization* S of a history H is a sequential history containing all the operations of H and no others. An important concept for consistency is the notion of a *legal sequential history*, defined as a sequential history in which read operations return the values of the most recent write operations.

Notation. For a history H and a process p_i, we denote by $H|p_i$ the sequential history of operations in H done by p_i. For a history H and objects $X_1, \ldots, X_n$, we denote by $H|(X_1, \ldots, X_n)$ the restriction of H to operations on objects $X_1, \ldots, X_n$. We denote by $H|w$ all the write operations in history H and by $H|p_i + w$ the operations in H done by process p_i and all the write operations done by all processes in history H.

3.2 Eventual Propagation

A history satisfies *eventual propagation* [11] if, intuitively, all the write operations done by the processes in the system are eventually seen by all processes. However, the order in which processes see the operations might be different. More formally, eventual propagation is defined below:

Definition 1 (Eventual Propagation and Serialization Set). *A history H satisfies* eventual propagation *if for every process p_i, there exists a legal serialization S_p of $H|p_i + w$. The set of legal serializations for all processes $S = \{S_p\}_i$ is called a* serialization set *[11] for history H.*

If a history H admits a legal serialization S, then a serialization set $\{S_p\}_i$ with $S_p = S|p_i + w$ can be constructed and it follows immediately that H satisfies eventual propagation.

3.3 Ordering Relations on Operations

There are several natural partial ordering relations that can be defined on the operations in a history H. Here we describe three of them: the *local* (or *process order*), the *causal order* and the *real-time order*.

Definition 2 (Ordering Relations). *Two operations o_1 and o_2 in a history H are ordered by local order (denoted $o_1 \longrightarrow o_2$) if there exists a process p_i that executes o_1 before o_2.*

The causal order extends the local order relation. We say that an operation o_1 directly precedes *o_2 in history H if either $o_1 \longrightarrow o_2$, or o_1 is a write operation, o_2 is a read operation and o_2 reads the result written by o_1. The causal order (denoted $\xrightarrow{*}$) is the transitive closure of the direct precedence relation.*

Two operations o_1 and o_2 in a history H are ordered by the real-time order (denoted $o_1 <_H o_2$) if $\mathsf{res}(o_1)$ *precedes* $\mathsf{inv}(o_2)$ *in history H.*

A serialization S of a history H induces a *total order* relation on the operations of H, denoted $\longrightarrow$. Two operations o_1 and o_2 in H are ordered by $\longrightarrow$ if o_1 precedes o_2 in the serialization S.

On the other hand, a serialization set $S = \{S_p\}_i$ of a history H induces a *partial order* relation on the operations of H, denoted $\longrightarrow$. For two operations o_1 and o_2 in H, $o_1 \longrightarrow o_2$ if and only if (i) o_1 and o_2 both appear in at least one serialization S_p and (ii) o_1 precedes o_2 in all the serializations S_p in which both o_1 and o_2 appear. If o_1 precedes o_2 in one serialization, but o_2 precedes o_1 in a different serialization, then the operations are concurrent with respect to $\longrightarrow$.

4 Classes of Consistency Conditions

The goal of this paper is to analyze the consistency of encrypted file systems generically and give necessary and sufficient conditions for its realization. A *consistency condition* is a set of histories. We say that a history H is C-*consistent* if $H \in \mathsf{C}$ (this is also denoted by $\mathsf{C}(H)$). Given consistency conditions C and C', C is *stronger* than C' if $\mathsf{C} \subseteq \mathsf{C}'$.

As the space of consistency conditions is very large, we need to restrict ourselves to certain particular and meaningful classes for our analysis. One of the challenges we faced was to define interesting classes of consistency conditions that include some of the well known conditions defined in previous work (i.e., linearizability, causal consistency, PRAM consistency). Generic consistency conditions have been analyzed previously (e.g., [11]), but the class of consistency conditions considered was restricted to conditions with histories that satisfy eventual propagation. Given our system model with a potentially faulty shared storage, we cannot impose this restriction on all the consistency conditions we consider in this work.

We define two classes of consistency conditions, differentiated mainly by the eventual propagation property. The histories that belong to conditions from the first class satisfy eventual propagation and are *orderable*, a property we define below. The histories that belong to conditions from the second class do not necessarily satisfy eventual propagation, but the legal serializations of all processes can be arranged into a *forking tree*. This class includes fork consistency [24], and extends that definition to other new, unexplored consistency conditions. The two classes do not cover all the existing consistency conditions.

4.1 Orderable Conditions

Intuitively, a consistency condition C is orderable if it contains only histories for which there exists a serialization set that respects a certain partial order relation. Consider the example of *causal consistency* [4] defined as follows: a history H is causally consistent if and only if there exists a serialization set S of H that respects the causal order relation, i.e., $\xrightarrow{*} \subseteq \longrightarrow$. We generalize the requirement that the serialization set respects the causal order to more general partial order relations. A subtle point in this definition is the specification of the partial order relation. First, it is clear that the partial order needs to be different for every condition C. But, analyzing carefully the definition of the causal order relation, we notice that it depends on the history H. We can thus view the causal order relation as a family of relations, one for each possible history H. Generalizing, in the definition of an orderable consistency condition C, we require the existence of a family of partial order relations, indexed by the set of all possible histories, denoted by $\{ \xrightarrow{\mathsf{C},} \}_H$. Additionally, we require that each relation $\xrightarrow{\mathsf{C},}$ respects the local order of operations in H.

Definition 3 (Orderable Consistency Conditions). *A consistency condition C is orderable if there exists a family of partial order relations $\{ \xrightarrow{\mathsf{C},} \}_H$, indexed by the set of all possible histories, with $\longrightarrow \subseteq \xrightarrow{\mathsf{C},}$ for all histories H such that:*

$$H \in \mathsf{C} \Leftrightarrow \textit{ there exists a serialization set } S \textit{ of } H \textit{ with } \xrightarrow{\mathsf{C},} \subseteq \longrightarrow .$$

Given a history H from class C, a serialization set S of H that respects the order relation $\xrightarrow{\mathsf{C},}$ is called a C-consistent serialization set *of H.*

We define class $\mathcal{C}_O$ to be the set of all orderable consistency conditions. A subclass of interest is formed by those consistency conditions in $\mathcal{C}_O$ that contain only histories for which there exists a legal serialization of their operations. We denote $\mathcal{C}_O^+$ this subclass of $\mathcal{C}_O$. For a consistency condition C from class $\mathcal{C}_O^+$, a serialization S of a history H that respects the order relation $\xrightarrow{\mathsf{C},}$, i.e., $\xrightarrow{\mathsf{C},} \subseteq \longrightarrow$, is called a C-*consistent serialization* of H.

Linearizability [16] and sequential consistency [19] belong to $\mathcal{C}_O^+$ (with the corresponding ordering relations $<_H$ and $\longrightarrow$, respectively), and PRAM [22] and causal consistency [4] to $\mathcal{C}_O \setminus \mathcal{C}_O^+$ (with the corresponding ordering relations $\longrightarrow$ and $\xrightarrow{*}$, respectively).

4.2 Forking Conditions

To model encrypted file systems over untrusted storage, we need to consider consistency conditions that might not satisfy the eventual propagation property. In a model with potentially faulty storage, it might be the case that a process views only a subset of the writes of the other processes, besides the operations it performs. For this purpose, we need to extend the notion of serialization set.

Definition 4 (Extended and Forking Serialization Sets). *An extended serialization set of a history H is a set $S = \{S_p \}_i$ with S_p a legal serialization of a subset of* *operations from H, that includes (at least) all the operations done by process p_i. A forking serialization set of a history H is an extended serialization set $S = \{S_p \}_i$ such that for all $i, j, (i \neq j)$, any $o \in S_p \cap S_p$, and any $o' \in S_p$:*

$$o' \longrightarrow o \Rightarrow (o' \in S_p \wedge o' \longrightarrow o).$$

A forking serialization set is an extended serialization set with the property that its serializations can be arranged into a "forking tree". Intuitively, arranging the serializations in a tree means that any two serializations might have a common prefix of identical operations, but once they diverge, they do not contain any of the same operations. Thus, the operations that belong to a subset of serializations must be ordered the same in all those serializations. A forking consistency condition includes only histories for which a forking serialization set can be constructed. Moreover, each serialization S_p in the forking tree is a C-consistent serialization of the operations seen by p_i, for C a consistency condition from $\mathcal{C}_{\mathcal{O}}^{+}$.

Definition 5 (Forking Consistency Conditions). *A consistency condition* $\mathsf{FORK\text{-}C}$ *is* forking *if:*

1. C *is a consistency condition from* $\mathcal{C}_{\mathcal{O}}^{+}$*;*
2. $H \in \mathsf{FORK\text{-}C}$ *if and only if there exists a forking serialization set* $S = \{S_p \}_i$ *for history* H *with the property that each* S_p *is* C*-consistent.*

We define class $\mathcal{C}_{\mathcal{F}}$ to be the set of all forking consistency conditions $\mathsf{FORK\text{-}C}$. It is immediate that for consistency conditions C, C_1 and C_2 in $\mathcal{C}_{\mathcal{O}}^{+}$, (i) C is stronger than $\mathsf{FORK\text{-}C}$, and (ii) if C_1 is stronger than C_2, then $\mathsf{FORK\text{-}C}_1$ is stronger than $\mathsf{FORK\text{-}C}_2$. $\mathcal{C}_{\mathcal{F}}$ extends the notion of fork consistency defined by Mazieres and Shasha [24].

5 Definition of Consistency for Encrypted Files

We can construct an encrypted file object using two components, the file object and the key object whose values are used to encrypt file contents. File and key objects might be implemented via different protocols and infrastructures. For the purpose of this paper, we consider each file to be associated with a distinct encryption key. We could easily extend this model to accommodate different granularity levels for the encryption keys (e.g., a key for a group of files).

Users perform operations on an encrypted file object that involve operations on both the file and the key objects. For example, a read of an encrypted file might require a read of the encryption key first, then a read of the file and finally a decryption of the file with the key read. We refer to the operations exported by the storage interface (i.e., operations on encrypted file objects) to its users as "high-level" operations and the operations on the file and key objects as "low-level" operations.

We model a cryptographic file system as a collection of encrypted files. Different cryptographic file systems export different interfaces of high-level operations to their users. We can define consistency for encrypted file objects offering a wide range of high-level operation interfaces, as long as the high-level operations consist of low-level write and read operations on key and file objects. We do assume that a process that creates an encryption key writes this to the relevant key object before writing any files encrypted with that key.

The encryption key for a file is changed most probably when some users are revoked access to the file, and thus, for security reasons, we require that *clients use the most recent key they have seen to write new file contents*. However, it is possible to use older versions of the encryption key to decrypt a file read. For example, in a *lazy revocation* model [12,17,5], the re-encryption of a file is not performed immediately when a user is revoked access to the file and the encryption key for that file is changed, but it is delayed until the next write to that file. Thus, in the lazy revocation model older versions of the key might be used to decrypt files, but new file contents are encrypted with the most recent key. In our model, we can accommodate both the lazy revocation method and the *active revocation* method in which a file is immediately re-encrypted with the most recent encryption key at the moment of revocation.

For completeness, here we give an example of a high-level operation interface for an encrypted file object ENCF, which is used in the example implementation given in the full version of this paper [26] :

1. Create a file, denoted as $\mathsf{ENCF.create\_file}(f)$. This operation generates a new encryption key k for the file, writes k to the key object and writes the file content f encrypted with key k to the file object.
2. Encrypt and write a file, denoted as $\mathsf{ENCF.write\_encfile}(f)$. This operation writes an encryption of file contents f to the file object, using the most recent encryption key that the client read.
3. Read and decrypt a file, denoted as $f \leftarrow \mathsf{ENCF.read\_encfile}()$. This operation reads an encrypted file from the file object and then decrypts it to f.
4. Write an encryption key, denoted as $\mathsf{ENCF.write\_key}(k)$. This operation changes the encryption key for the file to a new value k. Optionally, it re-encrypts the file contents with the newly generated encryption key if active revocation is used.

Consider a fixed implementation of high-level operations from low-level read and write operations. Each execution of a history H of high-level operations naturally induces a history H_l of low-level operations by replacing each completed high-level operation with the corresponding sequence of invocations and responses of the low-level operations. In the following, we define consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for encrypted file objects, for any consistency properties C_1 and C_2 of the key distribution and file access protocols that belong to classes $\mathcal{C}_{\mathcal{O}}$ or $\mathcal{C}_{\mathcal{F}}$.

Definition 6. *(Consistency of Encrypted File Objects) Let H be a history of completed high-level operations on an encrypted file object* ENCF *and* C_1 *and* C_2 *two consistency properties from* $\mathcal{C}_{\mathcal{O}}$. *Let H_l be the corresponding history of low-level operations on key object* KEY *and file object* FILE *induced by an execution of high-level operations. We say that H is* $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$**-consistent** *if there exists a serialization set $S = \{S_{p_i}\}_i$ of H_l such that:*

1. *S is* enc*-legal, i.e.: For every file write operation $o =$* $\mathsf{FILE.write}(c)$*, there is an operation* $\mathsf{KEY.write}(k)$ *such that: c was generated through encryption with key*

k, $\mathsf{KEY.write}(k) \longrightarrow o$ and there is no $\mathsf{KEY.write}(k')$ with $\mathsf{KEY.write}(k) \longrightarrow \mathsf{KEY.write}(k') \longrightarrow o$ for all i;

2. *$S|\mathsf{KEY} = \{S_p\ |\mathsf{KEY}\}_i$ is a C_1-consistent serialization set of $H_l|\mathsf{KEY}$;*
3. *$S|\mathsf{FILE} = \{S_p\ |\mathsf{FILE}\}_i$ is a C_2-consistent serialization set of $H_l|\mathsf{FILE}$;*
4. *S respects the local ordering of each process.*

Intuitively, our definition requires that there is an arrangement (i.e., serialization set) of key and file operations such that the most recent key write operation before each file write operation seen by each client is the write of the key used to encrypt that file. In addition, the serialization set should respect the desired consistency of the key distribution and file access protocols.

If both C_1 and C_2 belong to $\mathcal{C}_{\mathcal{O}}^{+}$, then the definition should be changed to require the existence of a serialization S of H_l instead of a serialization set. Similarly, if C_2 belongs to $\mathcal{C}_{\mathcal{F}}$, we change the definition to require the existence of an extended serialization set $\{S_p\ \}_i$ of H_l. In the latter case, the serialization S_p for each process might not contain all the key write operations, but it has to include all the key operations that write key values used in subsequent file operations in the same serialization. Conditions (1), (2), (3) and (4) remain unchanged.

The definition can be immediately generalized to multiple encrypted file objects, as was done in the full version of this paper [26].

6 A Necessary and Sufficient Condition for the Consistency of Encrypted File Objects

After defining consistency for encrypted file objects, here we give necessary and sufficient conditions for the realization of the definition. We first outline the dependency among encryption keys and file objects, and then define a property of histories that ensures that file write operations are executed in increasing order of their encryption keys. Histories that satisfy this property are called *key-monotonic*. Our main result, Theorem 1, states that, provided that the key distribution and the file access protocols satisfy some consistency properties C_1 and C_2 with some restrictions, the key-monotonicity property of the history of low-level operations is necessary and sufficient to implement $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ consistency for the encrypted file object.

6.1 Dependency among Values of Key and File Objects

Each write and read low-level operation is associated with a value. The value of a write operation is its input argument and that of a read operation its returned value. For o a file operation with value f done by process p_i, k the value of the key that encrypts f and $w = \mathsf{KEY.write}(k)$ the operation that writes the key value k, we denote the dependency among operations w and o by $\mathsf{R}(w, o)$ and say that file operation *o is associated with* key operation w.

The relation $\mathsf{R}(w, o)$ implies a causal order relation in the history of low-level operations between operations w and o. Since process p_i uses the key value k to encrypt the file content f, then either: (1) in process p_i there is a read operation $r = (k \leftarrow$

$\mathsf{KEY.read}()$) such that $w \xrightarrow{*} r \longrightarrow o$, which implies $w \xrightarrow{*} o$; or (2) w is done by process p_i, in which case $w \longrightarrow o$, which implies $w \xrightarrow{*} o$. In either case, the file operation o is causally dependent on the key operation w that writes the value of the key used in o.

6.2 Key-Monotonic Histories

A history of key and file operations is *key-monotonic* if, intuitively, it admits a consistent serialization for each process in which the file write operations use monotonically increasing versions of keys for encryption of their values. Intuitively, if a client uses a key version to perform a write operation on a file, then all the future write operations on the file object by all the clients will use this or later versions of the key.

We give an example in Figure 1 of a history that is not key-monotonic for sequential consistent keys and linearizable files. Here c_1 and c_2 are file values encrypted with key values k_1 and k_2, respectively. k_1 is ordered before k_2 with respect to the local order. $\mathsf{FILE.write}(c_1)$ is after $\mathsf{FILE.write}(c_2)$ with respect to the real-time ordering, and, thus, in any linearizable serialization of file operations, c_2 is written before c_1.

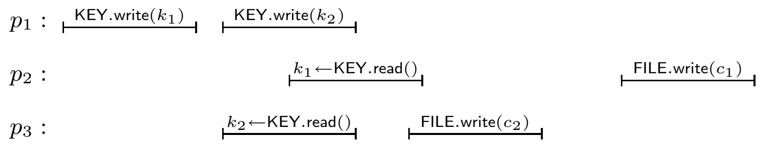


Fig. 1. A history that is not key-monotonic

To define key-monotonicity for a low-level history formally, we would like to find the minimal conditions for its realization, given that the key operations in the history satisfy consistency condition C_1 and the file operations satisfy consistency condition C_2. We assume that the consistency C_1 of the key operations is orderable. Two conditions have to hold in order for a history to be key-monotonic: (1) the key write operations cannot be ordered in opposite order of the file write operations that use them; (2) file write operations that use the same keys are not interleaved with file write operations using a different key.

Definition 7 (Key-Monotonic History). *Consider a history H with two objects, key* KEY *and file* FILE*, such that* $\mathsf{C}_1(H|\mathsf{KEY})$ *and* $\mathsf{C}_2(H|\mathsf{FILE})$*, where* C_1 *is an orderable consistency condition and* C_2 *belongs to either* $\mathcal{C}_\mathcal{O}$ *or* $\mathcal{C}_\mathcal{F}$*. H is a* key-monotonic history *with respect to* C_1 *and* C_2*, denoted* $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}(H)$*, if there exists a* C_2*-consistent serialization (or serialization set or forking serialization set) S of* $H|\mathsf{FILE}$ *such that the following conditions holds:*

- *(*KM_1*) for any two file write operations $f_1 \longrightarrow f_2$ with associated key write operations k_1 and k_2 (i.e.,* $\mathsf{R}(k_1, f_1)$*,* $\mathsf{R}(k_2, f_2)$*), it cannot happen that* $k_2 \xrightarrow{\mathsf{C}_1,\ |\mathsf{KEY}} k_1$.
- *(*KM_2*) for any three file write operations $f_1 \longrightarrow f_2 \longrightarrow f_3$, and key write operation k with* $\mathsf{R}(k, f_1)$ *and* $\mathsf{R}(k, f_3)$*, it follows that* $\mathsf{R}(k, f_2)$.

The example we gave in Figure 1 violates the first condition. If we consider $f_2 = \mathsf{FILE.write}(c_2)$, $f_1 = \mathsf{FILE.write}(c_1)$, then f_2 is ordered before f_1 in any linearizable serialization and k_1 is ordered before k_2 with respect to the local order. But condition (KM_1) states that it is not possible to order key write k_1 before key write k_2.

The first condition (KM_1) is enough to guarantee key-monotonicity for a history H when the key write operations are uniquely ordered by the ordering relation $\xrightarrow{\mathsf{C}_1,\ |\mathsf{KEY}}$. To handle concurrent key writes with respect to $\xrightarrow{\mathsf{C}_1,\ |\mathsf{KEY}}$, we need to enforce the second condition (KM_2) for key-monotonicity. Condition (KM_2) rules out the case in which uses of the values written by two concurrent key writes are interleaved in file operations in a consistent serialization. Consider the example from Figure 2 that is not key-monotonic for sequential consistent key operations and linearizable file operations. In this example c_1 and c_1' are encrypted with key value k_1, and c_2 is encrypted with key value k_2. A linearizable serialization of the file operations is: $\mathsf{FILE.write}(c_1)$; $\mathsf{FILE.write}(c_2)$; $\mathsf{FILE.read}(c_2)$; $\mathsf{FILE.write}(c_1')$, and this is not key-monotonic. k_1 and k_2 are not ordered with respect to the local order, and as such the history does not violate condition (KM_1). However, condition (KM_2) is not satisfied by this history.

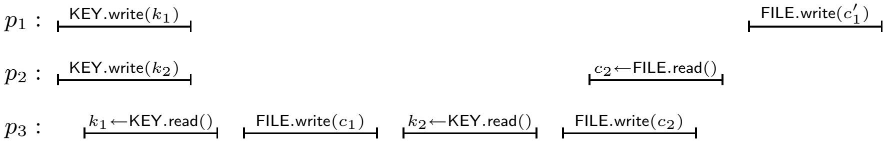


Fig. 2. A history that does not satisfy condition (KM_2)

In cryptographic file system implementations, keys are usually changed only by one process, who might be the owner of the file or a trusted entity. For single-writer objects, it can be proved that sequential consistency, causal consistency and PRAM consistency are equivalent. Since we require the consistency of key objects to be orderable and all orderable conditions are at least PRAM consistent (i.e., admit serialization sets that respect the local order), the weakest consistency condition in the class of orderable conditions for single writer objects is equivalent to sequential consistency. If the key distribution protocol is sequential consistent, the key-monotonicity conditions given in Definition 7 can be simplified. We present below the simplified condition. The proof of equivalence with the conditions from Definition 7 is given in the full version of this paper [26].

Proposition 1. *Let H be a history of operations on the single-writer key object* KEY *and file object* FILE *such that $H|\mathsf{KEY}$ is sequential consistent. H is key-monotonic if and only if the following condition is true:*

*(*SW-KM*) There exists a C_2-consistent serialization S (or serialization set or forking serialization set) of $H|\mathsf{FILE}$ such that for any two file write operations $f_1 \longrightarrow f_2$ with associated key write operations k_1 and k_2 (i.e., $\mathsf{R}(k_1, f_1)$, $\mathsf{R}(k_2, f_2)$), it follows that $k_1 \longrightarrow k_2$ or $k_1 = k_2$.*

6.3 Obtaining Consistency for Encrypted File Objects

We give here the main result of our paper, a necessary and sufficient condition for implementing consistent encrypted file objects, as defined in Section 5. Given a key distribution protocol with orderable consistency C_1 and a file access protocol that satisfies consistency C_2 from classes $\mathcal{C}_{\mathcal{O}}$ or $\mathcal{C}_{\mathcal{F}}$, the theorem states that key-monotonicity is a necessary and sufficient condition to obtain consistency $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$ for the encrypted file object. Some additional restrictions need to be satisfied. The proof of the theorem is in the full version of this paper [26].

Theorem 1. *Consider a fixed implementation of high-level operations from low-level operations. Let H be a history of operations on an encrypted file object* ENCF *and H_l the induced history of low-level operations on key object* KEY *and file object* FILE *by a given execution of high-level operations. Suppose that the following conditions are satisfied: (1)* $\mathsf{C}_1(H_l|\mathsf{KEY})$*; (2)* $\mathsf{C}_2(H_l|\mathsf{FILE})$*; (3)* C_1 *is orderable; (4) if* C_2 *belongs to* $\mathcal{C}_{\mathcal{O}}^{+}$*, then* C_1 *belongs to* $\mathcal{C}_{\mathcal{O}}^{+}$*. Then H is* $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$*-consistent if and only if H_l is a key-monotonic history, i.e.,* $\mathsf{KM}_{\mathsf{C}_1,\mathsf{C}_2}(H)$*.*

Discussion. Our theorem recommends two main conditions to file system developers in order to guarantee $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$-consistency of encrypted file objects. First, the consistency of the key distribution protocol needs to satisfy eventual propagation (as it belongs to class $\mathcal{C}_{\mathcal{O}}$) to apply our theorem. This suggests that using the untrusted storage server for the distribution of the keys, as implemented in several cryptographic file systems, e.g., SNAD [25] and SiRiUS [14], might not meet our consistency definitions. For eventual propagation, the encryption keys have to be distributed either directly by file owners or by using a trusted key server. It is an interesting open problem to analyze the enc-consistency of the history of high-level operations if both the key distribution and file-access protocols have consistency in class $\mathcal{C}_{\mathcal{F}}$. Secondly, the key-monotonicity property requires, intuitively, that file writes are ordered not to conflict with the consistency of the key operations. To implement this condition, one solution is to modify the file access protocol to take into account the version of the encryption key used in a file operation when ordering that file operation. We give an example of modifying the fork consistent protocol given by Mazieres and Shasha [24] in the full version of this paper [26].

Moreover, the framework offered by Theorem 1 simplifies complex proofs for showing consistency of encrypted files. In order to apply Definition 6 directly for such proofs, we need to construct a serialization of the history of low-level operations on both the file and key objects and prove that the file and key operations are correctly interleaved in this serialization and respect the appropriate consistency conditions. By Theorem 1, given a key distribution and file access protocol that is each known to be consistent, verifying the consistency of the encrypted file object is equivalent to verifying key monotonicity. To prove that a history of key and file operations is key monotonic, it is enough to construct a serialization of the file operations and prove that it does not violate the ordering of the key operations. The simple proof of consistency of the example encrypted file object presented in the full version of this paper [26] demonstrates the usability of our framework.

7 Conclusions

We have addressed the problem of consistency in encrypted file systems. An encrypted file system consists of two main components: a file access protocol and a key distribution protocol, which might be implemented via different protocols and infrastructures. We formally define generic consistency in encrypted file systems: for any consistency conditions C_1 and C_2 belonging to the classes we consider, we define a corresponding consistency condition for encrypted file systems, $(\mathsf{C}_1, \mathsf{C}_2)^{\mathsf{enc}}$. The main result of our paper states that if each of the two protocols has some consistency guarantees with some restrictions, then ensuring that the history of low-level operation is key-monotonic is necessary and sufficient to obtain consistency for an encrypted file object. The applicability of our definitions and main result to other classes of consistency conditions is a topic of future work.

Another contribution of this paper is to define two classes of consistency conditions that extend and generalize existing conditions: the first class includes classical consistency conditions such as linearizability and causal consistency, and the second one extends fork consistency. An interesting problem is to find efficient implementations of the new forking consistency conditions from the second class and their relation with existing consistency conditions.

References

1. M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proc. 20th ACM Symposium on Operating Systems (SOSP)*, pp. 59–74, ACM, 2005.
2. A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. 5th Symposium on Operating System Design and Implementation (OSDI)*, Usenix, 2002.
3. M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger, "The power of processor consistency," Technical Report GIT-CC-92/34, Georgia Institute of Technology, 1992.
4. M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: Definitions, implementation and programming," *Distributed Computing*, vol. 1, no. 9, pp. 37–49, 1995.
5. M. Backes, C. Cachin, and A. Oprea, "Secure key-updating for lazy revocation," in *Proc. 11th European Symposium On Research In Computer Security (ESORICS)*, Springer-Verlag, 2006.
6. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
7. B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway distributed shared-memory system," in *Proc. IEEE COMPCON Conference*, pp. 528–537, IEEE, 1993.
8. C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the internet," in *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 167–176, IEEE, 2002.
9. M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd Symposium on Operating System Design and Implementation (OSDI)*, pp. 173–186, Usenix, 1999.
10. M. Dubois, C. Scheurich, and F. Briggs, "Synchronization, coherence and event ordering in multiprocessors," *IEEE Computer*, vol. 21, no. 2, pp. 9–21, 1988.

11. R. Friedman, R. Vitenberg, and G. Chockler, "On the composability of consistency conditions," *Information Processing Letters*, vol. 86, pp. 169–176, 2002.
12. K. Fu, "Group sharing and random access in cryptographic storage file systems," Master's thesis, Massachusetts Institute of Technology, 1999.
13. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 15–26, 1990.
14. E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pp. 131–145, ISOC, 2003.
15. J. Goodman, "Cache consistency and sequential consistency," Technical Report 61, SCI Committee, 1989.
16. M. Herlihy and J. Wing, "Linearizability: A corretness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.
17. M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.
18. S. Lakshmanan, M. Ahamad, and H. Venkateswaran:, "A secure and highly available distributed store for meeting diverse data storage needs," in *Proc. International Conference on Dependable Systems and Networks (DSN)*, pp. 251–260, IEEE, 2001.
19. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, 1979.
20. D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, 1992.
21. J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository," in *Proc. 6th Symposium on Operating System Design and Implementation (OSDI)*, pp. 121–136, Usenix, 2004.
22. R. Lipton and J. Sandberg, "Pram: A scalable shared memory," Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, 1988.
23. D. Mazieres, M. Kaminsky, M. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Proc. 17th ACM Symposium on Operating Systems (SOSP)*, pp. 124–139, ACM, 1999.
24. D. Mazieres and D. Shasha, "Building secure file systems out of Byzantine storage," in *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 108–117, ACM, 2002.
25. E. Miller, D. Long, W. Freeman, and B. Reed, "Strong security for distributed file systems," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–13, 2002.
26. A. Oprea and M. K. Reiter, "On consistency of encrypted files," Technical Report CMU-CS-06-113, Carnegie Mellon University, 2006. Available from `http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-113.pdf`.
27. E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *Proc. First USENIX Conference on File and Storage Technologies (FAST)*, pp. 15–30, 2002.
28. F. J. Torres-Rojas, M. Ahamad, and M. Raynal, "Timed consistency for shared distributed objects," in *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 163–172, ACM, 1999.
29. R. Vitenberg and R. Friedman, "On the locality of consistency conditions," in *Proc. 17th International Symposium on Distributed Computing (DISC))*, pp. 92–105, 2003.
30. H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Transactions on Computer Systems*, vol. 20, no. 3, pp. 239–282, 2002.

Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data

Michael B. Greenwald[1], Sanjeev Khanna[2], Keshav Kunal[2],
Benjamin C. Pierce[2], and Alan Schmitt[3]

[1] Bell Labs, Lucent Technologies
[2] University of Pennsylvania
[3] INRIA

Abstract. Current techniques for reconciling disconnected changes to optimistically replicated data often use version vectors or related mechanisms to track causal histories. This allows the system to tell whether the value at one replica dominates another or whether the two replicas are in conflict. However, current algorithms do not provide entirely satisfactory ways of *repairing* conflicts. The usual approach is to introduce fresh events into the causal history, even in situations where the causally independent values at the two replicas are actually equal. In some scenarios these events may later conflict with each other or with further updates, slowing or even preventing convergence of the whole system.

To address this issue, we enrich the set of possible actions at a replica to include a notion of explicit conflict resolution between *existing* events, where the user at a replica declares that one set of events dominates another, or that a set of events are equivalent. We precisely specify the behavior of this refined replication framework from a user's point of view and show that, if communication is assumed to be "reciprocal" (with pairs of replicas exchanging information about their current states), then this specification can be implemented by an algorithm with the property that the information stored at any replica and the sizes of the messages sent between replicas are bounded by a polynomial function of the number of replicas in the system.

1 Introduction

Some distributed systems maintain consistency by layering on top of a consistent memory abstraction or ordered communication substrate. Others—particularly systems with autonomous nodes that can operate while disconnected—must relax consistency requirements to make progress, depending instead on a notion of *causal history* of events. If a replica learns of different updates to the same object, then the most causally recent update is considered "best" and is preferred over the others. However, if it happens that the replicas held at two sites are modified simultaneously, then neither update will appear in the other's causal history, and neither these sites nor any others that hear from them will be able to prefer one update over the other until the conflict has been *reconciled.*

In standard approaches based on causal histories (e.g. [1,2]), this reconciliation is itself an *event*— a new update that causally supersedes all of the conflicting

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 269–283, 2006.

ones. Unfortunately, this reconciliation event can create new conflicts. Until it propagates through the whole system, any update created on another replica before it hears of the resolution will be causally unrelated to the reconciliation event and will thus conflict with it. Indeed, as has been noted before [1,3], in some systems, the very same conflict might be resolved, independently, by inserting new reconciliation events at different sites, thus raising new conflicts even though the reconciled values may be identical. Most existing systems have found this potential behavior acceptable in practice—conflicts are infrequent or communication frequent enough to ensure that reconciliation events usually propagate throughout the system quickly. However, in some settings (described in detail below), conflicts due to reconciliation events can delay convergence or force users to manually reconcile the same conflict multiple times.

To improve the convergence behavior of such systems, we propose adding a new kind of *agreement event* that labels a set of updates as equivalent, together with a mechanism for declaring that one existing event dominates another. Our goals are to reduce the number of user interventions needed to bring conflicting updates into agreement and to speed global convergence after conflict resolution.

Beyond Causal Histories. Standard causal histories are an attractive way of prioritizing events in a distributed system, partly because they capture a natural relationship between updates and partly because their causal relationships can be represented very efficiently. In particular, it is well known that causal histories can be efficiently summarized using *vector clocks* [1]. Each replica R^α maintains a monotonically increasing counter n^α that is incremented at least once per update event on R^α. Each R^α also maintains a vector (the vector clock), indexed by replica identifiers β, that indicates the latest update of R^β that R^α has heard about (all previous updates of R^β are also in the causal history of R^α). If each update is associated with the local vector clock at the time of its creation, then we can determine the causal relationship between two events: if every entry in one vector clock c_1 is less than or equal to the corresponding entry in another vector clock c_2, then the update v_1, corresponding to c_1, is in the causal history of the update v_2, corresponding to c_2, and v_2 may safely overwrite v_1.

To record the resolution of a conflict using vector clocks, the local vector clock must be changed to reflect the fact that all the conflicting updates are now in the causal past. This can be achieved by first setting the local vector clock to the pointwise maximum of all the vector clocks associated with the conflicting updates and then incrementing the local counter [1].

Unfortunately, this technique can give rise to situations where the system cannot stabilize without further manual intervention—or indeed, in pathological cases, where it can never stabilize. In particular, if, at any point in time, two distinct sites resolve a conflict, even in an identical way, the system will consider the two identical resolutions to be in conflict. Consider the example in Figure 1. From an initial state where all replicas are holding the same value (ϵ), replicas R^a, R^b, and R^c all independently set their value to x at (local) time 1. Although all replicas "agree" in the sense that they are holding the same value, the system will only stabilize if every replica communicates its state (perhaps indirectly) to

	Replica R^a		Replica R^b		Replica R^c	
Event	Local time	value (vc)	Local time	value (vc)	Local time	value (vc)
	0	ϵ (0,0,0)	0	ϵ (0,0,0)	0	ϵ (0,0,0)
Local updates	1	x (1,0,0)	1	x (0,1,0)	1	x (0,0,1)
$R^a \rightarrow R^c$ and $R^b \rightarrow R^c$	1	x (1,0,0)	1	x (0,1,0)	2	x (1,1,2)
$R^a \rightarrow R^b$	1	x (1,0,0)	2	x (1,2,0)	2	x (1,1,2)

Fig. 1. A case in which vector clocks never converge, although all replicas hold the correct value

a single site, that site creates a new update event (with a vector clock that is greater than the pointwise maximum of all 3), and this new event gets communicated back to all the other sites before anything else happens. In Figure 1, the replicas do successfully transfer their state to R^c, which creates an event that could stabilize the system. Unfortunately, R^a also sends its state to R^b. In response to this badly timed message, R^b creates an event that resolves the conflict between R^a and R^b but conflicts with the agreement event generated at R^c. Neither R^c nor R^b's state now dominates the other's, and the system cannot converge until the new conflict between R^c and R^b is repaired.

A natural idea for improving matters is to allow a reconciling site to introduce an *agreement event* that somehow "merges" two causally unrelated updates instead of dominating them. Then if R^c declares that the update events at replicas R^a, R^b, and R^c are all equivalent, and later R^b declares that the events at replicas R^a and R^b are equivalent, the two reconciliations will not conflict.

Agreement events raise issues, however, that cannot be modeled naturally by causal histories. It may appear that agreements that may be helpful in the example above might be implemented by simply having the reconciling site *not* increment its local timestamp after taking the pointwise max of its vector clock with that of the other conflicting replicas; then two reconciliations at different hosts would not conflict. (In the example above, R^c would set its clock to $(1, 1, 1)$ and R^b would later set its to $(1, 1, 0)$—i.e., the reconciled state from R^c would dominate the "partially reconciled" state from R^b.)

However, this scheme is still not satisfactory: if any new updates happen before the reconciliation event(s) propagate completely through the system, spurious conflicts will still be created. Figure 2 shows what can happen. The three replicas, R^a, R^b, and R^c, again begin by all taking on the value x. Later, R^a sends a message to R^b, which reconciles the conflict between their (identical) values by merging R^a's vector clock with its own, yielding (1,1,0). Later, R^b sends a message to R^c, which similarly recognizes that their conflicting values are equal and updates its local clock to (1,1,1). If, at this point, R^c were to send its state to R^a and R^b before anything else happened, all would be well. However, suppose instead that R^a locally updates its value to y. This update clearly supersedes the first update of x on R^a; also, since the value of x on R^b has been reconciled with the old x on R^a, the new update of y at R^a should also supersede the x

	Replica R^a		Replica R^b		Replica R^c	
Event	Local time	value (vc)	Lcl time	value (vc)	Lcl time	value (vc)
Initial state	0	ϵ (0,0,0)	0	ϵ (0,0,0)	0	ϵ (0,0,0)
Local updates	1	x (1,0,0)	1	x (0,1,0)	1	x (0,0,1)
$R^a \rightarrow R^b$	1	x (1,0,0)	1	x (1,1,0)	1	x (0,0,1)
$R^b \rightarrow R^c$	1	x (1,0,0)	1	x (1,1,0)	1	x (1,1,1)
R^a updated	2	y (2,0,0)	1	x (1,1,0)	1	x (1,1,1)

Fig. 2. A case in which vector clocks "forget" a resolution event

on R^b, and similarly on R^c. However, at this point the system is totally stalled, although it is clear (to an omniscient observer) that all replicas should converge to y. No sequence of messages will ever reconcile R^a with either R^b or R^c. (Note that the value on R^c is not in the causal history of y, even if *both* the sender and receiver update their local clocks after communication.)

In a similar vein, vector clocks and standard causal histories provide no way of reconciling a conflict by simply declaring that one of the conflicting events is *better* than the others. For example, suppose replicas R^a and R^b are independently updated with conflicting values and each communicates its value to some large set of other nodes before anybody notices the conflict. If the user performing the reconciliation decides that R^a's value is actually preferable to R^b's, they would like to be able to declare this to the system so that, with no further intervention, every host that hears about both updates will choose R^a's value. Moreover, if, in the meantime, some host that heard about R^a's update has made yet a further update, this new value should also automatically be preferred over R^b's.

These shortcomings are not an artifact of a vector clock representation; the system stalls because causal histories do not remember equivalences between events. If R^c declares that the values at R^a, R^b and R^c are equivalent, and R^a simultaneously decides that the value y is preferable to its current value x, then we want the system to prefer one causally unrelated value to another. There is no way to put the value at R^c into the causal history of R^a. (We will see later that attempting to simply add equivalence edges can causes cycles in the causal history graph. Those cycles, in turn, can give rise to paradoxical behavior.)

Such scenarios become more likely as the frequency of updates (and hence conflicts and reconciliations) increases, relative to the speed with which information propagates between nodes. Thus, in systems where conflicts are rare, or where nodes are tightly coupled and communicate frequently, vector clock solutions are likely to be satisfactory; on the other hand, in systems where conflicts are more frequent and/or communication more intermittent, more sophisticated solutions, such as the one we propose here, may perform significantly better. (We explain in the next section how our proposal, which combines agreement and dominance declarations, smoothly handles the examples in Figures 1 and 2.)

Harmony: A Motivating Application. Our interest in conflict resolution algorithms originates in our work on Harmony [4,5], a generic "data synchronizer," capable of reconciling data from heterogeneous, off-the-shelf applications that were developed without synchronization in mind. For example, Harmony can be used to synchronize collections of bookmarks from several different browsers (Explorer, Safari, Mozilla, or OmniWeb), or to keep appointments in MacOS X iCal or Gnome Evolution up-to-date with our appointments in Palm Datebook or Unix ical formats. The current Harmony prototype is able to synchronize only pairs of replicas, with pairwise reconciliation triggered by explicit user synchronization attempts such as putting a PDA into a cradle (perhaps attached to a disconnected laptop). This scheme extends fairly smoothly from pairs to small collections of replicas by iterated pairwise synchronization, but becomes awkward as the set of replicas grows. The work in this paper was inspired by the goal of extending Harmony to handle large numbers of replicas.

Several features of Harmony conspire to make conflicts likely to appear relatively frequently. First, because of its loose coupling with the applications whose data it reconciles, Harmony is a state-based reconciliation system [6]. Unlike operation-based systems, where the system keeps a log of all operations and may be able to resolve conflicts by merging the operation logs on two replicas, state-based systems cannot, in general, merge updates that modified the same atomic values. Second, Harmony reconciles updates between systems such as PDAs that may operate disconnected for long periods of time. Third, we have observed that, even with small numbers of replicas, it often happens that identical updates are entered at different nodes—particularly when the same user owns multiple devices.

Our Results. Since causal histories are not able to satisfactorily handle reconciliation in systems such as Harmony, we develop in this work a new reconciliation framework offering notions of both dominance and agreement, allowing users to resolve conflicts by explicitly specifying the prior events they want to take into account. In §2 we specify this framework precisely by defining legal sequences of local updates, dominance and agreement events, and communications between replicas and showing how to calculate, at each replica, which events will be reported as "maximal" and which as "conflicting."

Our main contribution, in §3, is an algorithm implementing our specification under the assumption that communication is "reciprocal"—after one replica has sent its current state to another, it will wait for a message from the other before sending its own state to that replica again. This algorithm has the property that the information stored at any replica and the sizes of the messages sent between replicas are bounded, in the worst case, by a polynomial function ($O(n^4)$, to be precise) of the number of replicas in the system. §4 discusses related work. Omitted proofs can be found in an accompanying technical report, available on the Harmony home page [5].

2 An Agreeable Reconciliation Framework

A reconciliation framework has three choices when comparing the same object on two different replicas. It can decide that the two objects have *equivalent* values, and do nothing. It can decide that one value is *better than* the other, and modify one replica. Or, it can decide that the two objects are *in conflict* and require external reconciliation. Our goal is to design a consistency maintenance mechanism that can reduce the number of objects that the system decides are in conflict, with less user intervention than conventional causal histories.

The key to achieving this is recognizing "agreement events" as first class citizens. A reconciliation system based on causal history, implements the better-than relation through causal order: u is better-than v if v is in the causal history of u, they are equivalent only if they are identical, and in conflict if u and v are causally unrelated. In our framework it is no longer the case that the simple fact of a node knowing about an event implies that a new update event at that node is better than that prior event — instead we offer a richer 'better-than" relation (defined formally at the end of this section). The user may declare that two or more updates *agree*, or that an update *dominates* another update, or leave two updates unrelated. The system remembers these declarations, so that, if an update u is better-than another update v then u is also better-than all updates equivalent to v, even if they are not in the (conventional) causal history of u or v. Rather than basing our notion of better-than simply on a "knows about" relation (i.e., causal order), we now require users to specify whether the new update u "took v into account" (defined formally below) and, if so, whether through agreement or domination. Agreement events introduce the possibility that two distinct events can be considered equivalent.

This seemingly small shift raises a rather subtle new issue. By introducing "equivalence" we allow the possibility of cycles in the graph of the took-into-account relation. Consider a scenario where two conflicting values x and y were both known about by two different replicas. One decided that y was better than x; the other decided that x was better than y. When the replicas communicate with each other, they discover a cyclical took-into-account relation. Such cycles represent a new sort of conflict—a situation in which users at two or more replicas have given the system conflicting guidance about how to repair a previous conflict! How should we treat such cycles of taking-into-account? In general, there may be multiple distinct values in the cycle, so we cannot pick a single value from the cycle that the system should converge to. The question, then, is not how the values in the cycle relate to each other, but how other values relate to the cycle—i.e., how we can resolve this conflict and allow the replicas to converge by finding or creating values that are not taken into account by others. We address this issue with the notion of *dominance* defined later in this section.

Preliminaries. We assume a fixed set of n replicas, called R^a, R^b, etc. (The development extends straightforwardly to a dynamically changing set of replicas. The main challenge is discovering when information about replicas that have left the system can be garbage collected; standard techniques used in vector clock systems should apply.) The variables α, β, etc. range over indices of replicas. For simplicity, we focus on the case where each replica holds a single, atomic value.

External actions (by the user or a program acting on the user's behalf) that change the value at some replica are represented as *events*, written v_i^α, where α

is the replica where the event occurred and i is a local sequence number that distinguishes events on replica α.

An event is a *predecessor* of all local events that occur after it—that is, v_i^α is a predecessor of all v_j^α with $j > i$; similarly, v_i^α is a *successor* of all events v_j^α with $j < i$. We use v_{i+}^α and v_{i-}^α as variables ranging over successor and predecessor events of v_i^α. When the location or precise local sequence number of an event are not important, we lighten notation by dropping super- and/or subscripts, writing events as just v, v^α, v_+^α, v_-^α, etc.

Our specification uses a structure called a *history graph* (or just *graph*) to represent the state of knowledge at a particular replica at a particular moment in the whole system's evolution. A history graph is a directed graph whose vertices are events and whose edges represent "took into account" relations between events. There are two kinds of edges: an edge $v \longrightarrow w$, pronounced "*v takes w into account through dominance*," represents the fact that event v was created taking w into account and dominating it, while an edge $v \Longrightarrow w$, pronounced "*v takes w into account through agreement*," represents the fact that v and w were declared in *agreement* by the creator of v. (Note that we are not necessarily requiring that v and w have the same *value* in order to be declared in agreement; typically they will, but it may sometimes be useful to resolve a conflict between different values by declaring that either one is acceptable and there is no need for every replica to converge to the same one.) We use G^α to denote the history graph for replica R^α. The set of events in G^α at any given moment is the set of events in the standard causal history of R^α (in contrast, the set of edges in G^α may be only a subset of the set of edges representing causal order).

The set of events and edges reachable in a graph G from an event v, including v itself, is called the *cone* of v, written *cone*(v). This set represents the events v transitively took into account when it was created. We will maintain the invariant that edges originating at an event can be created only at its time of creation, so that the set of events reachable from v will not change over time; moreover, because entire history graphs are exchanged when replicas communicate (at the level of the specification, though of course not in the implementation we describe later), any graph G that contains v will also include *cone*(v); for this reason, we do not bother annotating *cone*(v) with G.

Another important invariant property is *equivalence*. We first define $G_\equiv^\alpha$, the graph obtained from G^α by symmetrizing its $\Longrightarrow$ edges, adding an edge $v \Longrightarrow u$ for each existing edge $u \Longrightarrow v$. Two events u and v are now said to be *equivalent* in G^α if there is a path from u to v in $G_\equiv^\alpha$ consisting only of $\Longrightarrow$ edges. Because replicas exchange whole history graphs, if two events become equivalent at some point in time in the history graph at some replica R^α, they will remain equivalent at all replicas that ever hear (transitively) from R^α. We refer to the partitions induced by this equivalence as *equivalence classes*, or just *classes*.

For a pair of classes E and E', we say E takes E' into account if there exist events $x \in E$ and $y \in E'$ with $y \in cone(x)$. We noted above that there can be cycles in the took-into-account relation: two distinct equivalence classes may each contain an event that has an event from the other in its cone. For example, suppose that the latest (conflicting) values in replicas R^a and R^b are v^a_i and v^b_j, respectively, and that G^a and G^b both contain the complete system history. R^a tries to reconcile the conflict by *adopting* the value of v^b_j (by creating an event v^a_{i+1} with the same value as v^b_j and declaring v^a_{i+1} to be in an equivalence class E with v^b_j). R^b tries to reconcile the conflict by similarly adopting the value of v^a_i, by putting v^b_{j+1} in an equivalence class E' with it. E takes E' into account, because v^a_i is in the cone of v^a_{i+1}; similarly, E' takes E into account because v^b_j is in the cone of v^b_{j+1}. We call such situations *reconciliation conflicts*, since they arise when users at different replicas make different decisions about which of a set of conflicting events should be preferred.

In general, a class can belong to multiple cycles—i.e., it can be involved simultaneously in multiple reconciliation conflicts. To arrive at a clear notion of "better-than", we will define a *dominance* relation. We consider strongly connected components of the graph $G^\alpha_\equiv$ (i.e., sets of events such that there is some path from every event in the set to every other event in the set), which we refer to simply as *components*. Every pair of classes in a component belongs to some cycle denoting a reconciliation conflict, and so intra-component "took into account" relations between events cannot be used to determine dominance.

Now, a class E is said to *dominate* a class E', written $E > E'$, if E and E' belong to different components and there exist events $x \in E$ and $y \in E'$ with $y \in \mathit{cone}(x)$. Note that $E > E'$ implies $E' \not> E$ because of the assumption that the two are in different components.

We say that an event $v^\beta_i \in G^\alpha$ is *latest* if no successor event v^β_{i+} belongs to G^α. We are particularly interested in events belonging to classes that are not dominated by other classes and, among these, in the ones that are latest: if the entire system is going to converge to a single value (or set of equivalent values), such events are the only possible candidates. Formally, we say that a class E is a *maximal class* if it contains a latest event and there is no class E' with $E' > E$. An event v is a *maximal event* if it is a latest event in a maximal class.

When can a replica R^α conclude that there is no conflict between the values in G^α? Based on our definition of dominance, it is easy to see that, if all maximal events belong to the same (maximal) class E, we can be sure that the events in E took every event in G^α into account and that no other events took them into account, implying that there is no conflict between these events (at least according to the present local state of knowledge) and that these events are "better than" all other events. Rule 3 in the specification below guarantees that R^α will then adopt an event from E.

Let us see how our model applies to the examples we discussed in §1. The initial values at the replicas are represented by v^a, v^b and v^c respectively. For the example in Figure 1, after receiving state updates from R^a and R^b, R^c joins v^a, v^b, and v^c into an equivalence class by creating a new event v^c_+ and adding $\Longrightarrow$ edges from v^c_+ to them. Independently, R^b, after receiving R^a's state, makes v^a, v^b and v^b_+ into an equivalence class. Fortunately, these new events v^c_+ and v^b_+ do not conflict, and anyone who later hears of both can calculate that v^a, v^b, v^c, v^b_+, and v^c_+ all belong to the same equivalence class, so that any new event dominating any of them will also dominate all the others. Similarly, in the scenario in Figure 2, R^b makes v^a, v^b, and v^b_+ equivalent and later R^c adds v^c to this equivalence class (via a new event v^c_+ with $\Longrightarrow$ edges to v^c, v^a, v^b, and v^b_+). Independently, R^a adds a new event v^a_+ (with value y), dominating v^a. Henceforth, regardless of the order of messages from R^a and R^c, any replica that learns of both v^a_+ and v^c_+ can see that v^a_+ dominates all the values from the other replicas.

Continuing the example, it is possible that, for some time, some other replica R^d may hear only from R^a and R^b(before R^b creates the event v^b_+) but not R^c and therefore believe that events v^a_+ and v_b are in conflict. Once it hears from R^c as well, the apparent conflict will disappear. But if, in the meantime, the user at R^d decides to repair the apparent conflict by declaring that v^b dominates v^a_+ (by creating an event v^d dominating v^a_+ and then another event v^d_+ in agreement with both v^b and v^d), then a reconciliation conflict will be created, requiring one more user intervention to eliminate.

We have now presented all the basic concepts on which our reconciliation scheme is based. It remains to specify exactly what state is maintained at each replica and how this state changes as various actions are performed. These actions are of two sorts: local actions by the user, and gossiping between replicas, in which one replica periodically passes its state to another, which updates its picture of the world and later sends the combined state along to yet other replicas. We will not be precise in this paper about exactly how replicas determine when and with whom to communicate—we simply treat communication as a non-deterministic transmission of state from one replica to another. (We have in mind a practical implementation based on a gossip architecture such as [7].) However, to ensure that our implementation in §3 can work in bounded space, we need to make one restriction on the pattern of communication: after a replica R^α has sent its state to a particular neighbor R^β, it should wait until it receives an update message from R^β before sending another of its own. (Indeed, in the accompanying technical report we prove that, with unrestricted asymmetric communication, no representation that operates in bounded space can implement the specification correctly.) This *reciprocality* of communication bounds the number of possible open events on each replica. To guarantee reciprocality, each replica maintains a boolean flag $\mathit{CanSend}(\beta)$ for each replica R^β, initially set to true. It is reset to false each time R^α sends a communication to R^β and reset to true each time R^α receives a communication from R^β.(This definition places a somewhat unrealistic constraint on the communication substrate: it assumes that messages are not lost and are not reordered in transit. We believe that this constraint can probably be relaxed, but we do not have a proof yet.)

Specification. The state of the entire system at any moment comprises the following information: a history graph G^α for each replica R^α, a reciprocity predicate $\mathit{CanSend}^\alpha$ for each replica R^α, and a current event $\mathit{Current}^\alpha \in G^\alpha$ for each replica R^α. The initial state of the system has all history graphs G^α containing a single vertex v_{init} and no edges, $\mathit{CanSend}^\alpha(\beta) = \mathit{true}$ for all α and β, and $\mathit{Current}^\alpha = v_{init}$ for all α. At any given moment, a user (or user-level program) at replica R^α can query the current event at R^α, as well as the current set of maximal events in G^α and, for each of these, the other events in its class.

Each step in the system's evolution must obey one of the following rules:

1. A replica R^α may generate a new event v_i^α, where $i = 1 + \max(j \mid v_j^\alpha \in G^\alpha)$, taking into account some subset W (containing $\mathit{Current}^\alpha$) of the maximal events in G^α. The current event $\mathit{Current}^\alpha$ is set to v_i^α. A vertex v_i^α and an edge $v_i^\alpha \longrightarrow w$ for each $w \in W$ are added to the graph G^α.
2. A replica R^α may generate a new event v_i^α, where $i = 1 + \max(j \mid v_j^\alpha \in G^\alpha)$, and declare it to be in agreement with some subset W of the maximal events in G^α. A vertex v_i^α, and an edge $v_i^\alpha \Longrightarrow w$ for each $w \in W$, are added to the graph G^α. If $\mathit{Current}^\alpha \notin W$ and $\mathit{Current}^\alpha$ is a predecessor of v_i^α, an edge $v_i^\alpha \longrightarrow \mathit{Current}^\alpha$ is also added to the graph. The current event $\mathit{Current}^\alpha$ is then set to v_i^α.

   The choice of W is constrained by one technical condition: Let $E_1 \ldots E_p$ be the maximal classes containing the subset of maximal events W. This operation is allowed only if for each replica R^β, the set of events from the creating replica R^β that will now be in the new merged class, call it E, correspond to a contiguous range of indices—that is, for any $i < j < k$ if $v_i^\beta \in E$ and $v_k^\beta \in E$ then $v_j^\beta \in E$. The interpretation of this restriction is that a user is not allowed to establish agreement between two distinct events v_i^β and v_k^β created by a replica R^β unless it can do so for every event that was created by R^β in between.
3. A replica R^α may send its current state to another replica R^β, provided that $\mathit{CanSend}^\alpha(\beta) = \mathit{true}$. The history graph G^β is replaced by $G^\beta \cup G^\alpha$. A new maximal event x (if one exists) in the combined G^β is *better-than* $\mathit{Current}^\beta$ (and hence overwrites it) if $\mathit{Current}^\beta$ is not a a maximal event in G^β. The reciprocity predicates are updated with $\mathit{CanSend}^\alpha(\beta) = \mathit{false}$ and $\mathit{CanSend}^\beta(\alpha) = \mathit{true}$.

3 A Bounded-Space Implementation

We now develop an efficient implementation based on a *sparse* representation of history graphs, written S^α. The crucial property that we establish is that the size of S^α depends only on the maximum number of distinct replicas that ever communicate with R^α. For analyzing this representation, it is helpful to be able to refer to the local state at any replica at particular points in time. We introduce an imaginary *global time counter* t, which is incremented each time any action is taken by any replica—i.e., each time the whole system evolves one step by a replica taking one of the steps described in §2. The graph at replica R^α at time t is written $G^\alpha(t)$.

There are two core concepts that facilitate our polynomial-space representation of all "relevant" information contained in a history graph. The first is the notion of *open* and *closed* events, and the second is the notion of a *sparse cone* of an event v. We start by decribing these concepts and some of their properties.

Open and Closed Events. The *creator replica* of an event $v = v_i^\alpha$ is the replica R^α at which the event was created. It is clear from the specification that only a creator replica can add edges originating from v to its graph, and only at the time v is created. It can later add an $\Longrightarrow$ edge into v (in addition to the $\longrightarrow$ edge that is always added), when it creates v's immediate successor. Another replica that later hears about v can create $\Longrightarrow$ or $\longrightarrow$ edges into v as long as v is a maximal event in its local graph.

No replica R can afford to forget about an event or any edges from or into it, as long as it is possible for some replica to create edges into it, lest R be the only witness to a relevant equivalence edge. Reciprocal communication enables us to track such "critical" events with bounded space.

An event v is *closed* if, at every replica R^α, if $v \in G^\alpha$ then $v_+ \in G^\alpha$ for some successor v_+ of v; an event that is not closed is *open*. If v is closed, then any replica that hears about v will simultaneously hear about a successor of v. It follows from this that a closed event can never be a latest event at any replica (hence also not a maximal one), and that, once an event is closed, it stays closed forever. No edges can be created to or from a closed event at any replica at any time in the future.

An omniscient observer can see when an event becomes closed. But how can a replica know that an event is closed using only locally available information?

We maintain a data structure O^α at every replica R^α that can be used to certify that events are closed. The creator replica of an event v marks it closed when it knows that all other replicas who ever heard of v, have also heard of a successor to v. The other replicas mark the event closed when they hear that it has been marked closed by the event's creator replica. We say that an event that is marked closed by replica R^α is *closed at* R^α. An event that is not closed at a given replica is *considered open* at that replica.

An event can be simultaneously considered open at certain replicas and closed at others. The data structure O^α ensures that, at any time t, for each non-latest event v_i^α considered open at a replica R^α, we can identify a pair of replicas in the system, say (R^β, R^γ), such that (i) R^γ first learnt about v_i^α from R^β and (ii) R^α is certain that R^β is aware of a successor of v but it is uncertain if this is also the case for R^γ. In this case, R^α can not yet consider v_i^α closed as R^γ may possibly create an edge to the event v_i^α. We refer to such a pair as a *witness* to event v_i^α being open at time t. The reciprocal communication property allows us to ensure that each pair of replicas can serve as a witness to at most two open events from any replica. We use this fact to argue that at most $O(n^3)$ events are considered open at any replica. The data structure O^α maintains $O(n)$ information per open event and hence has size $O(n^4)$. Theorem 3.2 shows that the space complexity of S^α (which includes O^α) is also bounded by $O(n^4)$.

Sparse Cone. The *sparse cone* of an element v_l^α, written *sparse-cone*(v_l^α), can be derived from its cone in the following manner. For each $\beta \neq \alpha$, let j be the largest index, if any exists, such that $v_j^\beta \in$ *cone*(v_l^α). If such a j does exist, then add the vertex v_j^β and a directed edge (v_l^α, v_j^β) to *sparse-cone*(v_l^α).

Note that both *cone*(v) and *sparse-cone*(v) are determined at the time of v's creation and are time invariant. Also, even though *cone*(v) can be arbitrarily large, *sparse-cone*(v) is $O(n)$ in size and implicitly contains all the necessary information from *cone*(v), in the sense that, for any element w, we can determine whether or not $w \in$ *cone*(v) by examining *sparse-cone*(v).

Sparse Representation. We now describe a polynomial-space representation that summarizes the information contained in $G^\alpha(t)$ at any time t. In the accompanying technical report we show how to maintain this representation incrementally as the system evolves, calculating the compact representation at each step from the compact representation at the previous step, and prove that the representation is correct in the sense that it will report the same maximal events (and equivalence classes) as the specification in §2.

We start with the observation that the graph $G^\alpha(t)$ may be viewed as simply a union of the cones of all the elements known to replica R^α at time t. We will represent $G^\alpha(t)$ by a pair of *sparse graphs*, denoted $H^\alpha(t)$ and $H^\alpha_\equiv(t)$. The sparse graph $H^\alpha(t)$ is defined to be simply the union of the sparse cones of latest events known at R^α at time t. It thus takes $O(n^2)$ space. The sparse graph $H^\alpha_\equiv(t)$, summarizes the information contained in $G^\alpha_\equiv(t)$ as follows. Let $v \leadsto w$ denote the existence of a path from an event v to event w in a graph $G^\alpha_\equiv$. For each open event v^β_i at $R^\alpha(t)$, $H^\alpha_\equiv(t)$ records, for every other replica R^γ, the earliest event v^γ_j from R^γ for which $v^\gamma_j \leadsto v^\beta_i$ in $G^\alpha_\equiv(t)$. (Even though the information contained in $G^\alpha_\equiv(t)$ can be derived from $G^\alpha(t)$, we need to explicitly maintain the graph $H^\alpha_\equiv(t)$ since $H^\alpha(t)$ does not contain all the information in $G^\alpha(t)$.) Formally, for every pair of events v^β_i and v^γ_j in $G^\alpha_\equiv(t)$ such that (i) $v^\gamma_j \leadsto v^\beta_i$ in $G^\alpha_\equiv(t)$, (ii) v^β_i is considered open at $R^\alpha(t)$, and (iii) there is no $j' < j$ such that $v^\gamma_{j'} \leadsto v^\beta_i$ in $G^\alpha_\equiv(t)$, we include in $H^\alpha_\equiv(t)$ the events v^β_i and v^γ_j and a directed edge (v^γ_j, v^β_i). Note that an edge (u, v) in $H^\alpha_\equiv$ merely indicates the existence of a path $u \leadsto v \in G^\alpha_\equiv$ but not whether its edges are $\longrightarrow$ or $\Longrightarrow$ or a mixture of the two.

3.1 Definition. The *sparse representation* at a replica R^α at time t is a 4-tuple $\mathcal{S}^\alpha(t) = \langle \mathrm{O}^\alpha(t), H^\alpha(t), H^\alpha_\equiv(t), \mathcal{C}^\alpha(t)\rangle$, where $\mathrm{O}^\alpha(t)$ is a data structure containing the set of events from each replica that are considered open at R^α as well as the tables to maintain these open events (defined in the accompanying technical report), $H^\alpha(t)$ is the sparse graph derived from $G^\alpha(t)$, $H^\alpha_\equiv(t)$ is the sparse graph derived from $G^\alpha_\equiv(t)$, and $\mathcal{C}^\alpha(t)$ is a collection of sets, one for each event v considered open at R^α, such that the set corresponding to v contains all events in the equivalence class of v.

Whenever replica R^α communicates to another replica R^β, it sends the tuple $\mathcal{S}^\alpha$. The next theorem bounds the size of this communication.

3.2 Theorem. At any time t, $\mathcal{S}^\alpha(t)$ takes $O(n^4)$ space, where n is the number of replicas.

We observed earlier that the number of open events at any replica can be bounded by $O(n^3)$ and the data structure $\mathrm{O}^\alpha(t)$ used to maintain them takes

$O(n^4)$ space. The graph $H^\alpha(t)$ takes $O(n^2)$ space as observed above. The graph $H^\alpha_\equiv(t)$ needs $O(1)$ space for each open event for a total of $O(n^3)$ space. Finally, we can show that the equivalence class of each open event can be described compactly using $O(n)$ space. This gives us a bound of $O(n^4)$ space for $\mathcal{C}^\alpha(t)$.

In order to establish that a replica working with the sparse representation will have the same user-visible behavior as if it were working with the complete history graphs, it suffices to show the following.

3.3 Theorem. A class E is maximal in $G^\alpha(t)$ iff E is maximal in $\mathcal{S}^\alpha(t)$.

The proof of this theorem in the accompanying technical report crucially relies on the properties of open and closed events and sparse cones. The main idea of the proof is to establish two key properties. First, for any pair of classes E, E' in $G^\alpha(t)$ such that each contains a latest element, we can determine, using the graph $H^\alpha_\equiv(t)$, whether or not they belong to the same component in $G^\alpha_\equiv(t)$. Second, if a class E containing a latest element is dominated by another class E' in $G^\alpha(t)$, we show that the graph $H^\alpha(t)$ contains a "witness" to this fact. Since a maximal class always contains a latest element, these two properties together ensure that the set of maximal classes is the same in both $G^\alpha(t)$ and $\mathcal{S}^\alpha(t)$. Finally, we note that, since a latest event is always open, $\mathcal{C}^\alpha(t)$ contains all elements in each maximal class.

4 Related Work

Both theoretical underpinnings and efficient implementation strategies for version vectors [1] and vector clocks [8,9] have received a great deal of attention in the literature and have been used in many systems (e.g. Coda [10,11,12], Ficus [13], and Bengal [14]); numerous extensions and refinements have also been studied—see [15] for a recent survey. We conjecture that some of these ideas can be applied to improve the efficiency of our sparse representation. However, we are not aware of any work in this context that explicitly addresses the main concern of our work—an explicit treatment of declarations of agreement (and dominance) between existing events.

A number of systems have used replica equality (e.g., identity of file contents) as an *implicit* indication of agreement. The user-level filesystem synchronization tool Unison [16], for example, considers two replicas of a file to be in agreement whenever their current contents are equal at the point of synchronization. This gives users an easy way to repair conflicts (decide on a reconciled value for the file, manually copy it to both replicas, and re-synchronize), as well as automatically yielding sensible default behavior when Unison is run between previously unsynchronized (but currently equal) filesystems. A similar strategy is used in Panasync [17].

Matrix clocks [18,19] generalize vector clocks by explicitly representing clock information about other processes's views of the system's execution. We leave for future work the question of whether agreement events such as the ones we are proposing could be generalized along similar lines.

A rather different approach to conflict detection is embodied, for example, in the *hash histories* of Kang et al. [3] and the *version histories* used in the Reconcile file synchronizer [20] and the Clique peer-to-peer filesystem [21]. Rather than deducing causal ordering from reduced representations such as clock vectors, these systems represent the causal history of the system directly—storing and transmitting (hashes of) complete histories of updates. An advantage of such schemes is that their cost is proportional to the number of updates to a file rather than the number of replicas in the system, which may be advantageous in some situations. This suggests that it may be worth considering the possibility of implementing something akin to our naive specification from §2 directly, bypassing the sparse representation.

Reconciliation protocols for optimistically replicated data can be divided into two general categories [22]: *state transfer* and *operation transfer* protocols. We have concentrated on state-based protocols in this work. However, a number of systems (e.g., Bayou [23], IceCube [24], and Ceri's work [25]) reconcile the *operation histories* of replicas rather than their states. It is not clear whether agreement events in the sense we have proposed them could meaningfully be accommodated in this setting.

References

1. Parker, Jr., D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Software Eng. (USA) **SE-9**(3) (1983) 240–247
2. Malkhi, D., Terry, D.B.: Concise version vectors in WinFS. In Fraigniaud, P., ed.: Proceedings of the 19th International Conference on Distributed Computing, DISC 2005. Volume 3724 of Lecture Notes in Computer Science., Springer-Verlag (2005) 339–353
3. Kang, B.B., Wilensky, R., Kubiatowicz, J.: The hash history approach for reconciling mutual inconsistency. In: 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03). (2003)
4. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania (2005) Supersedes MS-CIS-03-42.
5. Pierce, B.C., et al.: Harmony: A synchronization framework for heterogeneous tree-structured data (2006) `http://www.seas.upenn.edu/~harmony/`.
6. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. Journal of Computer and System Sciences (2006) To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.
7. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of PODC'87. (1987)
8. Fidge, C.: Logical time in distributed computing systems. Computer **24**(8) (1991) 28–33
9. Mattern, F.: Virtual time and global states of distributed systems. In et. al., M.C., ed.: Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms. Elsevier Science Publishers B. V. (1989) 215–226

10. Kumar, P.: Coping with conflicts in an optimistically replicated file system. In: 1990 Workshop on the Management of Replicated Data, Houston, TX (1990) 60–64
11. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers **39**(4) (1990) 447–459
12. Kumar, P., Satyanarayanan, M.: Flexible and safe resolution of file conflicts. In: Proceedings of the annual USENIX 1995 Winter Technical Conference. (1995) 95–106 New Orleans, LA.
13. Guy, R.G., Reiher, P.L., Ratner, D., Gunter, M., Ma, W., Popek, G.J.: Rumor: Mobile data access through optimistic peer-to-peer replication. In: Proceedings of the ER Workshop on Mobile Data Access. (1998) 254–265
14. Ekenstam, T., Matheny, C., Reiher, P.L., Popek, G.J.: The Bengal database replication system. Distributed and Parallel Databases **9**(3) (2001) 187–210
15. Baldoni, R., Raynal, M.: A practical tour of vector clock systems. IEEE Distributed Systems Online **3**(2) (2002) http://dsonline.computer.org/0202/features/ bal.htm.
16. Pierce, B.C., Vouillon, J.: What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania (2004)
17. Almeida, P.S., Baquero, C., Fonte, V.: Panasync: dependency tracking among file copies. In: EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop, ACM Press (2000) 7–12
18. Sarin, S.K., Lynch, N.A.: Discarding obsolete information in a replicated database system. IEEE Transactions onSoftware Engineering **13**(1) (1987) 39–47
19. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Principles of Distributed Computing. (1984) 233–242
20. Howard, J.H.: Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab (1999)
21. Richard, B., Nioclais, D.M., Chalon, D.: Clique: a transparent, peer-to-peer collaborative file sharing system. In: International Conference on Mobile Data Management (MDM), Melbourne, Australia. (2003)
22. Saito, Y., Shapiro, M.: Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto (2002)
23. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado. (1995) 172–183
24. Kermarrec, A.M., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube approach to the reconciliation of diverging replicas. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island. (2001) 210–218
25. Ceri, S., Houtsma, M.A.W., Keller, A.M., Samarati, P.: Independent updates and incremental agreement in replicated databases. Distributed and Parallel Databases **3**(3) (1995) 225–246

A Lazy Snapshot Algorithm with Eager Validation

Torvald Riegel[1], Pascal Felber[2], and Christof Fetzer[1]

[1] Dresden University of Technology, Dresden, Germany
torvald.riegel@inf.tu-dresden.de, christof.fetzer@tu-dresden.de
[2] University of Neuchâtel, Neuchâtel, Switzerland
pascal.felber@unine.ch

Abstract. Most high-performance software transactional memories (STM) use optimistic invisible reads. Consequently, a transaction might have an inconsistent view of the objects it accesses unless the consistency of the view is validated whenever the view changes. Although all STMs usually detect inconsistencies at commit time, a transaction might never reach this point because an inconsistent view can provoke arbitrary behavior in the application (e.g., enter an infinite loop). In this paper, we formally introduce a lazy snapshot algorithm that verifies at each object access that the view observed by a transaction is consistent. Validating previously accessed objects is not necessary for that, however, it can be used on-demand to prolong the view's validity. We demonstrate both formally and by measurements that the performance of our approach is quite competitive by comparing other STMs with an STM that uses our algorithm.

1 Introduction

The recent move to multi-core processors has resulted in an increased research interest in *software transactional memory* (STM) [1]. STMs have been introduced as a mean to support lightweight transactions in concurrent applications. Transactions execute concurrently and those that fail to commit automatically roll back and restart their execution.

In STMs there is currently a tradeoff between consistency and performance. Several high-performance STM implementations [2,3,4] use optimistic reads in the sense that the set of objects read by a transaction might not be consistent. Consistency is only checked at commit time, i.e., commit *validates* the transaction. However, having an inconsistent view of the state of the objects during the transactions might, for example, result in infinite loops or the throwing of exceptions. These failures must then be detected and masked by the STM or the program's runtime environment, which is often both difficult and costly. Explicit, programmer-controlled validation is usually considered too difficult for programmers.

Example 1. Consider a linked list and two operations: (1) *search* iterates through the list until it finds a specific element while (2) *sort* re-orders the elements.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 284–298, 2006.

Consider that the list contains elements o_2 and o_1 in that order. Transaction T_1 sorts the list, which leads to re-ordering o_1 before o_2. If transaction T_2 iterates through the list but reads o_2 before the execution of T_1 (o_2 was the first object of the list) and o_1 after the sort operation has completed, it will experience a cycle and will loop forever.

Validation, on the other hand, can be costly (see Section 4) if it is performed in the obvious way, i.e., checking every object previously read. Typically, the validation overhead grows linearly with the number of objects a transaction has accessed so far. When one is forced to validate after each step, this could result in a validation overhead that grows quadratically with the number of objects accessed by a transaction.

In this paper, we introduce a lazy snapshot algorithm (LSA) that efficiently constructs an always consistent snapshot for transactions. Reads of a transaction are invisible to other transactions; the consistency of a transaction is verified by maintaining a validity interval for snapshots. In this way, an STM can efficiently verify during each object access that the snapshot of the objects that a transaction has seen so far is consistent.

We have built an object-based STM using LSA, to which we will refer as LSA-STM in what follows. It ensures linearizability [5] for read-only and update transactions. Our performance measurements demonstrate that the performance of LSA is very competitive with other STM implementations even when ensuring linearizability and always providing transactions with a consistent view.

In earlier work [6], we showed how to use LSA to build STMs that provide snapshot isolation [7]. The key idea of snapshot isolation (SI) is to provide each transaction T with a consistent snapshot of all objects at a given time. Writes of T occur atomically but possibly at a later time than that of the snapshot. This decoupling of the reads and the writes has the potential of increasing the transaction throughput but also gives application developers less ideal semantics than, say, STMs that guarantee linearizability [5]. However, SI always provides a transaction with a consistent view which avoids the programming anomalies that we described above.

In what follows, we first give a brief overview of the related work (Section 2). We then introduce LSA and demonstrate some of its properties (Section 3). Finally, we describe the results of our performance evaluation (Section 4) and conclude the paper (Section 5).

2 Related Work

Software Transaction Memory is not a new concept [1] but it recently attracted much attention because of the rise of multi-processor and multi-core systems. There are word-based [8] and object-based [9] STM implementations. The design of the latter, Herlihy's DSTM, is used by several current STM implementations. Most STM implementations are obstruction-free [10] and use contention managers [9] to deal with conflicts and ensure progress. Our LSA-STM is object-based and obstruction-free [10] and thus, uses some of DSTM's concepts. However, LSA-STM is a multi-version STM, whereas DSTM keeps at most two versions of an object but only uses the most recent version. We previously presented SI-STM [6], which uses LSA but provides, in addition to strict transactional consistency, support for snapshot isolation, which can increase the performance of suitable applications.

In DSTM and most of the high-performance STMs in general, reads by a transaction are invisible to other transactions: to ensure that consistent data is read, one must validate that all previously read objects have not been updated in the meantime. If reads are to be visible, transactions must add themselves to a list of readers at every transactional object they read from. Reader lists enable update transactions to detect conflicts with read transactions. However, the respective checks can be costly because readers on other CPUs update the list, which in turn increases the contention of the memory interconnect. Scherer and Scott [11,12] investigated the trade-off between invisible and visible reads. They showed that visible reads perform much better in several benchmarks but, ultimately, the decision remains application-specific. Marathe *et al.* [13] present an STM implementation that adapts between eager and lazy acquisition of objects (i.e., at access or commit time) based on the execution of previous transactions. However, they do not explore the trade-off between visible and invisible reads but suggest that adaptation in this dimension could increase performance. Cole and Herlihy propose a snapshot access mode [14] that can be roughly described as application-controlled invisible reads for selected transactional objects with explicit validation by the application. Dice *et al.* show in [15], a recent improvement of earlier work [4], how to use a global version clock to improve the performance of low-overhead STMs. However, the validity of snapshots is fixed to the start time of a transaction and is not extended on demand. The only other multi-version STM that we are aware of is [16], although snapshots are not computed dynamically and conflict detection of update transactions only occurs at commit time. Furthermore, in that STM design, every commit operation, including the upgrade of transaction-private data to data accessible by other threads, synchronizes on a single global lock. No performance benchmark results were provided.

Read accesses in our LSA-STM are invisible to other transactions but do not require revalidation of previously read objects on every new read access. We show that our LSA facilitates inexpensive validation by maintaining a validity range in which a transaction is valid. In this way we get most of the benefits of visible and invisible reads but at a much lower cost.

3 Lazy Snapshot Algorithm

Before we can describe our lazy snapshot algorithm in Section 3.2, we first need to introduce some notations in Section 3.1. We show the correctness of LSA in Section 3.4.

3.1 Notations

A transactional memory consists of a set of shared objects $o_1, \dots, o_n \in O$. Transactions are either *read-only*, i.e., they do not write to any object, or are *update* transactions, i.e., write to one or more objects.

Our transactional memory has a global counter, CT, that counts the number of update transactions that have committed so far. When an update transaction commits, it acquires a unique CT timestamp and creates a new version of the state of the transactional memory with this timestamp. Unlike in many other systems, this counter is not incremented when a read-only transaction commits. The goal is to improve the caching hit rate for this counter. We use CT as our time base, that is, all times given in the following are given with respect to this CT counter. For example, we denote the content of object o_i at (commit) time t, by o_i^t. Note that CT is a simple integer counter.

A transaction T accesses a finite set of objects $O_T \subseteq O$. We assume that objects are only accessed and modified within transactions. Hence, we can describe a history of an object with respect to the global commit time CT. The sequence $H_i = (v_{i,1}, \dots, v_{i,j}, \dots)$ denotes all the times at which updates to object o_i are committed by some update transactions. $v_{i,1}$ is the time the object is created. Sequence H_i is strictly monotonically increasing, i.e., $\forall j < |H_i| : v_{i,j} < v_{i,j+1}$. To simplify our equations, we assume that the first element of H_i, i.e., $v_{j,1}$ is 0 (all objects are created at time zero) and the last element is ∞ if $|H_i|$ is finite.

We say that the version j of object o_i ($j < |H_i|$) is valid from $v_{i,j}$ to $v_{i,j+1} - 1$. We call this the *validity range* and denote this by

$$R_{i,j} := [v_{i,j}, v_{i,j+1} - 1].$$

A transaction T might not use the most recent version o_i^t of an object o_i when accessing the object the first time at t. Instead an older version might be used. Hence, for each object o_i in transaction O_T we denote the version of o_i by o_i^*, its version number by $v_{i,*}$, and its validity range by $R_{i,*}$ (o_i^*, $v_{i,*}$, and $R_{i,*}$ are specific to transaction T but, for simplicity, we do not make this explicit in our terminology).

By $\lfloor o_i^t \rfloor$ we denote the time of the most recent update of object o_i performed no later than time t, i.e., this update is still valid at global time t. We define $\lfloor o_i^t \rfloor$ as follows:

$$\lfloor o_i^t \rfloor := v_{i,j} \mid v_{i,j} \leq t \wedge t < v_{i,j+1}.$$

By $\lceil o_i^t \rceil$ we denote the time until which the version of object o_i that is valid at time t remains valid:

$$\lceil o_i^t \rceil := v_{i,j+1} - 1 \mid v_{i,j} \leq t \wedge v_{i,j+1} > t.$$

We define the *validity range* R_T of a transaction T to be the time range during which each of the objects accessed by T is valid. This is the intersection of the validity ranges of the individual versions accessed by a transaction:

$$R_T := \bigcap_{o \in O} R_{i,*}.$$

We say that the object versions accessed by transaction T, i.e., $\{o_i^* | \forall o_i \in O_T\}$ are a *consistent snapshot* if the validity range R_T of T is non-empty. Note that because of the intersection, the object versions contained in a consistent snapshot are always the most recent versions at any time $t \in R_T$.

An *update* transaction T writes to a subset of the objects in O_T. In our implementation, writing to an object always includes reading the object. We denote by $U_T \subseteq O_T$ the set of objects written to by T.

3.2 Snapshot Construction

The main idea of LSA (see Algorithm 1) is to construct consistent snapshots on the fly during the execution of a transaction and to—lazily—extend the validity range on demand. By this, we can reach two goals. First, transactions working on a consistent snapshot always read consistent data. Second, verifying that there is an overlap between the snapshot's validity range and the commit time of a transaction can ensure linearizability (if that is desired). Note that LSA-STM uses a lock-free implementation of LSA, whereas we assume sequential execution for the pseudocode in Algorithm 1 for simplicity. We will first describe the basic algorithm and then show that it is correct in Section 3.4.

Which objects are accessed by a transaction is determined during the execution of a transaction. The final R_T might not even be known at the commit time of the transaction. We therefore maintain a *preliminary validity range* R'_T. When a transaction T is started, we set R'_T to $[CT, \infty]$ (lines 2–3, $min(R'_T)$ and $max(R'_T)$ denote the lower and upper bound of R'_T). Note that R'_T will never assume a value smaller than the start time of T.

When accessing the most recent version of o_i, it is not yet known when this version will be replaced by a new version. We therefore approximate $R_{i,*}$ at time t by a *preliminary* range $R'_{i,*} = R_{i,*} \cap [0, t]$ and we set the new range to $R'_T \cap R'_{i,*}$ (lines 18–19). During the execution of a transaction, time will advance and thus the preliminary validity ranges might get longer. We can try to *extend* R'_T by recomputing $max(R'_T)$ (lines 14 and 29–37). Note that this is not required for correctness—it only increases the chance that a suitable object version is available.

Read accesses are optimistic and invisible to other transactions. LSA assumes that a system always keeps the most recent version of an object. In addition, LSA might also have access to some old versions (e.g., which have not yet been garbage collected) that can be used to increase the probability to create a consistent snapshot. When a transaction reads object o_i at time t, LSA tries to select the newest object version from H_i that still exists and that keeps the snapshot consistent, i.e., R'_T non-empty.

If the most recent version of o_i cannot be used because it was created after R'_T, we might still read some older version whose validity range overlaps R'_T. In that case, we simply set the new range to $R'_T \cap R'_{i,*}$ and we mark the transaction as "closed" to indicate that it cannot be extended anymore (lines 21–23). If no such version exists anymore, the transaction needs to be aborted. We omitted this in the simplified pseudocode of LSA.

By construction of R_T, LSA guarantees that a transaction started at time t has a snapshot that is valid at or after the transaction started, i.e., $\min(R'_T) \geq t$. Hence, a read-only transaction can commit iff it has used a consistent snapshot

Algorithm 1. Lazy Snapshot Algorithm (LSA)

```
 1: procedure START($T$)                                  ▷ Initialize transaction attributes
 2:     $T.min \leftarrow CT$                                          ▷ $= min(R' )$
 3:     $T.max \leftarrow \infty$                                      ▷ $= max(R' )$
 4:     $T.O \leftarrow \varnothing$                             ▷ Set of objects accessed by $T$
 5:     $T.open \leftarrow$ true                                  ▷ Can $T$ still be extended?
 6:     $T.update \leftarrow$ false                              ▷ Is $T$ an update transaction?
 7: end procedure

 8: procedure OPEN($T, o\ , m$)                  ▷ $T$ opens $o$ in mode $m$ (read or write)
 9:     if m = write then
10:         $T.update \leftarrow$ true
11:     end if
12:     if $\lfloor o \quad \rfloor > T.max$ then              ▷ Is most recent version too recent?
13:         if $T.update \wedge T.open$ then                                ▷ Try to extend?
14:             EXTEND($T$)
15:         end if
16:     end if
17:     if $\lfloor o \quad \rfloor \leq T.max$ then                  ▷ Can we use the latest version?
18:         $T.min \leftarrow \max(T.min, \lfloor o \quad \rfloor)$         ▷ Yes, $T$ remains open if it is still open
19:         $T.max \leftarrow \min(T.max, CT)$
20:     else if $\neg T.update \wedge VersionAvailable(o \quad )$ then
21:         $T.open \leftarrow$ false                          ▷ No, $T.max$ has reached its maximum
22:         $T.min \leftarrow \max(T.min, \lfloor o \quad \rfloor)$
23:         $T.max \leftarrow \min(T.max, \lceil o \quad \rceil)$
24:     else                                                     ▷ Cannot maintain snapshot
25:         ABORT($T$)
26:     end if
27:     $T.O \leftarrow T.O \cup \{o\ \}$                                          ▷ Access object
28: end procedure

29: procedure EXTEND($T$)                              ▷ Try to extend the validity range of $T$
30:     $T.max \leftarrow CT$
31:     for all $o\ \in T.O$ do                                ▷ Recompute the whole validity range
32:         $T.max \leftarrow \min(T.max, \max(R'_{\ ,*}))$
33:     end for
34:     if $T.max < CT \wedge T.update$ then
35:         ABORT($T$)                         ▷ Update transaction must access most recent versions
36:     end if
37: end procedure

38: procedure COMMIT($T$)                                          ▷ Try to commit transaction
39:     if $T.update$ then
40:         $CT \quad \leftarrow (CT \leftarrow CT + 1)$                 ▷ Acquire $T$'s unique commit time $CT$
41:         if $T.max < CT \quad - 1$ then
42:             EXTEND($T$)                   ▷ For update transactions, $CT$ and $R'$ must overlap
43:             if $T.max < CT \quad - 1$ then
44:                 ABORT($T$)
45:             end if
46:         end if
47:     end if                                                  ▷ $T$ can now be safely committed
48: end procedure
```

(i.e., R'_T is non-empty). The commit time CT is not increased when committing a read-only transaction because nothing has been modified.

3.3 Update Transactions

Informally, an update transaction T performs the following steps when committing: (1) acquire a unique commit time CT_T from the CT time base (line 40), (2) validate T (lines 41–46), and (3) set T's state to committed if the validation was successful and to aborted otherwise (not shown). Update transactions can only commit if their validity range and their unique commit time (i.e., the global version that they are going to produce) overlap; in this case, the transaction is atomic. This is checked during the validation step (i.e., $(CT_T - 1) \in R'_T$). Therefore, accessed object versions must always be the most recent versions during the transaction and the validity range must be "open". If this is not possible, the transaction is marked as aborted (lines 25 and 35) because it would not be able to commit anyhow.

If it is possible to update a most recent version (i.e., R'_T remains non-empty), LSA atomically marks the object that it is writing (visible write, not shown in Algorithm 1). When another transaction T_1 tries to write o_i, T_1 will see the mark and detect a conflict. In that case, one of the transactions might need to wait or be aborted. This task is typically delegated to a *contention manager* [9], a configurable module whose role is to determine which transaction is allowed to progress upon conflict.

Setting the transaction's state atomically commits—or discards in case of an abort—all object versions written by the transaction and removes the write markers on all written objects (as in DSTM [9]).

If a transaction reads from the most recent version of an object that is write-marked by another transaction that has not yet committed, then the validity of the most recent version of this object ends at CT (the current time); the updating transaction cannot commit at a time earlier than $CT + 1$. This allows the STM to defer read-write conflicts to the commit time of the updating transaction, which minimizes the duration of such conflicts and lets reading transactions run unobstructed for a longer time.

For STMs that provide snapshot isolation [6], validation requires checking that only all object versions written by T are still valid at $CT_T - 1$. Since we use visible writes, this check is performed implicitly because write/write conflicts will result in one of the two conflicting transactions being aborted (or delayed) by the contention manager.

3.4 Linearizability

We sketch that transactions executed by an STM in the way outlined previously are linearizable.[1] To show this, we need to show that T takes effect atomically in between the start of T at time t and its commit time CT_T. We show that for read-only transactions and then for update transactions. However, we introduce some lemmas first.

By the construction of LSA, it is guaranteed that the lower bound of R'_T is always greater than or equal to the start time of transaction T.

Lemma 1. *For any transaction T with a non-empty R'_T, it is guaranteed that $\min(R'_T)$ is greater or equal to the start time of T.*

Since the preliminary validity range of an object is always bounded by the current commit time, we know the following:

[1] We do not consider snapshot isolation (SI) here.

Lemma 2. *The preliminary time interval of transaction T at time t (after T has opened the first object) is at most t, i.e.,* $\max(R'_T) \leq t$.

Theorem 1. *LSA guarantees that each read-only transaction T that started at time t_s and that commits between t_c and before $t_c + 1$ is linearizable.*

Proof. T can only commit if its preliminary validity range R'_T is non-empty when T commits. We know from lemma 1 and 2 that R'_T is a subset of $[t_s, t_c]$. This means that T is executed atomically between its start and before T terminates.

Theorem 2. *Each update transaction T that started at time t, that commits at time CT_T, and that satisfy* $\max(R'_T) \geq CT_T - 1$, *is linearizable.*

Proof. On commit, LSA checks that $(CT_T - 1) \in R'_T$ (lines 41–46 in Algorithm 1) and hence that all object versions that T accessed are still valid up to the time CT_T at which T commits its changes. Since each transaction has a unique commit time, no other transaction can commit at CT_T. This means that, logically, T reads all objects and commits all its updates atomically.

3.5 Extensions and Global Time

Validation is the bottleneck of STMs that use invisible reads. Whereas LSA can verify validity for any commit time by trying to extend the validity range to this time, other STMs usually verify the state at the time of validation. One might expect that LSA needs to perform extension frequently when there are concurrent updates that increase commit time fast. However, LSA is quite independent of the speed in which concurrent transactions increase time: if there are no concurrent updates to the objects that a transaction T accesses, the most recent object versions are accessible and do not change. Thus, no extension is required for obtaining a consistent read snapshot. If CT has been increased concurrently and T is an update transaction, one extension from R_T to $CT_T - 1$ is needed. If concurrent updates are not disjoint, LSA will require at most one extension per accessed object. However, this worst case should be very rare in practice because it requires certain update patterns; for example, once an already accessed object gets updated, R_T will be closed and there will be no further extensions.

Furthermore, the number of required accesses to the commit time is small. All transactions must read the current time when they are started. Update transactions must additionally acquire a unique commit time. Further accesses are not required for correctness. For example, if an update transactions needs to access a most recent version, then it can extend to a time at which this version was valid, but this time does not need to be the current time. Time information gathered from the accessed objects can be used instead of the current time.

Although we evaluate LSA only on smaller systems (see Section 4), we believe that the properties previously described as well as other mechanisms, such as using multiple commit times to partition data, make it suitable even for larger systems with higher communication costs.

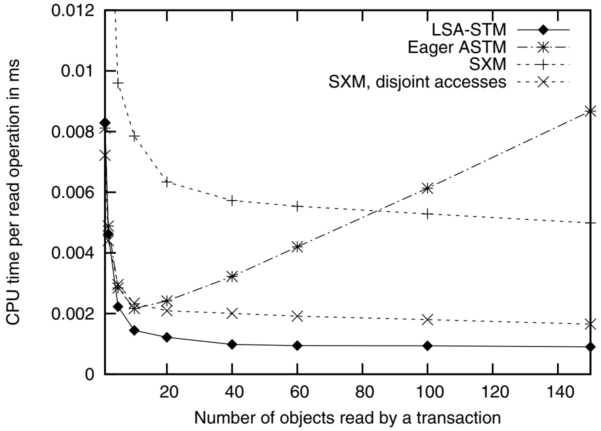


Fig. 1. LSA-STM read overhead in comparison to (Eager) ASTM and SXM

4 Performance Evaluation

To evaluate the performance of our LSA-STM, we compared it with two other implementations. The first one follows the design of SXM by Herlihy *et al.* [17], an object-based STM with visible reads, with a few minor extensions. The second follows the design of Eager ASTM by Marathe *et al.* as described in [13]. Henceforth, we shall call these STM implementations *SXM* and *ASTM*. All three STMs are implemented using Java. Read operations in SXM are visible to other threads, whereas they are invisible in ASTM and LSA-STM. All STM implementations guarantee that all objects read in a transaction always represent a consistent view.

We executed all benchmarks on a system with four Xeon CPUs, hyperthreading enabled (resulting in eight logical CPUs).[2] Results were obtained by executing eight runs of 10 seconds for every tested configuration and computing the 12.5%-trimmed mean, i.e., the mean of the six median values. All STMs use the Karma [11] contention manager.

Overheads of validation and of a global commit time. To highlight the differences between STM designs that use visible and invisible reads, Figure 1 shows the mean CPU time required for reading a single object in read-only transactions of different sizes. In this micro-benchmark, 8 threads read a given number of objects. All transactions read the same objects (with the exception of the SXM benchmark run with disjoint accesses) and there are no concurrent updates to these objects. The fixed overhead of a transaction gets negligible

[2] 8GB of RAM, Sun's Java Virtual Machine version 1.5.0 with default configuration (server-mode, Parallel garbage collector), start and maximum heap size of 1GB. The machine has a light background load that we cannot control. Executing with more than 8 threads can give us a larger percentage of the CPUs and potentially a slight speedup when increasing beyond 8 threads.

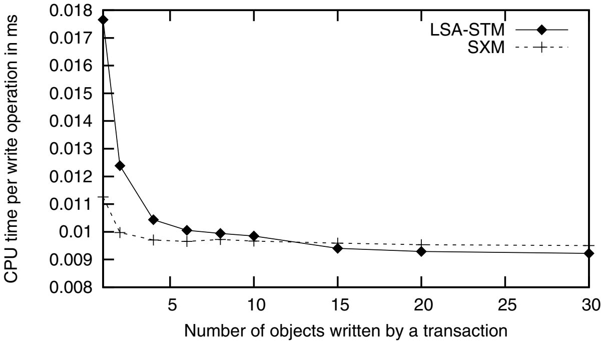


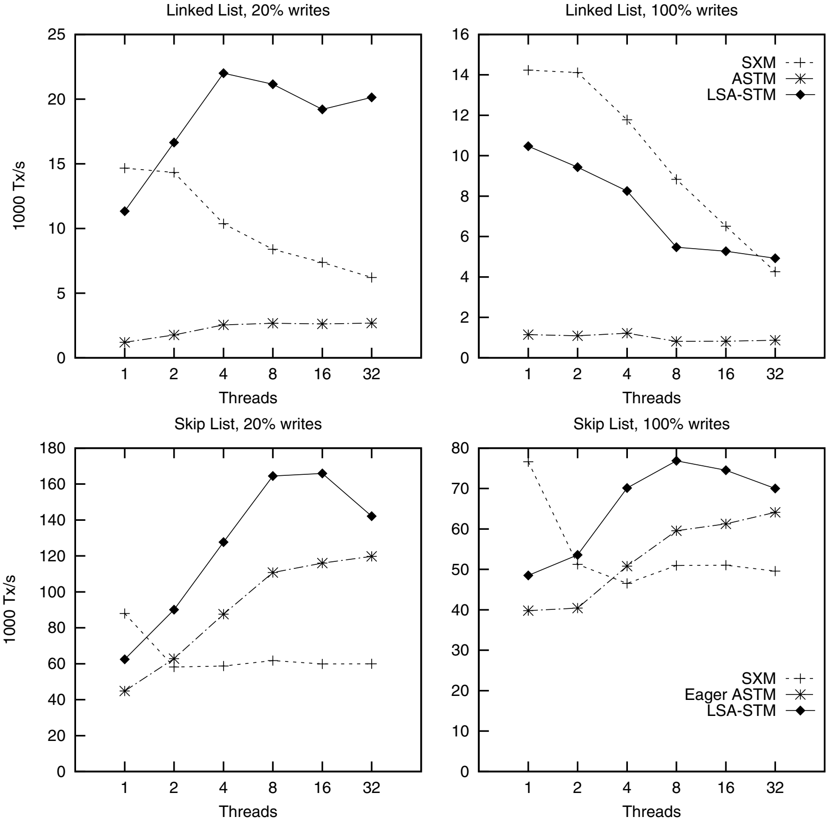
Fig. 2. LSA-STM write overhead

when the number of objects read during the transaction is high. SXM's visible reads have a higher overhead than LSA-STM's invisible reads. This overhead consists of the costs of the compare-and-swap (CAS) operation and possible cache misses and CAS failures if transactions on different CPUs add themselves to the reader list of the same object. ASTM has to guarantee the consistency of reads by validating all objects previously read in the transaction, which increases the overhead of read operations when transactions get large. Note that, although not shown here, ASTM transactions with only a single validate at the end of each transaction perform very similar to LSA-STM. However, for these transactions, consistency is not guaranteed during the execution of the transaction.

LSA-STM currently implements the global commit time *CT* as a shared integer counter. SXM and ASTM do not need such a counter that could become a source of contention if the rate of commits of update transactions is high. Figure 2 shows the overhead of write operations in LSA-STM by means of a microbenchmark similar to the one used for Figure 1. However, now the 8 threads write to disjoint, thread-local objects. Acquiring timestamps induces a small overhead, which, however, gets negligible when at least 10 objects are written by a transaction. Furthermore, the overhead is smaller than the costs of a single write operation. Note, of course, that the results in Figure 1 and Figure 2 are hardware-specific.

Throughput. Figure 3 shows throughput results for two micro-benchmarks that are often used to evaluate STM implementations, namely integer sets implemented via sorted linked lists and skip lists. Each benchmark consists of read transactions, which determine whether an element is in the set, and update transactions, which either add or remove an element. The sets consist of approximately 250 elements during the benchmark runs. We do not release objects early: although this would decreases the possibility of conflicts, it can mainly be used in cases in which the access path to an object is known.

We use the linked list to conveniently model transactions in which a modification depends on a larger amount of data that might be modified by other transactions.


Linked List, 20% writes

Linked List, 100% writes

Skip List, 20% writes

Skip List, 100% writes

Fig. 3. Throughput results for the linked list (top) and skip list (bottom) benchmarks

For the skip list, STMs using invisible reads (ASTM and LSA-STM) show good scalability and outperform SXM, which suffers from the contention on the reader lists. However, the transactions in the linked list benchmark are quite large (the integer sets contain 250 elements) and ASTMs validation is expensive. LSA-STM, on the contrary, uses version information to compute the validity range much faster and scales up well to the number of available CPUs.

In all previous benchmarks, we always configured LSA-STM to keep eight old versions per object besides the most recent committed version. Keeping several versions can typically increase the commit rate but also adds memory overhead. In the following, we examine this problem further.

Object versions accessed. In LSA-STM, references to object versions are stored in both a "locator" structure associated with transactional objects and an extra version array referenced by the locator. Like SXM and ASTM, LSA-STM is an object-based STM based on the design of DSTM [9] and thus uses locators to manage two object versions. However, whereas the other STMs use one of these versions as the working copy modified by updating transactions and the other version as a backup copy, LSA-STM can—because of LSA and validity range information—let reading transactions efficiently access the backup copy when an update is happening (it is the the most recent version) and when the working copy is committed (then the backup copy is the most recent old version). Thus, LSA-STM can provide one or two consistent versions of the object with the same space overhead. In the following, we denote accesses to the two versions (primary and backup) managed by the locator as accesses to version 0 or version 1, respectively. The extra version array stores references to older versions (the most recent version in the array has number 2).

Which object versions are accessed by a read-only transaction depends on how objects are concurrently updated by other transactions. To investigate this, we use a simple bank micro-benchmark, which consists of two transaction types: (1) transfers, i.e., a withdrawal from one account followed by a deposit on another account, and (2) computation of the aggregate balance of all accounts. Whereas the former transaction is small and contains 2 read/write accesses, the latter is a long transaction consisting only of read accesses (one per account and always in account order). There are 1,000 accounts and 8 threads perform either transfers or, with a 10% probability, balance computations.

Figure 4 shows access histograms of transactions computing the aggregate balance. There are three benchmark modes: (1) no hotspots, that is, update probability is equal for all accounts, (2) hotspots are encountered early during aggregate-balance computation, and (3) late hotspots. Hotspots are modeled by making the probability of updates to the first or last 50 accounts (accessed early or late, respectively) as probable as updates to other accounts.

In Figure 4, we can see how different update frequencies affect LSA's version selection (note the logarithmic scale). First, we observe that most accesses are performed to recent versions. When there are no hotspots, eight old versions are sufficient. When hotspots are encountered early during the runtime of a transaction, subsequent accesses will use even more recent object versions, because the relative update frequency of objects accessed late is smaller. In contrast, if hotspots are encountered late, the transaction has to use older versions if one of

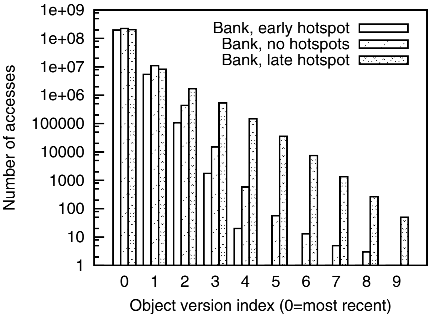


Fig. 4. Object versions accessed by balance computation transactions

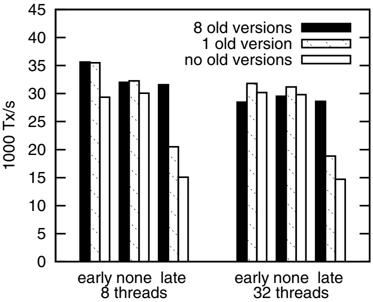


Fig. 5. Throughput results: versions kept per object, Bank with early, no, or late hotspots

the objects accessed early has been updated, which lets the validity range become closed. Thus, the probability that an old version will be useful increases with the size of the transactions and when hotspots happen late in their execution.

Throughput when keeping fewer old versions. Figure 5 shows the throughput of the bank application when LSA-STM is configured to keep an extra version array with eight or one version, or no extra versions at all besides the versions 0 and 1 in the locator.

We can observe that keeping old versions can be beneficial, especially if hotspots are encountered late. However, the figure shows as well that throughput can increase if there are less versions. We will address this problem in future work.

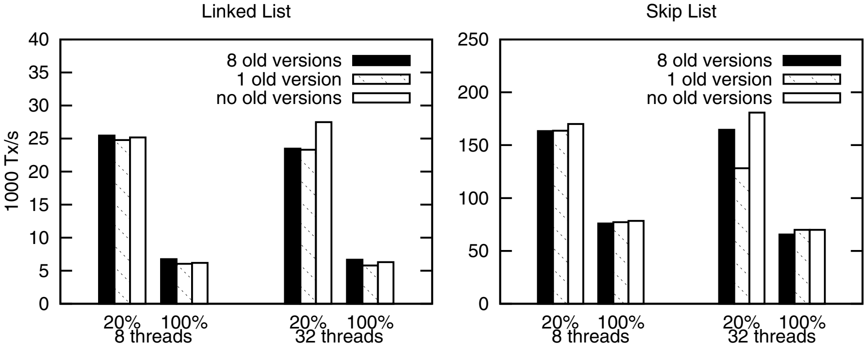


Fig. 6. Throughput results: versions kept per object, Integer Sets with 20% and 100% updates

Figure 6 shows throughput results for the linked list and skip list benchmarks, with 20% or 100% update transactions and 8 or 32 threads. The skip list is mostly unaffected by the number of old versions available, or benefits only slightly from old versions. On the contrary, fewer versions increase the throughput of the linked list significantly. Using old versions will close the validity range, which is disadvantageous for transactions that become update transactions after reading a lot of objects (we assume that the type of a transaction is not known *a priori*). Nevertheless, we have focused our study on eight extra versions because this solution adapts well to various workloads. Adaptive version selection strategies might be able to increase the throughput further.

Even without any extra versions, and thus with the same space overhead as SXM or ASTM, LSA-STM is able to provide high throughput thanks to LSA and up to two consistent versions being available for reading transactions.

Validity range extensions. The number of validity range extensions is very small in all of our benchmarks (not shown because of space limitations). The vast majority of transactions uses less than two to four extensions, depending on the transactions. In general, it can be observed that committed read-only transaction mostly use no or a single extension, whereas aborted read-only transactions often use at least one extension but seldom more. This is not surprising because high numbers of extensions can essentially be caused by scenarios in which (1) the update frequencies of objects accessed late during the transactions runtime are higher than those of objects accessed earlier, or (2) updates always happen immediately prior to accesses. For example, a single transition from non-hotspot to hotspot access patterns, as takes place in the benchmark, does not cause a lot of extensions. In the bank benchmark, read-only transactions almost never use extensions. In the linked list benchmark, however, almost all aborted read-only transactions use a single extension.

Update transactions behave as expected: the number of extensions for obtaining a snapshot is similar to that of read-only transaction, plus at most one extension per object update and at most one per commit. For example, less than 1% of the transfer transactions require an extension at all.

5 Conclusion

We introduced a lazy snapshot algorithm that creates consistent snapshots on the fly and can be used by STMs for read-only and update transactions. It is efficient both theoretically and practically. The idea is to maintain, for each transaction, a validity range based on global time. This range is sufficient to decide if a snapshot is consistent and transactions using this snapshot are linearizable. The snapshots are created in such a way that their freshness is maximized, e.g., a snapshot might actually become valid at a time after the snapshot is started. One issue is that the validity range of some of the objects is not known at the time the snapshot is created. This might require, in some cases, an *extension* of the preliminary validity range.

Acknowledgements. We thank the anonymous reviewers for their comments.

References

1. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC. (1995)
2. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. (2006)
3. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation. (2006)
4. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT. (2006)
5. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3) (1990) 463–492
6. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: TRANSACT06. (2006)

7. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of SIGMOD. (1995) 1–10
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Proceedings of OOPSLA. (2003) 388–402
9. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.: Software transactional memory for dynamic-sized data structures. In: Proceedings of PODC. (2003)
10. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems. (2003)
11. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs. (2004)
12. Scherer III, W., Scott, M.: Advanced contention management for dynamic software transactional memory. In: Proceedings of PODC. (2005)
13. Marathe, V.J., III, W.N.S., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of DISC. (2005) 354–368
14. Cole, C., Herlihy, M.: Snapshots and software transactional memory. Science of Computer Programming (2005)
15. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: 20th International Symposium on Distributed Computing (DISC). (2006)
16. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. In: Proceedings of SCOOL. (2005)
17. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of DISC. (2005)

Bounded Wait-Free *f*-Resilient Atomic Byzantine Data Storage Systems for an Unbounded Number of Clients
(Extended Abstract)

Rida A. Bazzi and Yin Ding

Computer Science Department
Arizona State University
Tempe, Arizona, 85287
{bazzi, yding}@asu.edu

Abstract. We present the first direct bounded wait-free implementation of a replicated register with atomic semantics in a system with an unbounded number of clients and in which up to f servers are subject to Byzantine failures. In a system with $n \geq 4f + i + 1$ servers, and in the presence of a single writer, our implementation requires 5 messages from the reader and at most $6 + 2(f - i)$ messages per correct server per read operation and 2 request and 2 reply messages per server for each write operation. Unlike previous solutions, the number of messages is independent of the number of write operations that are concurrent with a read operation. For the case of multiple writers, a read operation requires 5 messages for the reader and no more than $6 + 2c(f - i)$ reply messages per correct server, where c is the number of writers that execute concurrently with the read operations, and a write operation requires 4 request and 4 reply messages per server. The message requirements of our wait-free implementations are considerably better in the worst case than those of the best known non wait-free implementations. If there is a bound on the number of writers, the total number of messages sent by a server is linear in the number of read operations, so faulty servers that send too many messages will be detected as faulty. This implementation does not rule out the possibility that a reader receives and discards many delayed messages in a read operation, so it is bounded only in an amortized sense. We describe a bounded solution in which a read operation will not receive more than a constant number of messages from a server without detecting the failure of the server. No other solution is bounded – in an amortized sense or otherwise.

Keywords: Atomic, Byzantine, Fault Tolerance, Replication, Timestamps, Wait-Free.

1 Introduction

We study the problem of implementing a replicated data store in an asynchronous system in which f out of n servers can be faulty. We consider a system

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 299–313, 2006.

with an *unbounded number of clients* in which servers are subject to Byzantine failures and the data stored at servers is not self-verifying[1]. We aim at providing an implementation that tolerates any number of client crash failures (wait-free with respect to client failures) and up to f server failures (f-resilient).

Distributed implementations of shared registers in systems in which servers are subject to Byzantine failures have been proposed by a number of researchers [4, 9, 10]. In such systems it is straightforward to provide implementations of safe registers [8]. To provide stronger semantics, classical implementations of atomic registers using registers with safe semantics can be used ([7] for example). Such implementations are not appropriate for distributed shared storage systems because they have high latency and they assume an upper bound on the numbers of readers and writers and therefore would require resources to accommodate the maximum number of clients in the system. We are interested in solving the problem in a system with an unbounded number of clients so that the used resources at any given time are a function of the actual number of clients in the system. Martin et al. [10] were the first to present a solution that provides atomic semantics for non self-verifying data in our model. Their solution uses a technique in which servers "forward" to every reader with an ongoing read operation *all* late *writes* until the reader decides on a value to read. Using a similar forwarding mechanism, we gave a solution that improved on the solution of Martin et al. in the readers' memory requirements [4]. This improvement was achieved by introducing, and providing an implementation of, non-skipping timestamps whose size grows logarithmically in the number of operations, in contrast to timestamps used by other solutions whose size grow arbitrarily.

To tolerate client crashes, Martin et al.'s and our previous solution use a diffusion mechanism: a server does not process a write request until it forwards the request to all other servers and receives acknowledgements from $n-f$ servers. In addition to the high message complexity of such a solution, $\Omega(n^2)$, it cannot be used to solve the problem in a model in which servers do not communicate with each other, but rather act as shared objects [5]. Our previous solution as well as that of Martin et al. can require an unbounded number of messages to be sent to reader whose read operation is concurrent with many write operations. A natural question to ask is the following: can the number of messages exchanged be made dependent only on the number of writers and not on the number of write operations? It turns out that it is relatively easy to come with a wait-free solution that does not use message diffusion and that requires only a constant number of messages per read operation, but that solution requires unbounded storage at the servers. So, the more interesting question to answer is the following: is there a solution in which the number of messages exchanged and the storage at each server in a given time period is only dependent on the number of clients and not on the number of operations during that period? We call such a solution, if it exists, a bounded solution. In this paper we first show that such if $n > 4f + i$, where $i \geq 0$ is a system parameter that affects the solution's efficiency, then

[1] In other words, no cryptographic techniques are used to verify that the data stored at a server is authentic and not made-up by the server.

an amortized bounded solution exists. If c is the number of *distinct writers* concurrent with a read operation and r is the number of read operations executed so far by the reader, then, at any given time, if the reader has received from a given server a cumulative number of messages that is larger than $W \times r$, where W is a constant that we define later, then the reader can detect that the server is faulty and ignore all further messages from the server. The solution does not rule out the possibility that many of the messages sent by a server will be handled by one read operation if these messages take a long time in transit. So, the solution is bounded only in an amortized sense. Still, it has significantly better message complexity than previous solutions.

We show how the amortized bounded solution can be made into a bounded solution in which a given read or write *operation* will not handle more than a constant number W messages from a given server without detecting that the server is faulty. If the number of writers is finite, then a read operation will always handle a constant number of messages.

An interesting feature of both solutions is that messages that are sent to readers use timestamps whose size is logarithmic in the number of operations in the system; in practice their size can be bounded by a large constant. Since the number of messages sent by a server for a given read operation is constant and the number of messages handled by a read operation is also constant, it follows that our solution requires only (almost) constant size buffers at both readers and servers. Unfortunately, both solutions allow some messages sent by servers to writers to be unbounded in size. Still, it is interesting that messages sent to readers are almost bounded even though the solution is atomic. In fact, readers have to writeback values to ensure atomicity and we introduce a new way to handle writebacks by readers to avoid having the readers acting like an unbounded number of writers. Our bounded solution is a significant improvement over previously proposed solutions because the number of messages sent by correct servers and the number of messages *handled* by read operations are constant if the number of writers in the system are constant and if faulty servers do not want to be detected as faulty[2]. No other solution is comparable in this regards. Even our amortized bounded solution improves greatly on the solution of Martin et al. [10] and our previous solution [4]. For the case of multiple writers the amortized solution achieves:

- For a read operation, 5 messages are sent by the reader to each server, and $6 + 2c(f - i)$ messages are sent by each correct server (c is define above), whereas in previous work [4, 10] the number of messages sent by a correct server is equal to 1 plus the number of *write operations* concurrent with a read operation.
- The solution is wait-free without requiring servers to communicate with each other. Previous solutions required $\Omega(n^2)$ messages to be exchanged by servers per write request to tolerate client crashes.

[2] If the faulty servers do not mind being detected as faulty, there is nothing that prevents them from sending any number of messages to a reader.

Our amortized bounded solution requires $n \geq 4f + i + 1$, where i is a systems parameter that affects the number of messages sent by servers (see above). In our solution each server keeps only three copies of the data (two most recent values and one *writeback* value), which is almost optimal. Our solution, as well as previous solutions, requires each server to maintain a list of active readers and therefore, it requires space that is bounded by the number of active readers in any given time period, but practically that space will be small because only reader identifiers and not data values are maintained and that space might be required anyway in a practical setting if connections are established between readers and servers. In addition, our solution requires the server to maintain for a reader a finite list of timestamps, but since the timestamps in our solution are non-skipping, their space requirement is practically constant as we explained above. Finally, all solutions we are aware of always require space that is proportional to the number of readers or active readers in the system.

2 Related Work and Contributions

Researchers studied the problem of implementing shared registers in a variety of models, including the shared object model [2] and the distributed storage (with active servers) model. Earlier work in the shared object model considered benign and arbitrary failures and distinguished between responsive and non-responsive failures [6]. An object that is subject to responsive failures always responds, so it is possible to detect object crashes and in this model Byzantine objects have to always respond or their failure will be detected. In this paper we consider non-responsive failures. This is the harder model to handle and it is the model considered in Martin et al.'s work as well as our previous work and more recent related work [1]. Jayanti et al. [6] provided wait-free implementations of safe registers using shared registers that are subject to non-responsive arbitrary failures, but they did not provide direct implementations of registers with stronger semantics.

In message passing systems, Attiya et al. [3] seem to be the first to consider the implementations of shared memory primitives (in the presence of benign failures). In the presence of Byzantine failures, Byzantine quorum systems are typically used for replicated register implementation [8]. To write a value, a writer sends the value to all servers in a quorum set, and, to read a value, a reader collects data from all elements of a quorum set that has a large enough intersection with the write so that the reader can get the most up-to-date value even in the presence of Byzantine servers in the intersection. The simple implementation provides safe semantics [8] and stronger semantics can be enforced using classical results [7].

Martin et al. [10] were the first to present a solution that provides atomic semantics for non self-verifying data in an asynchronous system with an unbounded number of readers and writers, and in which servers are subject to Byzantine failures and in which servers can: do a number of operations atomically; asynchronously forward messages to clients; and store data. Their solution uses message diffusion to tolerate client failures and cannot tolerate client failures if servers cannot communicate. Our previous solution is for the same system model [4]. Our server model is similar to active disks which were earlier considered by Chockler and Malkhi [5] for providing solutions to the consensus problem in the presence of non-responsive crash failures. The difference is that we allow servers to asynchronously forward messages to clients. In practice, active servers should be able to asynchronously forward messages to clients.

More recently, researchers considered f-resilient implementations of shared registers using only read/write registers as base objects. The resulting solutions are either not wait-free or not atomic and they do not support multiple writers. Abraham et al. [2] present a wait-free single writer multiple readers (SWMR) safe register that works for $n = 3f + 1$ (which is optimal), and a regular SWMR register in which the read operations are guaranteed to terminate only when they overlap with a finite number of write operations (FW-termination). Their work uses read/write register objects subject to Byzantine failures. To our knowledge, no previous direct implementation of wait-free registers in the shared object model satisfies atomic semantics. The best direct wait-free implementation provides a single-writer/single-reader register with regular semantics, but it only uses the weaker read/write objects and not the powerful servers that we use [1]. That work seems to be the first work that leverages classical techniques for wait-free implementations of shared registers [11] to directly implement shared register with stronger than safe semantics in the presence of Byzantine failures.

The techniques we use in this paper for the single writer case are only in some aspects similar to those in [1, 11] and there are major differences between the techniques we use and their techniques, especially that we support multiple writers and provide atomic semantics. For example, in [1], the register keeps the two most up to date written values and the writer alternates in writing to the one or the other copy and a reader will read a value that is equal to the old value in a sufficient number of register or equal to the new value in a sufficient number of registers. In our implementation, we also use two most up to date value (which was also used in the early work of Peterson [11]), but we also need the servers to keep a separate *writeback* value. The value read is only required to appear in a sufficient number of servers: it can be old in some servers and new in some other servers or it can be a writeback value still in some other servers.

More importantly, we use a novel technique to execute writebacks. In all other work that we are aware of, the writeback value replaces existing old value. In our solution, we keep a separate writeback value and have a new *writeback throttling* technique in which the writeback operation is executed in two phases to ensure that the separate writeback value does not get way ahead of values directly written by a writer. In a first phase, a reader will send a writeback request, and, only after its request is acknowledged, it will send the actual value being written back. Instead of writing back a value by sending an explicit writeback message, we could have required the reader to do a full write operation at the end of reading. This would even make the protocol less complicated and the proof of atomicity will also be simpler, but this approach has the disadvantage of exposing the reader to messages that are potentially unbounded in size because the messages sent to writers are not bounded. Our solution has to somehow hook the writeback on a write message without exposing the reader to write messages from servers. This is achieved through the *writeback throttling* technique.

To ensure atomicity, we require the reader to read twice before deciding on potential timestamps for a value to read. Reading twice is used in the literature, but this is the first time it is used in this context.

Furthermore, to ensure termination and boundedness, we use a new *timestamp requesting* approach in which the reader determines a finite number of timestamps as being the only possible timestamps for a value to be read. This is coupled with functionality that allow a writer to detect ongoing read operations. This functionality will guarantee that the reader will read a value with one of the determined timestamps or it will be detected by the writer in which case it can get identical values forwarded directly to it by the write operation. We manage to use those techniques and other techniques to achieve the first bounded wait-free implementation that does not use message diffusion.

3 System Model

The system consists of two sets of processes: a set of n server processes (servers) and an unbounded set of client processes (clients). Clients have unique identifiers that are completely ordered.

Clients communicate with servers and servers communicate with clients and with other servers through reliable FIFO message passing.[3] To send a message, a process uses the *send* function that takes an intended recipient and a message *content* as parameters. We identify a sent message as a triplet (s, r, c), where s is the sender, r is the recipient and c is the content of the message. Every message sent is guaranteed to be delivered to the intended recipient at a later time, but there is no bound on the time elapsed between the time a message is sent and the time it is delivered. A message delivered to p is of the form (s, c), where (s, p, c) is a message sent at an earlier time. To receive messages, a process p uses the *receive* function which returns a message that has been delivered to p and that p has not previously received.

Up to f of the server processes can be faulty and $n \geq 4f + i + 1$ for some $f \geq i \geq 0$. Faulty server processes exhibit arbitrary failure behavior: they can send arbitrary messages and change state arbitrarily.

Client processes can invoke a "read" or a "write" protocol which specifies the state changes and the messages to send to servers to initiate a "read" or a "write" as well as the state changes and messages to send in response to server messages. A *reader* is a client process that executes a read protocol. A *writer* is a client process that executes a write protocol. When we consider faulty clients, we assume that they fail by crashing. When a client fails, it stops sending messages.

[3] Message passing is assumed to be FIFO to simplify the exposition. All our results can be modified using standard techniques to work for systems with non-FIFO message passing.

4 Single Writer Amortized Bounded Solution

4.1 Overview

Servers store the two most up to date values they are aware of (v_{cur} and v_{pre}) as well as a separate *writeback* value (v_{wb}). A reader requests values from servers and attempts to find a value that appears in enough responses. Since the protocol maintains only a finite number of values, handling a read operation that is concurrent with multiple write operations requires special care. In particular, if old values are replaced by new values at different servers, it is possible that the reader will not be able to find a value that was written by a particular write operation because each server might have a value written by a different write operation. To protect against this possibility, a write operation tries to detect concurrent read operations. For such detected operations, the writer will act "on behalf" of the readers and instructs the servers to forward to the detected readers the value being written. This forwarding can occur before the server receives the direct read requests from the readers. This forwarding will ensure that the reader will be able to get a snapshot of the server values that guarantees the reader to be able to find a value that was stored at a sufficient number of servers and that is equal to one of the recently written values. Still, it is possible that a writer does not detect (or *see*) the concurrent read operation, so another mechanism is needed to ensure termination at all times.

In our solution, the reader determines a finite set of timestamps of values that it needs for termination and requests that servers forward values with these timestamps whenever they receive them. We ensure that the reader will get enough values with identical timestamps equal to one of the timestamps determined by the reader or, if that does not happen, the writer must have written too many values and will *see* the pending read operation in which case it will request that servers forward one value completely to enable it to terminate. In fact, in order for the writer to see the reader, it is enough for the writer to completely write a value with timestamp $t_{largest}+1$, where $t_{largest}$ is the largest timestamp that the reader receives from a correct server in reply to its second read request phase and that is not one of the i largest timestamps that the reader receives in reply to its second read request (in our solution, a reader issues two read requests and collects two sets of values to ensure atomicity; see description below). If the writer does not write a value whose timestamp is $t_{largest}+1$, then the values with timestamp $t_{largest}$ or the value with timestamp $t_{largest}-1$ is completely written and is not overwritten by a value with higher timestamp. Unfortunately, the reader does not know the value of $t_{largest}$, otherwise it could request a value whose timestamp is $t_{largest}$ or $t_{largest}-1$. To guarantee that it requests $t_{largest}$, the reader request the $(f+1)$'th largest to $(i+1)$'th largest ts_{cur} timestamps that the reader receives in reply to its second read request, and the timestamps equal to these timestamps minus one. (Note: from $(f+1)$'th largest to $(i+1)$'th largest ts_{cur} may not be the consecutive values. Also, the larger the value of i the less values will be requested; in fact, if $i=f$, only 2 values will be requested) So the number of the potential timestamps is at most $2(f-i+1)$, which is finite.

Two interesting aspects of our protocol relate to how atomicity is ensured. In an atomic implementation, we would like a read operation that starts after another read operation terminates to return a value that is at least as recent as the value returned by the older read operation (read-read atomicity). Also, we would like that a read operation that starts after a write operation terminates to return a value that is at least as recent as the value written by the write operation (read-write atomicity).

The standard technique to provide read-read atomicity is to have a reader write back the value that it reads. Since we are only maintaining a finite number of values at a server (in particular, only one writeback value), we would like to ensure that a later read operation will receive enough identical values that are at least as recent as the value returned by the older read. If there are multiple writebacks concurrent with one read operation, it is possible that the older (and more relevant) writeback value is erased. One possible approach would have later writebacks check for concurrent readers, but this will complicate the solution and will expose the readers to potentially unbounded messages from faulty servers as explained in Section 2. To get around this difficulty, we propose a novel *writeback throttling* mechanism that prevents writebacks from getting much ahead of writes. The idea is to divide a writeback into two phases. In a first phase, a reader sends a writeback request specifying the timestamp of the value it wants to write back. A server will grant the writeback request only after it receives the data value that immediately precedes the writeback value; the immediately preceding value has a timestamp that is equal to the writeback timestamp minus one (timestamps are non-skipping). This will ensure that the writeback value can only be *one step ahead* of directly written values and makes our solution possible.

To provide read-write atomicity, we require the reader to collect two sets of data, in effect reading twice, before deciding on a value to *read*. Reading twice is needed to ensure that either the reader will decide to read a value whose timestamp is larger than that of the latest written value or that the latest written value did not get overwritten many times between the two reads, in which case the reader can read the latest written value.

So, in summary of the main techniques in this solution, we are using the detection of concurrent readers, forwarding, and requesting for potential timestamps to ensure termination, and, we are using *writeback throttling* and repeated reads to ensure read-read and read-write atomicity respectively. We should note here that the forwarding we are using is different from that we used in the past in that it is writer-initiated and not server-initiated. Also, the requesting of potential timestamps is new for this solution. In the detailed description that follows we also describe how readers detect faulty servers that send too many messages.

4.2 Client Side

We assume that a writer will not write a new value until it finishes writing the previous value. Each read operation has an id (cur_opID) which is incremented before each read operation. The id is used to ensure that the cumulative number of messages received by a reader from a given server is less than $2(f-i)+6$ (the maximum number of messages per read operation) times the number of read operations. If this total is exceeded, the server is added to a *faulty* set (initially empty) and its messages are rejected. This is enforced in the **receive** function (Figure 2) which wraps the system's *receive function.* Note that we could have made the detection of failures tighter, but we want to keep the description simple. Each message is tagged with the id of the operation to which it pertains and messages that do not pertain to the current operation are ignored.

Read operation. A reader starts by sending a READ REQUEST I message to all servers and waits for replies from $n - f$ of them (Figure 1, lines 1-6). A reply of a server consists of a triplet of pairs of the form (v, t), where v is a value and t is a timestamp. These three pairs are the two most up-to-date values that the server has received directly from a writer and a writeback value that the server receives from readers at the end of read operations. Then, the reader starts another round of read by sending a READ REQUEST II message and waits for another $n - f$ triplets from different servers. The replies received in the two rounds are kept in two *received* arrays that can contain up to n entries, one for each server, and whose entries are triplets of values. Any forwarded messages received by the reader during the two phases is kept in a *forwarded* array. (lines 7-14). The reader sorts the timestamps received in the two rounds and chooses two timestamps as the potential target timestamps: (1) $ts_{t_1} = f + 1 + i$'th largest current timestamp value (ts_{cur}) received, and (2) $ts_{t_1} - 1$. (lines 15-16) Then the reader tries to decide a value with a timestamp ts_{t_1}, and, if that is not successful, it tries to decide on a value with timestamp $ts_{t_1} - 1$. A reader decides a value (using the *decide* function) if there at least $f + 1$ replies *by different servers* that contain the same value with the timestamp specified as an input to the *decide* function. (lines 17-20) The other parameters of the *decide* function specify the entries that can be used in looking for identical values. For the case of the *received* arrays, these identical values can appear in any pair in the triplet. If the reader can decide a value, it writes back the value read and finishes the operation (see below). If there are not enought identical replies for the reader to decide, the reader determine a set of POTENTIAL Target timestamps (line 21) and requests that servers send values whose timestamps are equal to one of the potential target timestamps or to one of the potential target timestamps minus 1 (the request for potential target timestamps -1 is implicit in the reader's code but is enforced in the server code). The smallest explicitly requested potential target timestamp is ts_{t1} and the smallest implicitly requested potential target timestamp is $ts_{t_1} - 1$. The data value that the reader will end up *reading* will have a timestamp that is not less than $ts_{t_1} - 1$. This is enforced in lines 23-40 (lines 30,32 and 35 in particular). The reader waits for replies from the servers until it can *read* a value.

After the reader *reads* a value, it has to ensure that later readers will not read older values. This is achieved by writing back the value read. As explained in the protocol overview, we use a novel *writeback throttling* mechanism when writing back values. The writeback operation proceeds in two phases. In the first phase, the reader sends WRITEBACK REQUEST messages to all servers indicating the timestamp ts_{target} of the value it wishes to writeback (line 41). The WRITEBACK REQUEST will be acknowledged by a server only if the server has a $ts_{cur} = ts_{target} - 1$. This ensures that no server will have a writeback value that is way ahead of a directly written value. This property is essential for ensuring atomicity. When a reader receives a WRITEBACK REQUEST ACK from a server, it adds the server to the set $S_{req_ack}$of servers that acknowledged the writeback request and it sends to the server a WRITEBACK MESSAGE containing the writeback value (lines 42-48). The reader waits until it receives $n - f$ acknowledgments to its WRITEBACK messages at which time it sends the WRITEBACK messages to the rest servers (lines 49-50). After the reader receives *(n-f)* acknowledgments, it terminates(lines 52-53).

Write operation. The writer starts by sending a WRITE message to all servers and waits until it receives $n - f$ replies (Figure 3, lines 1-6). The goal of this request is not only to write a new value, but to detect readers that have already started their second read requests, but not finished reading - the set of concurrent readers. In response to a WRITE message, the writer receives from each correct server s a reply that contains the set R_{concur} of readers that have started their second read requests, but whose *read* operations have not terminated (no writeback request message received by s). The writer can tell that the second read request phase of a reader r is concurrent with the write operation if $f + 1$ servers reply with messages indicating that they received a second read request from r (line 7). To each server, the writer sends a FORWARD message specifying the set of concurrent readers, R_{concur}, that the writer calculated acknowledgments in line 7 and from which the server might not have received read requests (line 8). We assume that whenever the writer wants to write a new value, it increments by 1 the value of the timestamp it used in its last write operation. After the writer receives $n - f$ acknowledgments, it terminates. We omit from the writer code functionality to bound the total number of messages and detect failures; it is similar to that of the reader.

4.3 Server Side

Reader messages. Upon receipt of a READ REQUEST I message, the server sends the reader the most up-to-date values that the server has (Figure 4, lines 1-2). Upon receipt of a READ REQUEST II message, the server adds the reader to the set G_{send2} of readers from whom it received second read requests and sends the reader the most up-to-date values that the server has (lines 3-5).

Upon receipt of a WRITEBACK REQUEST message, the server first removes the reader from the set of G_{fwd} and G_{sent2}(lines 6-8), then checks if the proposed writeback timestamp is not very large (line 9). If it is very large, the server adds the reader to the set of readers with pending writebacks (line 10). If the proposed timestamp value is not very large, the server acknowledges the writeback request, thereby allowing the reader to send the actual writeback value (lines 11-12). A reader that is added to the set of readers with pending writebacks will be acknowledged once the value preceding the value being written back is received directly from the writer. The preceding value is guaranteed to eventually arrive because a writer does not start writing a value before finishing writing a previous value and timestamps are non-skipping.

Upon receipt of a WRITEBACK message, the server updates the most up-to-date writeback value (v_{wb}) and corresponding timestamp if the received value is more up-to-date than v_{wb}, then it removes the reader from the set G_{send2} sends an acknowledgement to the reader (lines 13-16)

Upon receipt of a REQUEST TARGET message, the server checks whether it has a value whose timestamp is requested by the reader explicitly (lines 18-22) or implicitly (lines 23-27). If it has one or more of the requested values, it sends those values to the reader. The server will keeps the list of the timestamps requested by the reader in the request pending set, $Req_{pending}$ (line 28).

Writer messages. Upon receipt of a WRITE message, the server updates its values (v_{cur} and v_{pre}, but not v_{wb}) and timestamps if the received value is newer than the most up-to-date stored value (lines 30-32) (Since communication is FIFO, there is really no need to check that the written value is more up-to-date than the stored values because every new value written by the writer will me more up-to-date than v_{cur} and v_{pre}. This will not be the case for the multiple writer solution). Then, it will check whether the new data is one of pending reader requesting, and send to it if so (line 33-39). Next, the server sends the list of readers in the set G_{send2}) but not in G_{fwd}; these are the readers have started their second read request phase, but from which the server has not yet received writeback messages and not forwarded a value to them (line 40). Finally, the server checks if there are pending writebacks of a value that immediately follow the value just received. If there are such pending writebacks the server acknowledges them (lines 41-43). It is important to note here that checking that a value immediately follows another value is straightforward because timestamps are non-skipping and successive value have timestamps that differ exeactly by 1.

Upon receipt of a FORWARD message, for every reader in the set R_{concur} but not G_{fwd} whose second phase is concurrent with the writer and to which the writer has not previously forwarded a value, the server sends (v_{cur}, ts_{cur}) to the reader (lines 45-46). Then, the server adds the set of concurrent readers to the sets G_{fwd} and sends an acknowledgement to the writer (lines 47-48).

Due to space limitations, we state without proof the following Theorem.

Theorem 1. *The proposed solution provides a wait-free atomic implementation of a distributed register in the presence of up to f Byzantine faulty servers. At any given time, the* average *number of messages received by a reader from a server that is not detected to be faulty is less than or equal to* $6 + 2(f - i)$ *per read operation.*

Next we describe at a high level the bounded solution for the single writer case.

5 Bounded Solution

In the amortized bounded solution, a client always sends its requests to all servers even if the servers did not reply to previous requests. This can result in a large

```
Initially: $faulty = \emptyset$, $cur\_opID = 0$, $rpl\_num[i] = fwd\_numID[i] = 0$, $1 \le i \le n$.

read(out: ts,value)
$cur\_opID = cur\_opID + 1$

:1st Read Request:
send ($cur\_opID$, READ REQUEST I) to all servers
$received1[i] =$ null, $1 \le i \le n$
repeat
    $(s, opID, (tag, [(v\ \ , t\ \ ), (v\ \ , t\ \ ), (v\ \ , t\ \ )])) =$ receive()
    if $(tag = r1)$and$(opID = cur\_opID)$ then
        $received1[s] = [(v\ \ , t\ \ ), (v\ \ , t\ \ ), (v\ \ , t\ \ )]$
until $|\{i : received1[i] \neq$ null$\}| \ge n - f$

:2nd Read Request:
send ($cur\_opID$, READ REQUEST II) to all servers
$received2[i] =$ null, $1 \le i \le n$
$forwarded[i] =$ null, $1 \le i \le n$
repeat
    $(s, opID, (tag, data))$=receive()
    if $(tag = r2)$and$(opID = cur\_opID)$ then $received2[s] = data$
    elseif $(tag = fwd)$and$(opID = cur\_opID)$ then $forwarded[s] = data$
until $|\{i : received2[i] \neq$ null$\}| \ge n - f$
$sorted\_rcvd2[] = received2[]$ sorted in descending order according to $t$
$ts_{\ 1} = sorted\_rcvd2[f + i + 1]$

:1st Try to Decide:
$value = decide(ts_{\ 1}, \{received1, received2\})$
if $value \neq$ null then WriteBack$(value, ts_{\ 1})$
$value = decide(ts_{\ 1} - 1, \{received1, received2\})$
if $value \neq$ null then WriteBack$(value, ts_{\ 1} - 1)$

: Request Potential Target Timestamps:
$Targets = [sorted\_rcvd2[i + 1].t\ \ , \ldots, sorted\_rcvd2[f + 1].t\ \ ]$
send ($cur\_opID$, POTENTIAL TARGETS, $Targets$) to all servers

: 2nd Try to Decide:
$target\_rcvd[j][s] =$ null  $i + 1 \le j \le f + 1$, $1 \le s \le n$
$target\_rcvd_{-1}[j][s] =$ null  $i + 1 \le j \le f + 1$, $1 \le s \le n$
$value =$ null
repeat
    $value = decide(ts_{\ 1}, forwarded)$
    if $value \neq$ null then WriteBack$(value, ts_{\ 1})$
    for $j = i + 1$ to f+1 do
        $value = decide(sorted\_rcvd2[j].t\ \ , target\_rcvd[j])$
        if $value \neq$ null then WriteBack$(value, sorted\_rcvd2[j].t\ \ )$
        $value = decide(sorted\_rcvd2[j].t\ \ - 1, target\_rcvd_{-1}[j])$
        if $value \neq$ null then WriteBack$(value, sorted\_rcvd2[j].t\ \ )$
    $(s, opID, (tag, data))$=receive()
    if $(tag = fwd)$and$(opID = cur\_opID)$ then $forwarded[s] = data$
    elseif $(tag = (tar, j, 0)$and$(opID = cur\_opID)$and$(i + 1 \le j \le f + 1))$
        $target\_rcvd[j][s] = data$
    elseif $(tag = (tar, j, -1)$and$(opID = cur\_opID)$and$(i + 1 \le j \le f + 1))$
        $target\_rcvd_{-1}[j][s] = data$
until $value \neq$ null

WriteBack$(value, ts\ \ \ \ )$
send ($cur\_opID$, WRITEBACK REQUEST) $ts\ \ \ \ $ to all servers
$S_{\ \_\ } = \emptyset$
repeat
    $(s, opID, (wrqack,$ WRITEBACK REQUEST ACK$))$=receive()
    if $(opID = cur\_opID)$ then
        $S_{\ \_\ } = S_{\ \_\ } \cup \{s\}$
        send ($cur\_opID$, WRITEBACK$(value, ts\ \ \ \ $)) to $s$
until $|S_{\ \_\ }| \ge n - f$
forall $s \notin S_{\ \_\ }$ do
    send ($cur\_opID$, WRITEBACK, $(value, ts\ \ \ \ $)) to $s$
$(s, opID, (wback,$ WRITEBAC ACK$))$=receive()
wait for $n - f$ WRITEBACK ACK $(opID = cur\_opID)$ messages
return // exits the read operation
```

Fig. 1. Read: Client Side

```
receive()
$m = receive(s, opID, (tag, data))$
$rpl\_num[s] = rpl\_num[s] + 1$
if $s \notin faulty$ then if $(opID > cur\_opID)$ or $(rpl\_num[s] > cur\_opID \times (2(f - i) + 6))$ then
    $m.tag = \perp$; add $s$ to $faulty$
return $m$
```

Fig. 2. Reader's *receive*() function

```
send WRITE MESSAGE $(value, ts)$ to all servers
$received[i] =$ null, $1 \leq i \leq n$
repeat
    receive $(s, (R\ _2))$
    $received[s] = (R\ _2)$
until $|\{i : received[i] \neq$ null$\}| \geq n - f$

$R \qquad = \{(r, opID) : |\{s : (r, opID) \in received[s].R \quad \}| \geq f + 1\}$
send FORWARD MESSAGE $(R \qquad)$ to all servers
wait for acknowledgments from $n - f$ different servers
```

Fig. 3. Write: Code execute by client

number of *replies* to delayed messages reaching a reader during a given read operation. These replies can be either direct replies to requests or values sent in response to a *target timestamps* request. Another source of delayed messages are *forwarded* messages. A writer might send many forward messages for read operations pertaining to the same reader and the actual forwarded messages might reach the reader together during a given read operation.

Delayed Replies. Handling delayed replies to request from the reader is relatively easy. A reader can simply not send a request to a server in a given operation until the server replies to all previous requests from the reader. For example, if a server s does not reply to all the requests of operation 5 and the reader meanwhile successfully completes operations 6, 7, and 8, the reader will refrain from sending any requests to s when it executes operations 6, 7, and 8. When the reader receives all replies to operation 5, it can send s the requests of the current operation, 9 for example. Implementing this idea requires some care. For example, s might send all replies to operation 5, after the reader has sent some, but not all, messages for operation 9. The reader should be able to let s catch up on the requests in operation 9. This is achieved by keeping a buffer of all outgoing requests during an operation. Whenever the reader determines that server has sent all its pending replies, it will send it all requests that are stored in the buffer. The buffer is reset at the end of the a read operation. The implementation requires that the reader does some bookkeeping to keep track of the pending replies its expects from each server. Also, whenever a server replies to a writeback message for a given operation, the reader can assume that it will not send replies to *target timestamp* requests for that operation.

Forwarded messages. Handling forwarded messages is more interesting and less straightforward. The amortized bounded solution requires that a server that receives a *forward request* must forward the value to the reader even if it did not receive a direct read request from the reader. This is needed to ensure termi-

```
Read:
upon receipt of ($opID$, READ REQUEST I) from a reader $p$
    send $(r1, opID, \{(v \quad, t \quad), (v \quad, t \quad), (v \quad, t \quad)\})$ to $p$
upon receipt of a ($opID$, READ REQUEST II) from a reader $p$
    send $(r2, opID, \{(v \quad, t \quad), (v \quad, t \quad), (v \quad, t \quad)\})$ to $p$
    $G \quad {}_2 = G \quad {}_2 \cup \{p\}$

upon receipt of ($opID$, WRITEBACK REQUEST, $ts$ ) from a reader $p$
    $G \quad = G \quad - \{(p, opID)\}$
    $G \quad {}_2 = G \quad {}_2 - \{(p, opID)\}$
    if $ts \; > t \quad + 1$ then
        $G \qquad = G \qquad \cup \{(p, opID, ts_0)\}$
    else
        send ($opID$, $wrqack$, WRITEBACK REQUEST ACK) to $p$
upon receipt of ($opID$, WRITEBACK, $(v_0, ts \;)$) from a reader $p$
    if $(ts \; > ts \quad)$ then
        $v \quad = v \;, ts \quad = ts$
    send ($opID$, wback, WRITEBACK ACK) to $p$

upon receipt of ($opID$, POTENTIAL TARGETS, $Req$ ) from a reader $p$
    for $j = i + 1$ to $f + 1$ do
        if $(ts \quad = Req \; [j])$ then                      // If a requested timestamp equals
            send $(opID, (tar, j, 0), (v \quad, t \quad))$ to $p$   // current or previous timestamp,
        if $(ts \quad = Req \; [j])$ then                      // send that value to requesting client
            send $(opID, (tar, j, 0), (v \quad, t \quad))$ to $p$
    for $j = i + 1$ to $f + 1$ do
        if $(ts \quad = Req \; [j] - 1)$ then                  // If an implicitly requested timestamp
            send $(opID, (tar, j, -1), (v \quad, t \quad))$ to $p$  // equals current or previous timestamps,
        if $(ts \quad = Req \; [j] - 1)$ then                  // send that value to requesting client
            send $(opID, (tar, j, -1), (v \quad, t \quad))$ to $p$
    $Req \qquad = Req \qquad \cup \{(p, opID, Req \;)\}$

Write:
upon receipt of a WRITE MESSAGE $(v_0, ts \;)$ from the writer $w$
    if $(ts \; > ts \quad)$                                    // If new value and timestamp received,
        $v \quad = v \quad, ts \quad = ts$                     // overwrite old value and timestamp,
        $v \quad = v \;, ts \quad = ts$
        forall $(p, opID, Req \;) \in Req \qquad$ do          // and send new value to clients waiting
            for $j = i + 1$ to $f + 1$ do                      // for a value with the new timestamp
                if $(ts \quad = Req \; [j])$ then send $(opID, (tar, j, 0), (v \quad, t \quad))$ to $p$
            for $j = i + 1$ to $f + 1$ do
                if $(ts \quad = Req \; [j] - 1)$ then send $(opID, (tar, j, -1), (v \quad, t \quad))$ to $p$
    send $(G \quad {}_2 - G \quad)$ to $w$                     // Send to writer list of pending reads
                                                               // that did not receive a forward message
    forall $(p, opID, ts) \in G \qquad$ such that $ts = ts \quad + 1$
        $G \qquad = G \qquad - \{(p, opID, ts)\}$
        send ($opID$, $wrqack$, WRITEBACK REQUEST ACK) to $p$

upon receipt of a FORWARD MESSAGE $(R \qquad {}_2)$ from the writer $w$
    forall $(r, opID) \in (R \qquad {}_2 - G \quad)$
        send $(opID, fwd, (v \quad, t \quad))$ to $r$
    $G \quad = G \quad \cup R \qquad {}_2$
    send ACK FORWARD to $w$
```

Fig. 4. Write & Read: Server Side

nation. It is also problematic because a server that is not communicating with the reader is required to send a message to the reader. This problem cannot be solved by simply requiring that the message be forwarded only by servers that have heard directly from the reader because the reader would not be guaranteed to receive enough identical forwarded messages. Also, this problem cannot be solved by requiring that the writer only forwards if $2f + 1$ servers tell the writer that they have received the reader's read request, because the writer is not guaranteed to get that many messages for a concurrent reader. Our way out of this problem is through the use of multiple forward requests from a writer. The writer will send at most $f+1$ forward requests for a given read operation and a given server will forward a value only if it has received directly the read request. To avoid that the same set of servers always forward the message to the reader (in which case the reader might never be able to decide on a value), the solution requires that every time a forward is to occur, there must be new servers involved in the forward. This way, after $f+1$ forwards, the reader will be guaranteed to receive forward messages from $f+1$ correct servers. To implement this idea, the writer includes in the forward request the id's of the server that claim to have received the reader's request. When a new forward request arrives, a server that has received a request from the reader will check if the new set of id's contain any new servers; if it does, the server forwards the message.

6 Multi-writer Solution

The multi writer case is a relatively simple extension of the single writer case. Each server keeps 3 values for each writer and all the values are sent to readers. To ensure non-skipping timestamps, the writer will first execute a complete read of a value, then do a complete write on behalf of the writer whose value it read, then it increments the timestamp and writes its value. This *helping* technique is common in wait-free implementations.

If the number of writers is bounded by c, then the number of messages sent by servers to readers can be bounded by $6 + 2c(f - i)$.

References

1. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Wait-free regular storage from byzantine components. *Unpublished manuscript.*
2. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos. In *Proc. 23rd ACM Symp. on Prin. of Dist. Comp. (PODC)*, 2004.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
4. R. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *Proc. 18th Int. Symp. on Dist. Comp. (DISC)*, 2004.
5. G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proc. 21st ACM Symp. on Prin. of Dist. Comp. (PODC)*, 2002.
6. P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3):451–500, 1998.
7. M. Li, J. Tromp, and P. M. B. Vitányi. How to share concurrent wait-free variables. *J. ACM*, 43(4):723–746, 1996.
8. D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of the 29th ACM Symp. on Th. of Comp. (STOC)*, pages 569–578, 1997.
9. D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Proc. Symp. on Rel. Dist. Sys. (SRDS)*, pages 51–58, 1998.
10. J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. 18th Int. Symp. on Dist. Comp. (DISC)*, pages 311–325, October 2002.
11. G. Peterson and J. Burns. Concurrent reading while writing ii: The multiwriter case. In *Proc. 28th IEEE Symp. Found. Comput. Sci. (FOCS)*, 1987.

Time and Communication Efficient Consensus for Crash Failures

Bogdan S. Chlebus[1,⋆] and Dariusz R. Kowalski[2]

[1] Department of Computer Science and Eng., UCDHSC, Denver, CO 80217, USA
[2] Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK

Abstract. This paper is about consensus solutions optimized simultaneously for the time *and* communication complexities. Synchronous message passing with processors prone to crashes is the computing environment. The number f of crashes can be arbitrary as long as it is smaller than the number n of processors in the system. As a building block to our consensus solutions, we consider the gossiping problem in which processors have input rumors and the goal of every processor is to learn all the rumors of the processors that have not crashed. We show that gossiping can be achieved by a deterministic algorithm working in $\mathcal{O}(\log^3 n)$ time and sending $\mathcal{O}(n \log^4 n)$ point-to-point messages. These results improve upon the best previously known deterministic solution of gossiping that operated in $\mathcal{O}(\log^2 n)$ time and generated $\mathcal{O}(n^{1+\varepsilon})$ messages, for any constant $\varepsilon > 0$. The efficient gossiping algorithm is applied to the problem of reaching consensus. In the *Consensus* problem, each processor starts with its input value and the goal is to have all processors agree on exactly one value among the inputs. First we develop a deterministic algorithm solving *Consensus* in $\mathcal{O}(n)$ time while sending $\mathcal{O}(n \log^5 n)$ messages. The best previously known algorithms solving *Consensus* in $\mathcal{O}(n)$ time had the message complexity bounded by $\mathcal{O}(n^{1+\varepsilon})$, for any constant $\varepsilon > 0$. Next we improve the *Consensus* solution so that it is early stopping, which means that it terminates in $\mathcal{O}(f + 1)$ time, where f is the number of crashes in an execution, while preserving the message complexity $\mathcal{O}(n \log^5 n)$.

1 Introduction

Reaching consensus in a distributed setting is usually achieved after an exchange of initial values that is sufficiently informative for every processor to decide. The exchange of information among all the processors is a fundamental algorithmic primitive in its own right. An abstraction of this tasks called *Gossiping* can be formulated as follows: initially every processor holds an input value, called its *rumor*, the goal is to have all processors learn all the rumors.

This abstraction of rumor spreading and gathering is well defined when all the processors remain non-faulty in the course of an execution of a gossiping protocol. The gossip abstraction needs to be modified when crashes occur, since it is possible that these processors that managed to learn some specific rumor

[⋆] The work of this author is supported by NSF Grant 0310503.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 314–328, 2006.

have crashed before the rumor was learned by all the processors. Chlebus and Kowalski [4] introduced the following form of the gossiping problem for a crash-prone environment: every processor is to learn either the rumor of processor v or the fact that v has crashed, for each processor v. We call *Gossip-with-Crashes* this version of the gossiping problem. Observe that it is possible, when processor v crashes in an execution of a solution of *Gossip-with-Crashes*, that some processors learn the original input rumor of v, while others learn that processor v has crashed, and some processors may get to know both these facts.

This paper considers the distributed environment of a synchronous message-passing system with processors prone to crashes. The number f of crashes is required to satisfy only the requirement $f < n$, where n is the number of processors in the system. Scenarios of occurrences of crashes are controlled by a fully adaptive adversary, while the number f of crashes in never a part of code of an algorithm. The model of distributed computing abstracts from the underlying communication network by assuming that any processor can send a message to any subset of processors, and the message is delivered to all its recipients within one round. If a processor crashes while sending a message, then some recipients may receive the message and some may not. The performance of a distributed protocol in such a system can be measured by natural metrics of time and communication, the latter defined conservatively as the number of point-to-point messages. For instance, consider a simple gossiping protocol in which every processor sends its input rumor to all the remaining processors in one round of communication. A processor receives the rumors of the processors that did not crash during these concurrent broadcast operations, and also possibly some of these that crashed in the course of broadcasting. If a message is not received from a processor, then this indicates that the sender crashed. This gives a simple gossiping algorithm of a constant-time performance but with a drawback that $\Omega(n^2)$ messages are exchanged in the worst case.

The above example indicates that there is a tradeoff between the time and communication complexities of solutions to *Gossip-with-Crashes*. We would like to have a gossiping protocol optimized for both time and communication, preferably with $\mathcal{O}(\text{polylog } n)$ time and $\mathcal{O}(n \text{ polylog } n)$ communication. Then such a gossiping primitive could be applied to solve other distributed problems with the goal to obtain solutions optimized for both the time *and* communication complexities.

The *Consensus* problem is about having processors decide on a common value, assuming that each processor starts with its initial input value. The problem is defined as follows.

(1) Termination: Eventually every non-faulty processor decides on some *decision value*.

(2) Validity: Any decision value is among the input values.

(3) Agreement: The decision values of any two non-faulty processors are equal.

We consider two levels of specification of algorithms. On the *existential* level, certain combinatorial structures that are a part of code of an algorithm may only be known to exist. The stronger *explicit* level of specification requires the code to be instantiated completely within time that is polynomial in the length of the input. The algorithms we consider are existential unless stated otherwise.

Our results. The summary of the contributions is as follows.

I. We show that *Gossip-with-Crashes* can be solved by a deterministic algorithm working in $\mathcal{O}(\log^3 n)$ time and sending $\mathcal{O}(n \log^4 n)$ point-to-point messages. The algorithm can be explicitly instantiated to work in $\mathcal{O}(\text{polylog } n)$ time while sending $\mathcal{O}(n \text{ polylog } n)$ point-to-point messages. The best previously known gossiping algorithm given by Georgiou, Kowalski and Shvartsman [14] operated in $\mathcal{O}(\log^2 n)$ time and used $\mathcal{O}(n^{1+\varepsilon})$ messages, for any fixed $\varepsilon > 0$; an explicit version given by Kowalski, Musial and Shvartsman [21] works in $\mathcal{O}(\text{polylog } n)$ time. The improvement we obtain is in decreasing the message complexity per processor from polynomial to polylogarithmic, while maintaining the polylogarithmic time.

II. We develop a deterministic algorithm to solve *Consensus* in $\mathcal{O}(f + 1)$ time, where f is the number of crashes occurring in an execution, while sending $\mathcal{O}(n \log^5 n)$ messages. The algorithm can be made explicit and work in $\mathcal{O}(f + 1)$ time and send $\mathcal{O}(n \text{ polylog } n)$ messages. The algorithm solving *Consensus* in the synchronous setting with processor crashes that worked in $\mathcal{O}(n)$ time and was most efficient in terms of communication among algorithms known previously was given by Galil, Mayer and Yung [4]; the algorithm had message complexity $\mathcal{O}(n^{1+\varepsilon})$, for a constant $\varepsilon > 0$. Our contribution is in decreasing the message complexity per processor from polynomial to polylogarithmic while simultaneously achieving the optimum time $\mathcal{O}(f + 1)$.

The main contribution of this paper is in showing how to solve *Gossip-with-Crashes* and *Consensus* efficiently in terms of both the time *and* message complexities, when the number f of crashes can be as large as $f = n - 1$ and the scenarios of failures are controlled by a fully adaptive adversary. The previously known solutions achieved only at most one measure of efficiency, among time and communication, to be close to the optimum value within a polylogarithmic factor, while the other complexity measure had at least a polynomial overhead per processor. The main technical obstacle to obtain the results of this paper is that the adversary is fully adaptive and limited only by the absolutely minimal requirement to leave at least one non-faulty processor in an execution.

Previous work. Gossiping in a synchronous deterministic environment with processors prone to failures with the specification that every processor needs to get to know, about each other processor v, either the original input value of v or merely the fact that processor v has crashed, was considered by Chlebus and Kowalski [4]. They developed an algorithm solving *Gossip-with-Crashes* in $\mathcal{O}(\log^2 f)$ time and with $\mathcal{O}(n \log^2 f)$ messages if $n - f = \Omega(n)$, where f is the number of crashed processors. They also showed that $\Omega(\frac{\log n}{\log(n \log n) - \log f})$ time is necessary if communication is by way of only $\mathcal{O}(n \text{ polylog } n)$ messages. In the general case of up to $f = n - 1$ crashes, a result given in [4] stated that a sub-quadratic amount of communication $\mathcal{O}(n^{1.837})$ was achievable in $\mathcal{O}(\log^2 f)$ time. This performance was later improved to the communication $\mathcal{O}(n^{1+\varepsilon})$ and $\mathcal{O}(\log^2 n)$ time by Georgiou, Kowalski and Shvartsman [14], where $\varepsilon > 0$ is an arbitrary constant being a part of code of the algorithm. These performance bounds were shown to be achievable by an explicit algorithm by Kowalski, Musial and Shvartsman [21].

The optimal complexity of *Consensus* with crashes is known with respect to the time and the number of messages when these performance metrics are considered separately. The linear lower bound $\Omega(n)$ on the number of messages holds. Dwork, Halpern and Waarts [9] found a solution with $\mathcal{O}(n \log n)$ messages but with an exponential time. Galil, Mayer and Yung [12] developed an algorithm with $\mathcal{O}(n)$ messages, thus showing that this is the optimum number of messages. The drawback of their solution is that it runs in an over-linear $\mathcal{O}(n^{1+\varepsilon})$ time, for any $0 < \varepsilon < 1$. The time complexity $f + 1$ is optimal, as shown by Fischer and Lynch [10]. A *Consensus* solution has the early-stopping property if the running time is $\mathcal{O}(f + 1)$, where the number f is the actual number of failures occurring in an execution. Galil, Mayer and Yung [12] found an early-stopping solution with $\mathcal{O}(n + fn^{\varepsilon})$ communication complexity, for any $0 < \varepsilon < 1$. Chlebus and Kowalski [4] showed that *Consensus* can be solved in $\mathcal{O}(f + 1)$ time and with $\mathcal{O}(n \log^2 f)$ messages if only the number $n - f$ of non-faulty processors satisfies $n - f = \Omega(n)$.

Related work. In most of the prior research on the *Gossiping* problem in networks prone to failures, either link failures or processor failures controlled by oblivious adversaries were considered, while the methods applied were mainly graph-theoretic; see [17] for an overview. One may consider dissemination of information similarly as spreading of an infectious disease, which has been studied in applied mathematics [3]. The general rumor-spreading paradigm has proved to be versatile also in distributed environments and in communication networks. Demers *et al.* [6] introduced epidemic algorithms for updating data bases, in which a processor regularly chooses other processors at random and transmits the rumors. Such general randomized epidemic algorithms were studied by Karp, Schindelhauer, Shenker and Vöcking [18]. Rumor-spreading algorithms to learn about the nearest resource location was given by Kempe, Kleinberg and Demers [20] and Kempe and Kleinberg [19]. Bagchi and Hakimi [2] investigated gossiping in networks with Byzantine node failures, in the case when nodes can test other nodes. Application of gossiping to gathering information about occurrences of failures was proposed by van Renesse, Minsky and Hayden [25].

The *Consensus* problem was introduced by Pease, Shostak and Lamport [22]. Fisher, Lynch and Paterson [11] showed that the problem is unsolvable in an asynchronous setting, even with only one crash failure. Hadzilacos and Toueg [16] discussed the relevance of the consensus problem to fault-tolerant broadcast and other communication primitives. Fisher and Lynch [10] showed that a synchronous solution requires $f+1$ rounds. Garay and Moses [13] developed an algorithm with polynomial-size messages operating in $f + 1$ rounds, for $n > 3f$ processors subject to Byzantine failures. The message complexity of *Consensus* in executions when no failures happen was studied by Amdur, Weber, and Hadzilacos [1] and by Hadzilacos and Halpern [15].

The message complexity of *Consensus* in the case of Byzantine faults was studied for both pure Byzantine faults and in a less demanding situation when some cryptographic authentication mechanism is available, which makes forging signatures of forwarded messages impossible. Dolev and Reischuk [8] showed a lower bound $\Omega(nf)$ on the number of signatures, for any algorithm using authentication, which is also a lower bound on the total number of messages for any protocol without authentication. They showed that any algorithm with authentication has to send $\Omega(n+f^2)$ messages, and that achieving this number of messages is possible. A conclusion is that, when faults are sufficiently malicious, $\Omega(n^2)$ is a lower bound on the number of messages.

2 Technical Preliminaries

All graphs that we consider are undirected. The notation $[k]$ denotes $\{1,\dots,k\}$, for an integer $k>0$. The names of all processors make the set $[n]$. To simplify the notation, we assume that n is a power of 2.

Expanders. The communication schemes of our algorithms are based on suitable expander graphs. Such graphs have good connectivity properties and can be defined in many ways, see [24,26]. Following Pippenger [23], we define an *a-expander,* for a positive integer a, to be a simple graph $G=(V,E)$ with $|V|=n\geq a$ nodes such that every set $A\subseteq[n]$ of size a has more than $n-a$ neighbors. This is equivalent to requiring that two disjoint sets of size a each are connected by an edge. We use the following property of expanders:

Fact 1 ([5]). *Consider a 2^a-expanding graph $G(a)$ for $a\leq\log n-2$. For every set $A\subseteq[n]$ of at least $3\cdot2^a$ nodes of $G(a)$ there is a set $B\subseteq A$ of a size at least $2^{a+1}+1$ such that the subgraph of $G(a)$ induced by the set B is of a diameter at most $2a+2$.*

Communication graphs. Processors send messages directly to these processors that are their neighbors in suitable *communication graphs.* We use graphs denoted by $G(i)$ for this purpose, where $G(i)=([n],E_i)$ is a 2^i-expanding graph, for $0\leq i\leq\log n$. Graph $G(0)$ is the complete graph on n nodes, which guarantees a possibility for any pair of nodes to exchange messages directly if necessary.

Overhead δ_n. The probabilistic method allows to show that there are 2^a-expanding graphs of n nodes with the maximum degree $\mathcal{O}(n2^{-a}\log n)$. Ta-Shma, Umans, and Zuckerman [26] showed a polynomial-time construction of 2^a-expanding graphs with the maximum degree $\mathcal{O}(n2^{-a}$ polylog $n)$. We denote the maximum degree of the used 2^a-expanding graphs by $\Delta(a,n)$, or by $\Delta(a)$ when the number n is understood from context. The maximum of the fractions $2^a\Delta(a,n)/n$, taken over all the integers a such that $0\leq a\leq\log n$, is denoted by δ_n. The number δ_n can be interpreted as the complexity overhead resulting from using communication graphs. There exists a family of n-node 2^a-expanding graphs with $\delta_n=\mathcal{O}(\log n)$, while the construction of [26] yields 2^a-expanding graphs with δ_n that is polylogarithmic in n.

Schedules. A permutation π of $[n]$ is called a *schedule* when the elements of $[n]$ are interpreted as jobs and the permutation π specifies in which order to perform them. Let $\mathcal{S} = \langle \pi_1, \ldots, \pi_n \rangle$ be a list of schedules. Let $Q \subseteq [n]$ be a subset of $q = |Q| \leq n$ *active* processors. Let $d > 0$ be an integer parameter called *delay.*

Consider the following synchronous distributed $\mathcal{S}$*-steered operation* in which only active processors perform jobs. In every consecutive round each active processor v performs the first job according to π_v about which v does not know if it has been already performed. If a job is performed for the first time at some round τ, then all the active processors learn about this by the end of round $\tau + d - 1$.

Measuring work. Following Kowalski, Musial and Shvartsman [21], we use the following measure of quality of parameters $\mathcal{S}$, q and d. Let $\mathcal{W}(\mathcal{S}, q, d)$ be the maximum number of all the available processor steps in $\mathcal{S}$-steered operations to perform n jobs, taken over all the executions in which only q processors are active and accounted for and with delay d. This measure allows to capture an estimate of work needed to perform n jobs by any set of q processors using their individual schedules, when the information about the completion of any job may be delayed for up to d rounds. The following facts provide the existence of lists $\mathcal{S}$ good in terms of the metric $\mathcal{W}$.

Fact 2 ([14,21]). *There exists a list $\mathcal{S}$ of schedules such that the inequality $\mathcal{W}(\mathcal{S}, q, d) \leq (n + 10dq) \log n$ holds, for all parameters $1 \leq d, q \leq n$.*

Fact 3 ([21]). *There is an explicit list $\mathcal{S}$ for which the estimate $\mathcal{W}(\mathcal{S}, q, d) = \mathcal{O}((n + dq)$ polylog $n)$ holds for all the parameters such that $1 \leq d, q \leq n$.*

Overhead α_n. Every round of computation of an active processor contributes a unit to the measure $\mathcal{W}(\mathcal{S}, q, d)$ until the processor halts. Let α_n denote the maximum of the ratios $\mathcal{W}(\mathcal{S}, n/d, d)/n$ taken over all the values $1 \leq d \leq n$, for a given list $\mathcal{S}$ of n schedules, each of n jobs. The number α_n can be interpreted as the complexity overhead resulting from the processors communicating according to their schedules. By Fact 2, there exists a list of schedules such that $\alpha_n \leq 11 \log n$. By Fact 3, there is an explicit list of schedules with $\alpha_n = \mathcal{O}(\text{polylog } n)$.

From communication graphs and schedules to gossiping. Any two processors can communicate directly, so the communication schemes are used to optimize for message complexity only. Now we describe the underlying connections between gossiping and both the communication graphs and schedules.

One component of the underlying communication structure is provided by a sequence of communication graphs with suitable expansion properties. These graphs are not sufficient by themselves, because when processors crash, then a connected graph may be broken into separate connected components. Communication graphs $G(i)$, as specified in Fact 1, are used; such explicit graphs exist as shown in [26]. Fact 1 guarantees that at least one large component remains when sufficiently many nodes do not crash.

Schedules provide the remaining part of the communication scheme. They are used, concurrently with the communication graphs, to determine patterns to reach to nodes outside neighborhoods in the communication graphs. This is needed to ensure that the processors in a large component of a communication graph collect rumors from the nodes that do not belong to the component.

A gossiping algorithm is structured as a sequence of iterations called epochs. The scheme of communication in an epoch is determined by a communication graph, which depends on the number of the epoch. The set $\mathcal{S}$ of schedules used in all epochs depends on the number n of processors. The communication graphs determine the number of messages by their degrees, since a processor communicates with its neighbors in a round. The number of point-to-point messages sent in an epoch, which results from using schedules in $\mathcal{S}$, is bounded from above by $\mathcal{W}(\mathcal{S}, q, d)$, for suitable values of q and d that depend on the epoch. The parameter q is a lower bound on the number of non-faulty processors chosen to make this epoch efficient. The parameter d corresponds to a delay of information flow between nodes in the largest connected component in the communication graph, measured as the diameter of the largest connected component times the number of messages per round. The time of any epoch is $\Theta(\mathcal{W}(\mathcal{S}, q, d)/(qd))$. Facts 2 and 3 allow to obtain a polylogarithmic estimate of this time bound.

3 Gossiping in Crash-Prone Environments

In this section we describe a deterministic algorithm called Alg-Gossip.

Private data structures of the processors. Every processor v maintains an array called $\texttt{Rumors}_v$ of rumors it has collected and a list called $\texttt{Uninformed}_v$ of processors that should be informed in case v gathered complete information about the other processors. Initially $\texttt{Uninformed}_v$ contains all the processors except for v. These two structures are ordered according to permutation π_v, which is the schedule of v.

The processors to communicate directly with are determined in the course of an execution based on the values stored in Rumors and Uninformed and on the communication graph. A message sent to a neighbor in a communication graph is called *spreading*. A message sent to a processor determined by the contents of Rumors is called *requesting*. A message sent to a processor determined from the contents of list Uninformed is called *informing*. If a processor sends a message to a processor from which it received a requesting message in the current round, then this message is called *replying*.

Each processor v maintains a list called $\texttt{Faulty}_v$, which contains all the processors about which v has learned that they are faulty. “Learning” is used a technical term defined as follows: processor *v learns that processor w is faulty* if either w is on some list $\texttt{Faulty}_z$ received by v in the current round, or w did not answer to a requesting message sent by v in the previous round, or w is a neighbor of v in the communication graph $G(\log n - i)$ and v did not receive a spreading message from w in the current round.

Epochs. Algorithm Alg-Gossip iterates epochs numbered 2 through $\log n$. In epoch i, graph $G(\log n - i)$ is used as the communication graph. The processors

Receiving: Every processor receives the messages sent to it.

Local computation: Each working processor v updates its private data structures as follows:

- If v received an unknown rumor of processor w, then v puts the rumor at $\texttt{Rumors}_v[w]$.
- If v received the information that processor w was faulty and v did not have a rumor of w and did not get it in the receiving round, then v puts marker $\perp$ at position $\texttt{Rumors}_v[w]$.
- If v is working or informing and received the information that processor w is faulty, then v adds name w to its list $\texttt{Faulty}_v$ and removes w from list $\texttt{Uninformed}_v$ in case w is there.
- If array $\texttt{Rumors}_v$ becomes completely filled, then v changes its status to informing.

Sending: Every non-faulty processor v sends the following:

- If v is working or informing, then v sends a spreading message with its local knowledge to these neighbors in the communication graph $G(\log n - i)$ that are not in list $\texttt{Faulty}_v$.
- If v is working, then v sends a requesting message to $\Delta(\log n - i)$ processors, the first according to the sequence π_v about which v has no information, or to all the remaining such processors, if there are less than $\Delta(\log n - i)$ of them.
- If v is working, then v sends an informing message to $\Delta(\log n - i)$ processors, the first according to the sequence π_v from list $\texttt{Uninformed}$, or to all the remaining processors in case there are less than $\Delta(\log n - i)$ of them, and next removes them from this list.
- Processor v sends a replying message with its local knowledge to all the processors from which v received a requesting message in this round.

Fig. 1. One phase of epoch i in algorithm Alg-Gossip

use also permutations from some list $\mathcal{S}$ of schedules. The specific goal of epoch i is as follows: if at least $3n/2^i$ processors are non-faulty during the whole epoch, then some subset of them of a size at least $2n/2^i$ will gather the needed information about all the processors and spread the information first among themselves, and later, in a cooperative fashion, will spread this knowledge to all the non-faulty processors. An epoch lasts exactly $T = 2\alpha_n \cdot (2\log n) + 2$ phases, each consisting of three rounds; see Figure 1.

At the end of epoch i, processor v sets its current status as follows: v becomes waiting if v was informing or waiting and $\texttt{Uninformed}_v$ is not empty; v becomes done if v was informing or waiting and $\texttt{Uninformed}_v$ is empty; v remains working in all the remaining cases. Note that only the processors that start epoch i as working are active during epoch i, while all the waiting and done processors merely reply to the received requesting messages, if any.

Lemma 1. *Algorithm* Alg-Gossip *sends $\mathcal{O}(n\delta_n\alpha_n \log^2 n)$ messages and works in $\mathcal{O}(\alpha_n \log^2 n)$ time.*

As discussed in Section 2, there exist communication graphs with $\delta_n = \mathcal{O}(\log n)$ and explicit communication graphs with $\delta_n = \mathcal{O}(\text{polylog } n)$. Similarly, there are schedules with $\alpha_n = \mathcal{O}(\log n)$ and explicit schedules with $\alpha_n = \mathcal{O}(\text{polylog } n)$. This combined with Lemma 1 implies:

Theorem 1. *Algorithm* Alg-Gossip *works in $\mathcal{O}(\log^3 n)$ time sending $\mathcal{O}(n \log^4 n)$ point-to-point messages. It can be explicitely instantiated so that it works in $\mathcal{O}(\text{polylog } n)$ time while sending $\mathcal{O}(n \text{ polylog } n)$ point-to-point messages.*

4 Consensus in Linear Time

We consider the case of binary consensus, which restricts the input values to be either 0 or 1, to simplify the exposition. The algorithm described in this section is called LT-Consensus. Each processor uses a variable called its *preference*, initialized to its input value. A preference is a possible decision value, given the current knowledge of a processor. At the beginning each processor is *hesitant*. If a processor is convinced that its preference is final, it changes its status from hesitant to *convinced*. Algorithm LT-Consensus operates by going through phases numbered from 2 to $\log n$. The preferences can be modified during a phase. If the status is changed, then this happens at the end of a phase. The final decision is made at the end of the algorithm. Phase i, for $2 \le i \le \log n$, consists of three parts as specified in Figure 2. After the last phase, a processor decides on its current preference.

Correctness of consensus. We break the proof of correctness into parts corresponding to the termination, validity and agreement conditions. We show the termination property by deriving an estimate on time performance.

Lemma 2. *Agorithm* LT-Consensus *works in $\mathcal{O}(n)$ time.*

It follows from the specification of algorithm LT-Consensus that if there is no occurrence of 0 as an initial value, then no preference on 0 is ever set. Similarly, if there is no occurrence of 1 as an initial value, then no preference of 1 occurs in an execution. This shows validity of algorithm LT-Consensus.

Lemma 3. *After phase i of algorithm* LT-Consensus*, there are at most $\frac{3n}{2^{\ +1}}$ non-faulty and hesitant processors.*

Lemma 4. *Consider some processors v and w that are non-faulty and hesitant at the end of Part 2 of the phase i and become convinced at the end of the phase i of algorithm* LT-Consensus*. If processor v learns about the preference* 1 *of some other processor in Part 1 or Part 2 of phase i, then processor w also learns about the preference* 1 *in Part 2 of phase i.*

Part 1: Gossiping the preferences. Alg-Gossip is executed by the hesitant processors. We treat convinced processors as "crashed" so they do not cooperate in collecting and spreading rumors, but they always reply to requesting messages; instead of the status "faulty" they have the status "convinced" in arrays `Rumors`. The preferences at the start of this part are treated as rumors. When the gossiping is complete, each hesitant processor updates its preference to the maximum preference received.

Part 2: Flooding the preferences. This part takes $\frac{3n}{2} + 4\log n$ rounds and consists of exchanging and updating preferences by flooding in the communication graph $G(\log n - i)$. Only the hesitant processors participate. In the first round, each hesitant processor v sends its preference to all its neighbors in the communication graph. During the remaining rounds, a hesitant processor v changes its preference from 0 to 1 only if it has heard a preference 1 from some of its neighbors, and if so then it also sends its new preference to all its neighbors in the same round.

Part 3: Checking the sizes of hesitant connected components. This part takes $2\log n$ rounds. The hesitant processors cooperatively learn the size of their connected components in the communication graph $G(\log n - i)$. Every hesitant processor maintains a list of nodes in the connected component it knows about, which is included in every message. Initially a processor knows only about itself. In the first round, each hesitant processor sends its name to all the neighbors. When a hesitant processor receives a message, then it merges the obtained list of nodes with its own, and if the list expands, then the list is sent to all the neighbors.

Concluding actions. A hesitant processor changes its status to convinced, if it heard about at least $2n/2^i$ processors, including itself, during Part 3 of this phase; otherwise, if $i < \log n$, then the hesitant processor resets its preference to the initial value.

Fig. 2. Phase i of algorithm LT-Consensus

Lemma 5. *Agreement is achieved by all the processors running algorithm* LT-Consensus.

Proof. Assume, to arrive at a contradiction, that there are some two processors, say, v and w that decide on different values. We first consider the case when these processors are convinced.

Case 1. Processors v and w become convinced at the end of the same phase i.

Without the loss of generality, we may assume that v decides on 0 and w decides on 1. Note that w could learn about some preference 1 only in Part 1 or Part 2 of phase i. Hence, by Lemma 4, processor v learns about preference 1 by the end of Part 2. More precisely, if it does not learn about it during Part 1, then by Lemma 4 it learns it by the end of Part 2.

Case 2. Processors v and w become convinced at the ends of different phases i and j, respectively.

Without the loss of generality, we may assume that v decides on 0 while w decides on 1. It follows, from the property of gossip in Part 1 of the phase i, that $i < j$. Otherwise processor v would know the preference of the convinced processor w, which is 1, by the end of Part 1 of phase $i > j$, and also would set its preference to 1. Then it would become convinced at the end of phase i, and finally decided on 1 at the end of the execution. Consider the phase i. It follows that all the non-faulty and hesitant processors that do not become convinced at the end of this phase change their preferences to the initial value, which must be 0. This is because if some of these processors, say z, has 1 as the initial value, then it has value 1 as its preference during the whole phase i. This means that all the processors that are non-faulty at the end of phase i learn about the preference on 1 during Part 1 of this phase, so v does too. Consequently, processor v would prefer value 1 and become convinced, hence it would decide on 1, which is a contradiction. All the processors that have become convinced by the beginning of phase $i + 1$ decide on 0, and all the remaining non-faulty and hesitant processors have preferences equal to 0 at the beginning of phase $i+1$. By the same argument as for the validity property, where all the current preferences of the non-faulty processors are equal to 0, we obtain that all the processors decide on 0, which is a contradiction with the decision of processor w. This completes the proof of agreement among the convinced processors.

It remains to consider the non-faulty processors that remain hesitant till the end of an execution. By Lemma 3, there is at most one such a processor; let it be denoted by v. If v is the only non-faulty processor at this point, than the agreement holds. Otherwise there is another non-faulty processor w, which then must become convinced in some phase $i \leq \log n$. Suppose, to the contrary, that w decides on a different value than processor v.

If w becomes convinced after the last phase, then, by the property of gossip in Part 1 of phase $\log n$, both the preferences of v and w are set to 1 after Part 1. The preferences remain unchanged by the time the decision is made, so both processors decide on the same value, which is a contradiction.

Suppose w becomes convinced after phase $i < \log n$. We already proved that all the convinced non-faulty processors decide on the same value, which means on the same value as w does. There are two subcases. If w decides on 1, then v learns about this in Part 1 of phase $\log n$, updates its preference to 1, and decides on 1, which is a contradiction. The remaining subcase is when w decides on 0. Since v decides on 1, either the initial value of v is 1 or v learns about the preference 1 during phase $\log n$ from some processor that is non-faulty at the beginning of phase $\log n$ and not convinced. In both cases processor w would have learned about preference 1, being the initial value of one of these two processors, in Part 1 of phase i, but w decides on 0, which is a contradiction. This completes the proof of the property of agreement.

Lemma 6. *The total number of point-to-point messages sent by algorithm* LT-Consensus *is* $\mathcal{O}(n\delta_n\alpha_n \log^3 n)$.

Proof. The message complexity of Parts 1 in all the phases is $\log n - 1$ times the message complexity of gossiping in Part 1. This, in view of Lemma 1, gives $\mathcal{O}(n\delta_n\alpha_n \log^3 n)$ point-to-point messages sent. The message complexity of Parts 2 in all phases is

$$\mathcal{O}\big(\sum_{i=1}^{\log n} \big(\frac{n}{2^i} + 4\log n\big) \cdot \Delta(\log n - i)\big) = \mathcal{O}(n\delta_n \log n)$$

by Lemma 3. The total number of point-to-point messages sent in Parts 3 in all the phases is

$$\mathcal{O}\big(\sum_{i=1}^{\log n} \frac{n}{2^i} \cdot \Delta(\log n - i)\log n\big) = \mathcal{O}(n\delta_n \log^2 n)\ ,$$

also by Lemma 3.

Now we summarize the properties of algorithm LT-Consensus. As discussed in Section 2, there exist communication graphs with $\delta_n = \mathcal{O}(\log n)$ and explicit communication graphs with $\delta_n = \mathcal{O}(\text{polylog } n)$. Regarding the parameter α_n, there are schedules with $\alpha_n = \mathcal{O}(\log n)$ and explicit schedules with $\alpha_n = \mathcal{O}(\text{polylog } n)$. These facts combined with the preceding analysis of time and message complexities yield the following result for arbitrary $f < n$:

Proposition 1. *Algorithm* LT-Consensus *sends* $\mathcal{O}(n \log^5 n)$ *messages and works in* $\mathcal{O}(n)$ *time. It can be explicitly instantiated to work in* $\mathcal{O}(n)$ *time and with* $\mathcal{O}(n \text{ polylog } n)$ *messages.*

5 Early-Stopping Consensus

In this section we develop algorithm ES-Consensus which is an early stopping solution to *Consensus*. We assume that n is a power of 2 to avoid rounding.

Let Gossip(k) denote a variant of algorithm Alg-Gossip, in which the goal is to have every processor learn, about all processors v such that $1 \leq v \leq k$, what is the rumor of v or that v has already crashed. What the algorithm does depends on the size of k. When $k > \log^3 n$, then we use algorithm Alg-Gossip modified such that rumors of processors with names larger than k are omitted from circulated messages. For $k \leq \log^3 n$, a simple algorithm suffices, in which each processor $p \in [k]$ broadcasts its rumor directly to all processors in one round. Let $T_g(k)$ denote the number of rounds needed for Gossip(k) to terminate. Number $T_g(k)$ is a part of code of the algorithm so that each processor stops after exactly $T_g(k)$ rounds. Note that $T_g(k) = \mathcal{O}(k)$, since Alg-Gossip takes $\mathcal{O}(\log^3 n)$ rounds. Additionally, the communication during this subroutine is $\mathcal{O}(\min\{nk, n\log^4 n\}) = \mathcal{O}(n \log^4 n)$.

```
Part 1: local consensus: If $v \in [2^j]$ then do the following:
    (a) run LT-Consensus($2^j$) on preferences during exactly $T_c(2^j)$ rounds
    (b) update the preference to the decision made in LT-Consensus($2^j$)
    (c) update the status to leaving
Part 2: first gossip:
    (a) run Gossip($2^j$) with preferences and statuses as rumors during exactly $T_g(2^j)$ rounds
    (b) update the preference to the minimum of the received preferences
Part 3: second gossip:
    (a) run Gossip($2^j$) with preferences and statuses as rumors during exactly $T_g(2^j)$ rounds
    (b) update the preference to the minimum of the received preferences
    (c) if the preference of a leaving processor was received during gossip in
        (a) in this part, then set status to leaving
Part 4: third gossip:
    (a) if $2^j > \log^3 n$ then run Gossip($n$) during exactly $T_g(n)$ rounds else
        run PartialGossip, with preferences and statuses as rumors
    (b) update the preference to the minimum of the current preference and
        the received preferences of leaving processors
    (c) if $2^j > \log^3 n$ and the status is leaving then decide on the preference
        and halt; else, if $2^j \leq \log^3 n$ and the status is leaving and a message
        was received during the last round of (a) of this part, then decide on
        the preference and halt
    (d) if the preference of a leaving processor was received during the gossip
        in (a) of this part, then set the status to leaving
```

Fig. 3. Epoch j in algorithm ES-Consensus, for $1 \leq j \leq \log n - 1$. Code for processor v

We will also use a subroutine PartialGossip which works in four rounds as follows. In round 1, all processors send their rumors to all processors in $[\log^3 n]$. In round 2, every processor in $[\log^3 n]$ broadcasts its knowledge, consisting of rumors and the information about crashes, to all processors. Round 3 is the same as round 1, and round 4 is the same as round 2. The message complexity is $\mathcal{O}(n \log^3 n)$. This procedure assures that if there is a processor in $[\log^3 n]$ which is non-faulty by the round of termination of this procedure, then the gossip has been completed.

Let LT-Consensus(k) denote the $(k-1)$-reliable LT-Consensus algorithm performed by the first k processors, which takes exactly $T_c(k) = \mathcal{O}(k)$ rounds. Similarly as for the Gossip(k) subroutine, each processor halts after exactly $T_c(k)$ rounds, despite the fact it might decide before this round. The decision made during this procedure is not the final one, but is used only to possibly modify the current preference. The message complexity of this subroutine is $\mathcal{O}(2^j \log^5 2^j) = \mathcal{O}(2^j \log^5 n)$.

The algorithm keeps updating local preferences of processors. A preference contains a value from among the initial ones. Each preference is initialized to the input value. Each processor has a status. It is initialized to *working* and next may be changed to *leaving*. The algorithm proceeds through *epochs* numbered from 1 to $\log n$. Epoch j, for $1 \le j \le \log n - 1$, consists of four *parts*, see Figure 3. A part is broken into up to four consecutive *components*, marked by letters (a) through (d). The final epoch number $\log n$ consists of calling CONSENSUS(n) on preferences, making the final preference the decision and halting.

Correctness and complexity. Any preference value is among the initial values, which implies validity.

Lemma 7. *Every two halted processors decide on the same value.*

Lemma 8. *Every processor halts by round $\mathcal{O}(f+1)$.*

Consider epoch j. There are $\mathcal{O}(2^j \log^5 n)$ messages sent in CONSENSUS(2^j). There are $\mathcal{O}(\min\{n2^j, n\log^4 n\})$ messages exchanged during GOSSIP(2^j). Finally, there are $\mathcal{O}(n \log^3 n)$ messages sent during PARTIALGOSSIP. The total number of messages is bounded from above by

$$\mathcal{O}\Big(\sum_{j \le b}(2^j \cdot \log^5 n + n \log^4 n)\Big) = \mathcal{O}(n \log^5 n) \ .$$

These facts yield the following final result:

Theorem 2. *Algorithm* ES-CONSENUS *solves* Consensus *in $\mathcal{O}(f+1)$ time and with $\mathcal{O}(n \log^5 n)$ messages, when $f < n$ is the number of crashes in an execution. It can be explicitly instantiated to work in $\mathcal{O}(f+1)$ time and with $\mathcal{O}(n$* polylog *$n)$ messages.*

References

1. S. Amdur, S. Weber, and V. Hadzilacos, On the message complexity of binary agreement under crash failures, *Distributed Computing,* 5 (1992) 175 - 186.
2. A. Bagchi, and S.L. Hakimi, Information dissemination in distributed systems with faulty units, *IEEE Transactions on Computing,* 43 (1994) 698 - 710.
3. N.T.J. Bailey, *"The Mathematical Theory of Infectious Diseases and its Applications,"* Charles Griffin, London, 1975.
4. B.S. Chlebus, and D.R. Kowalski, Gossiping to reach consensus, in *Proceedings,* 14*th ACM Symposium on Parallel Algorithms and Architectures (SPAA),* 2002, pp. 220 - 229.
5. B.S. Chlebus, D.R. Kowalski, and A.A. Shvartsman, Collective asynchronous reading with polylogarithmic worst-case overhead, in *Proceedings,* 36*th ACM Symposium on Theory of Computing (STOC)*, 2004, pp. 321 - 330.
6. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swineheart, and D. Terry, Epidemic algorithms for replicated database maintenance, in *Proceedings,* 6*th ACM Symposium on Principles of Distributed Computing (PODC),* 1987, pp. 1 - 12.

7. R. De Prisco, A. Mayer, and M. Yung, Time-optimal message-efficient work performance in the presence of faults, in *Proceedings, 13th ACM Symposium on Principles of Distributed Computing (PODC)*, 1994, pp. 161 - 172.
8. D. Dolev, and R. Reischuk, Bounds on information exchange for Byzantine agreement, *Journal of the ACM,* 32 (1985) 191 - 204.
9. C. Dwork, J. Halpern, and O. Waarts, Performing work efficiently in the presence of faults, *SIAM Journal on Computing,* 27 (1998) 1457–1491.
10. M. Fisher, and N. Lynch, A lower bound for the time to assure interactive consistency, *Information Processing Letters*, 14 (1982) 183 - 186.
11. M. Fisher, N. Lynch, and M. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM,* 32 (1985) 374 - 382.
12. Z. Galil, A. Mayer, and M. Yung, Resolving message complexity of Byzantine agreement and beyond, in *Proceedings, 36th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1995, pp. 724 - 733.
13. J.A. Garay, and Y. Moses, Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds, *SIAM Journal on Computing,* 27 (1998) 247 - 290.
14. C. Georgiou, D.R., Kowalski, and A.A. Shvartsman, Efficient gossip and robust distributed computation, *Theoretical Computer Science*, 347 (2005) 130 - 166.
15. V. Hadzilacos, and J.Y. Halpern, Message-optimal protocols for Byzantine agreement, *Mathematical Systems Theory,* 26 (1993) 41 - 102.
16. V. Hadzilacos, and S. Toueg, Fault-tolerant broadcast and related problems, in *"Distributed Systems,"* 2nd ed., S. Mullender (Ed.), Eddison-Wesley, 1993, pp. 97 - 145.
17. J. Hromkovic, R. Klasing, A. Pelc, P. Ruzicka, and W. Unger, *"Dissemination of Information in Communication Networks: Broadcasting, Gossiping, Leader Election, and Fault-Tolerance,"* Springer-Verlag, 2005.
18. R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking, Randomized rumor spreading, in *Proceedings, 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000, pp. 565 - 574.
19. D. Kempe, and J. Kleinberg, Protocols and impossibility results for gossip-based communication mechanisms, in *Proceedings, 43rd IEEE Symposium on Foundations of Computer Science (FOCS),* 2002, pp. 471 - 480.
20. D. Kempe, J. Kleinberg, and A. Demers, Spatial gossip and resource location protocols, *Journal of the ACM*, 51 (2004) 943 - 967.
21. D.R. Kowalski, P. Musial, and A.A. Shvartsman, Explicit combinatorial structures for cooperative distributed algorithms, in *Proceedings, 25th International Conference on Distributed Computing Systems (ICDCS)*, 2005, pp. 48 - 58.
22. M. Pease, R. Shostak, and L. Lamport, Reaching agreement in the presence of faults, *Journal of the ACM,* 27 (1980) 228 - 234.
23. N. Pippenger, Sorting and selecting in rounds, *SIAM Journal on Computing,* 16 (1987) 1032–1038.
24. O. Reingold, S.P. Vadhan, and A. Wigderson, Entropy waves, the zig-zag graph product, and new constant-degree expanders and extractors, *Annals of Mathematics*, 155 (2002) 157 - 187.
25. R. van Renesse, Y. Minsky, and M. Hayden, A gossip-style failure detection service, in *Proceedings, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998, pp. 55 - 70.
26. A. Ta-Shma, C. Umans, and D. Zuckerman, Loss-less condensers, unbalanced expanders, and extractors, in *Proceedings, 33rd ACM Symposium on Theory of Computing (STOC),* 2001, pp. 143 - 152.

Subconsensus Tasks: Renaming Is Weaker Than Set Agreement

Eli Gafni[1], Sergio Rajsbaum[2], and Maurice Herlihy[3]

[1] Computer Science Department UCLA Los Angles, CA 90095
eli@cs.ucla.edu

[2] Instituto de Matemáticas, Universidad Nacional Autónoma de México, Ciudad Universitaria, D.F. 04510, Mexico
rajsbaum@math.unam.mx

[3] Brown University, Computer Science Department, Providence, RI 02912
mph@cs.brown.edu

Abstract. We consider the the relative power of two important synchronization problems: set agreement and renaming. We show that renaming is strictly weaker than set agreement, in a round-by-round model of computation. We introduce new techniques, including previously unknown connections between properties of manifolds and computation, as well as novel "symmetry-breaking" constructions.

1 Introduction

A *task* in an asynchronous system is a problem where each process starts with a private input value, communicates with the others, and halts with a private output value. A *protocol* is a program that solves the task. In asynchronous systems, it is desirable to design protocols that are *wait-free*: any process that continues to run will halt with an output value in a fixed number of steps, regardless of delays or failures by other processes.

We are interested in the *relative power* of tasks: given two tasks, can one be used to implement the other, or are they incomparable? One way to measure the relative power of tasks is by *consensus numbers* [11]. If a task can solve consensus [7] for n processes is *universal* in a system of n processes in the sense that it can be used to construct a protocol that solves any other task. Moreover, any task that solves consensus for n processes also solves consensus for fewer.

One question that has received substantial attention, and yet is still far from understood, is the relative power of tasks too weak to solve consensus for two processes, the so-called "subconsensus" tasks. For example, it is known that read/write memory cannot solve consensus. After a substantial effort, it was discovered that two other tasks, set agreement [6] and renaming [1], which are both subconsensus tasks, cannot be implemented in read/write memory [3,10,13]. It follows that subconsensus tasks have a "fine structure", inaccessible by consensus-based analysis.

In this paper, we shed further light on this fine structure by showing that the renaming task is strictly weaker than the set agreement task, in a natural model of asynchronous computation. a surprising result, since the two tasks are superficially quite dissimilar.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 329–338, 2006.

We introduce new techniques that may be helpful for other problems: a previously unknown connection between properties of manifolds and computation, as well as novel "symmetry-breaking" constructions.

2 Model

2.1 Combinatorial Topology

We start by reviewing standard terminology from combinatorial topology (mostly taken from Munkres [12]). We employ concepts from combinatorial topology, in a style similar to Attiya and Rajsbaum [2].

A *simplicial complex* $\mathcal{A}$ is a collection of finite non-empty sets, such that if A is an element of $\mathcal{A}$, so is every non-empty subset of A. An element A of $\mathcal{A}$ is called a *simplex*, and an element of A is called a *vertex*. The dimension of A is one less than its number of elements. A subcollection $\mathcal{B}$ of $\mathcal{A}$ is called a *subcomplex* of $\mathcal{A}$ if $\mathcal{B}$ is itself a complex. A subset B of simplex A is itself a simplex called a *face* of A. We refer to an n-dimensional simplex as an n-simplex, and we sometimes use superscripts (for example A^n) to indicate a simplex's dimension. The *dimension* of a complex is the largest dimension of any of its simplexes. An n-dimensional complex $\mathcal{A}$ is *full to dimension* n if every simplex in $\mathcal{A}$ is a face of an n-simplex. All complexes considered here are full to some dimension.

Let $\mathcal{A}$ and $\mathcal{B}$ be complexes. A *simplicial map* $\mathcal{M} : \mathcal{A} \rightarrow \mathcal{B}$ carries vertexes of $\mathcal{A}$ to vertexes of $\mathcal{B}$ so that every simplex of $\mathcal{A}$ maps to a simplex of $\mathcal{B}$. An n-complex is *colored* if there exists a map id sending each vertex to $\{0, \ldots, n\}$ such that each n-simplex is labeled with $n + 1$ distinct colors. If A is a colored simplex, $ids(A)$ denotes its set of colors. If $\mathcal{A}$ and $\mathcal{B}$ are colored complexes, then a simplicial map $\mathcal{M} : \mathcal{A} \rightarrow \mathcal{B}$ is *color-preserving* if $id(v) = id(\mathcal{M}(v))$ for every vertex v in $\mathcal{A}$.

For a simplex S, $\chi(S)$, the *standard chromatic subdivision*, is defined as follows. Each vertex of $\chi(S)$ is a pair (s, F), where F is a face of S, and s a vertex of F. A set of vertexes of $\chi(S)$ define a simplex if for each pair (s, F) and (s', F'), $id(s)$ and $id(s')$ are distinct, and either $F \subseteq F'$, or $F' \subseteq F$. As s ranges over the vertexes

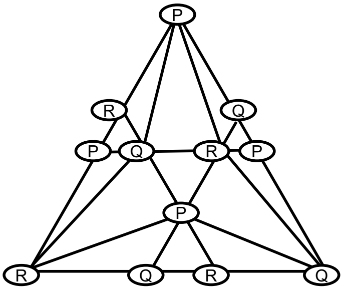


Fig. 1. Standard Chromatic Subdivision

of S, the vertexes (s, S) define the *central simplex* of $\chi(S)$. (The standard chromatic subdivision is the colored analog of the *standard barycentric subdivision* [12, p.96].)

A complex is *strongly connected* if any two n-simplexes can be joined by a "chain" of n-simplexes in which each pair of neighboring simplexes has a common $(n-1)$-dimensional face. A complex $\mathcal{M}$ is a *manifold with boundary* (sometimes called a *pseudomanifold with boundary*) if it is strongly connected, and each $(n-1)$-simplex in $\mathcal{M}$ is contained in precisely one or two n-simplexes. An $(n-1)$ simplex in $\mathcal{M}$ is an *interior* simplex if it is contained in two n-simplexes, and a *boundary* simplex otherwise. The *boundary* subcomplex of $\mathcal{M}$, denoted $\partial\mathcal{M}$, is the subcomplex induced by its boundary simplexes. For brevity, we will use "manifold" as an abbreviation for "manifold with boundary".

2.2 Distributed Computation

We are given a set of $n+1$ process ids $\{0, \ldots, n\}$. An initial or final state of a process is modeled as a vertex $v = (P, w)$, a pair consisting of a process id P and a value w. We say the vertex is *colored* with the process id. A set of $n+1$ mutually compatible initial or final states is modeled as an n-simplex $A^n = (a_0, \ldots, a_n)$.

A *task specification* is given by a colored *input complex* $\mathcal{I}$, a colored *output complex* $\mathcal{O}$, and a recursive relation Δ carrying each m-simplex of $\mathcal{I}$ to a set of m-simplexes of $\mathcal{O}$, for each $0 \leq m \leq n$. Δ has the following interpretation: if the $(m+1)$ processes named in S^m start with the designated input values, and the remaining $n-m$ processes fail without taking any steps, then each simplex in $\Delta(S^m)$ corresponds to a legal final state of the non-faulty processes. A *protocol* is a program in which processes receive their inputs, communicate via shared objects, and eventually return with mutually compatible decision values. Any protocol has an associated *protocol complex* $\mathcal{P}$, in which each vertex is labeled with a process id and that process's final state (called its *view*). Each simplex thus corresponds to an equivalence class of executions that "look the same" to the processes at its vertexes. The protocol complex corresponding to executions starting from a fixed simplex S^m is denoted $\mathcal{P}(S^m)$. We typically use $\mathcal{P}$ to denote both a protocol and its complex. A protocol *solves* a task if there exists a color-preserving simplicial *decision map* $\delta : \mathcal{P} \to \mathcal{O}$ such that for each simplex $R^m \in \mathcal{P}(S^m)$, $\delta(R^m) \in \Delta(S^m)$. Note that a protocol can also be considered as a task.

3 Tasks

In the *immediate snapshot* protocol [3], the processes share an array of registers, one for each process, which for brevity we refer to as the *immediate snapshot memory*. This memory provides a single operation: *write-read*, which writes the process state to its register and takes an instantaneous snapshot of the others. Logically, the processes act as if the round is divided into *phases*, where in each phase a set of processes is chosen, disjoint from any set chosen in an earlier phase. The processes in the set simultaneously write to their own registers, and then read all the others. The protocol complex for the one-round immediate snapshot is just the standard chromatic subdivision. An algorithm for implementing immediate snapshot from asynchronous read/write registers is given by Borowsky and Gafni [3,5].

In the *k-Set Agreement* task [6], each process starts with a private input value, communicates with the others, and then halts after choosing a private output value. Each process is required to choose some process's input, and at most k distinct values may be chosen. For brevity we use *set agreement* as shorthand for $(n+1)$-process n-Set Agreement, since this is the easiest set agreement task known to be impossible for $n+1$ processes using shared read/write memory [3,9,10,13,14].

Set agreement has a particularly simple interpretation in the simplicial model. A protocol $\mathcal{P}$, starting from an input n-simplex S^n, solves set agreement if and only if there exists a (necessarily not color-preserving) simplicial map

$$\delta : \mathcal{P}(S^n) \to S^n$$

that (1) carries each $\mathcal{P}(S^m)$ to S^m, for dimensions m in $0, \ldots, n$, and (2) does not map any n-simplex of $\mathcal{P}(S^n)$ *onto* S^n.

The *Renaming* task [1] can be formulated in several equivalent ways. Processes are issued unique input names from a large name space, and must choose unique output names taken from a smaller name space. To rule out trivial solutions, the protocol must be *anonymous* [10]): the value chosen must depend on the protocol execution, not on the specific process id. Formally, a protocol complex $\mathcal{P}(S^n)$ is *symmetric* if any permutation π of the process ids induces a simplicial map $\pi : \mathcal{P}(S^n) \to \mathcal{P}(S^n)$. All complexes considered in this paper are symmetric. Recall that the protocol has a decision function $\delta : \mathcal{P}(S^n) \to \mathcal{O}$, where $\mathcal{O}$ is the output complex. The protocol is *anonymous* if

$$\pi(\delta(\mathcal{P}(S^n))) = \delta(\pi(\mathcal{P}(S^n)))$$

that is, relabeling process ids does not change the protocol's decisions.

In the *Weak Symmetry-Breaking* (WSB) task, tasks have no input values and Boolean output values. In every execution in which all $n+1$ processes participate, at least one process decides *true* and at least one process decides *false*. Like renaming, any protocol that implements WSB must be anonymous.

Definition 1. *A task $\mathcal{M}$ is a* manifold task *if for every input m-simplex S, for m in $0, \ldots, n$, $\mathcal{M}(S)$ is a manifold of dimension m and $\mathcal{M}(\partial S) = \partial \mathcal{M}(S)$.*

The immediate snapshot task is an example of a manifold task. A manifold task is a special case of a *divided image* [2].

4 Round-by-Round Computations

We study tasks in the *round-by-round model* [8], where processes execute in a sequence of asynchronous rounds. Each round is associated with an object (which could be an immediate snapshot memory, or another object). In each round, each process applies one operation to that round's object. Once a process has finished a round, it never revisits that round's objects. The result of one round are carried to the next not though objects, but through processor-local state.

It is known that for read/write memory alone, the round-by-round model is equivalent to the usual model in which processes can access any object at any time [4,5]. We conjecture that the two models are equivalent for tasks as well, but the question remains open.

The principal attraction of the round-by-round model is that manifold tasks are closed under composition: given a set of tasks whose one-round protocol complexes are manifolds, the result of their multi-round composition is also a manifold. For example, consider the effect of iterating the immediate snapshot protocol. Running it once yields the standard chromatic subdivision, and running it multiple times yields an *iterated subdivision*, where each simplex R in $\chi^{r-1}(I)$ is replaced with a copy of $\chi(R)$, yielding the subdivision $\chi^r(I)$.

Although the protocol complex for the multi-round snapshot complex is a subdivision of the input simplex, this property is not true of multi-round manifold tasks in general. For example, the Moebius task introduced below has a boundary complex equivalent to the boundary of a subdivided simplex, but the complex interior has a "hole" (that is, non-trivial homology), so the multi-round composition of this task cannot be a subdivided simplex. Nevertheless, a multi-round composition of a manifold task is manifold task.

Lemma 1. *If single-round protocol $\mathcal{R}$ is a manifold task, and $\mathcal{A}$ is a manifold, then $\mathcal{R}(\mathcal{A})$ is a manifold, and $\mathcal{R}(\partial\mathcal{A}) = \partial\mathcal{R}(\mathcal{A})$.*

Proof. Let R^{n-1} be an $(n-1)$-simplex in $\mathcal{R}(\mathcal{A})$. There are several cases to consider. Suppose R^{n-1} is an interior simplex of $\mathcal{R}(A^n)$ for some n-simplex $A^n \in \mathcal{A}$. Because $\mathcal{R}(A^n)$ is a manifold with boundary, there exist exactly two n-simplexes R_0^n and R_1^n in $\mathcal{R}(A^n)$ such that $R^{n-1} = R_0^n \cap R_1^n$.

Suppose R^{n-1} is a simplex of $\mathcal{R}(A^{n-1})$, for some interior simplex $A^{n-1} \in \mathcal{A}$. There are exactly two n-simplexes A_0^n and A_1^n in A^n such that $A^{n-1} = A_0^n \cap A_1^n$. Because R^{n-1} is a boundary simplex of $\mathcal{R}(A_0^n)$, there exists exactly one n-simplex R_0^n in $\rho(A_0^n)$ such that $R^{n-1} \subset R_0^n$. Similarly, because R^{n-1} is a boundary simplex of $\rho(A_1^n)$, there exists exactly one n-simplex R_1^n in $\rho(A_1^n)$ such that $R^{n-1} \subset R_1^n$. It follows that there exist exactly two n-simplexes R_0^n and R_1^n in $\rho(A^n)$ such that $R^{n-1} = R_0^n \cap R_1^n$.

Suppose R^{n-1} is a simplex of $\mathcal{R}(A^{n-1})$, for some boundary simplex $A^{n-1} \in \mathcal{A}$. Because A^{n-1} is a boundary simplex, there is exactly one n-simplex A^n in $\mathcal{A}$ such that $A^{n-1} \subset A^n$. Because R^{n-1} is a boundary simplex of $\mathcal{R}(A^n)$, there exists exactly one n-simplex R^n in $\mathcal{R}(A^n)$ such that $R^{n-1} \subset R^n$. It follows that R^{n-1} is a boundary simplex of $\mathcal{R}(\mathcal{A})$. The last item implies that $\mathcal{R}(\partial\mathcal{A}) \subseteq \partial\mathcal{R}(\mathcal{A})$.

If $R^{n-1} \in \partial\mathcal{R}(\mathcal{A})$, then $R^{n-1} \in \partial\mathcal{R}(A^n)$, for some A^n in $\mathcal{A}$. Because $\mathcal{R}$ is a manifold task, $\partial\mathcal{R}(A^n) = \mathcal{R}(\partial A^n)$ R^{n-1} is also in $\mathcal{R}(\partial A^n)$, and therefore $\partial\mathcal{R}(\mathcal{A}) \subseteq \mathcal{R}(\partial\mathcal{A})$.

It remains to check $\mathcal{R}(\mathcal{A})$ is strongly connected. Pick two n-simplexes R_0 and R_1 of $\mathcal{R}(\mathcal{A})$. Let $R_0 \in \mathcal{R}(A_0)$ and $R_1 \in \mathcal{R}(A_1)$. Because $\mathcal{A}$ is a manifold, it is strongly connected, and there exists a chain of n-simplexes $B_0, \ldots, B_k$ such that $A_0 = B_0$, $A_1 = B_k$, and each B_i and B_{i+1} share an $(n-1)$-dimensional face. We argue by induction on k. When k is zero, the result follows because $\mathcal{R}(A_0) = \mathcal{R}(A_1)$ is a manifold.

Assume the result for chains of length $k-1$. Let S be an $(n-1)$-simplex on the common boundary of $\mathcal{R}(A_{k-1})$ and $\mathcal{R}(A_k)$, and let S_0 (S_1) be the unique n-simplex in $\mathcal{R}(A_{k-1})$ ($\mathcal{R}(A_k)$) having S as a face. By the induction hypothesis, there exists a chain from R_0 to S_0, and from S_1 to R_1. Since S_0 and S_1 share a common $(n-1)$-dimensional face, we have constructed a chain from R_0 to R_1. □

Corollary 1. *The result of composing manifold tasks is a manifold task.*

5 Structural Results

5.1 Set Agreement

Theorem 1. *No manifold task can solve set agreement.*

Proof. Let $\mathcal{M}$ be a manifold task, and let S^n be an input n-simplex. A *Sperner labeling* is defined as follows[1]. Define a map carrying each vertex of $\mathcal{M}(S^n)$ to a vertex of S^n. For each face S^m of S^n, where the dimension m ranges from 0 to n, map each vertex of $\mathcal{M}(S^m)$ with one of the vertexes in S^m. Sperner's Lemma states that there must exist at least one n-simplex in $\mathcal{M}(S^n)$ that maps *onto* S^n (that is, maps to $n+1$ distinct vertexes).

Now suppose we can solve set agreement by some repeated composition of $\mathcal{M}$ and one-round immediate-snapshot protocols. By Corollary 1, the resulting complex is itself a manifold. The decision map δ induces a Sperner labeling: for each face S^m of S^n, for m in $0, \ldots, n$, δ carries each vertex of $\mathcal{M}(S^m)$ to a vertex of S^m. It follows that there is some n-simplex of $\mathcal{M}(S^n)$ that maps on to every vertex of S^n, contradicting the definition of the set agreement task. □

5.2 Renaming vs Weak Symmetry-Breaking

Lemma 2. *Renaming implements Weak Symmetry-Breaking.*

Proof. The $n+1$ processes call a Renaming protocol to choose unique names in the range $0, \ldots, 2n-1$. Each process then chooses *true* if its chosen name is even, and *false* if odd. At least process must choose *true* and at least one *false*. This protocol is anonymous. □

Lemma 3. *Weak Symmetry-Breaking implements Renaming.*

Proof. Attiya et al. [1] give a renaming algorithm for $n+1$ processes and $2n+1$ names with the property that if $k \leq n+1$ processes actually participate, then they are assigned names at most $2k-1$ names. Consider the algorithm illustrated in Figure 2. Create two instances of this renaming algorithm: R_0 and R_1. The processes first execute a weak symmetry-breaking task. The $k > 0$ processes that decide *true* each take a name from R_0, choosing at most $2k-1$ names in the range $0, \ldots, 2k-2$. The other $n-k+1$ processes that decide *false* each take an intermediate name from R_1, choosing at most $2n-2k+1$ names in the range $0, \ldots, 2n-2k$. Each process in the second group chooses a name by subtracting its intermediate name from $2n$, yielding a name in the range $2n-2k-1, \ldots, 2n-1$. Since these ranges do not overlap, this algorithm solves renaming for $n+1$ processes and $2n$ names. □

[1] Other authors call this construct a *Sperner coloring*, but we have already used "coloring" to mean labeling with process ids.

```
// data fields
WSB wsb = new WSB();
Renaming R0 = new Renaming();  // renaming task instance
Renaming R1 = new Renaming();  // renaming task instance
// algorithm
int chooseName(int n) {        // there are n+1 processes
  boolean side = wsb.choose();
  int name;
  if (side) {
    return R0.choose();
  } else {
    return (2*n) - R1.choose();
  }
}
```

Fig. 2. Implementing Renaming from WSB

Corollary 2. *Renaming is equivalent to Weak Symmetry-Breaking.*

5.3 Set Agreement vs Weak Symmetry-Breaking

We now show set agreement solves weak symmetry-breaking. Assume we have a protocol that solves set agreement. We also need an instance of the Attiya et al. [1] protocol that assigns to each of $n + 1$ processes a unique name in the range $0, \ldots, 2n + 1$. This protocol uses only read/write memory, and is anonymous. Consider the protocol shown in Figure 3.

The processes first call the Attiya protocol to choose ids in the range $0, \ldots, 2n - 1$. This step ensures that the protocol as a whole is anonymous. The processes that choose

```
int[2][n] output; // two (n+1)-element arrays, initially -1
ABDPR rename;     // ABDPR renaming algorithm
SetAgree sa[2];   // set agreement protocol objects
boolean choose(int me) {
  int id = rename.choose(me); // anonymous id
  if (id < n+1) {
    output[0][id] = sa[0].decide(id);
    foreach (int i in output[0]) {
      if (i == id) return true;
    }
    return false;
  } else {
    output[1][id] = sa[1].decide(id);
    foreach (int i in output[1]) {
      if (i == id) return false;
    }
    return true;
}
```

Fig. 3. Implementing WSB from set agreement

names in the range $0, \dots, n$ call the first set agreement object's decide method, using its own anonymous id as input. The process then reads through the array, returning *true* if it finds its own id, and *false* otherwise. The processes that choose names in the range $n+1, \dots, 2n-2$ do the same using the other set agreement object and the other output array, except that they return inverted values.

Lemma 4. *Some process decides* true.

Proof. At least one process goes to the first set agreement object. Of these, the process whose id is first to be written to the output array decides *true*. □

Lemma 5. *Some process decides* false.

Proof. There are two cases to consider: (1) all processes go to the first set-agreement, or (2) some go to the first and some to the second. In the first case, if all processes decide *true*, then all $n + 1$ inputs were chosen as decision values, violating the Set Agreement specification. In the second case, the processes that go to the second set-agreement object, the process whose id is first to be written decides *false*. □

Corollary 3. *Set Agreement solves Weak Symmetry-Breaking.*

This result shows that any task that solves Set Agreement also solves renaming, ruling out the possibility that the two tasks are incomparable.

We next introduce a new task, called the *Moebius* task, because in two dimensions its one-round protocol complex is a Moebius strip. (Of course, in higher dimensions, the complex is not a Moebius strip, although it is still a non-orientable manifold with boundary.)

This task is defined in even dimensions, so let $n = 2N$. Consider a system of processes, $P_0, \dots, P_{2N}$. Let $S_0, \dots, S_{2N}$ be $2N + 1$ disjoint $(2N)$-simplexes, and let S_{ij} be the face of S_i opposite vertex P_j.

Let $X_i = \chi(S_i)$, the standard chromatic subdivision of S_i, and let $X_{ij} = \chi(X_{ij})$.

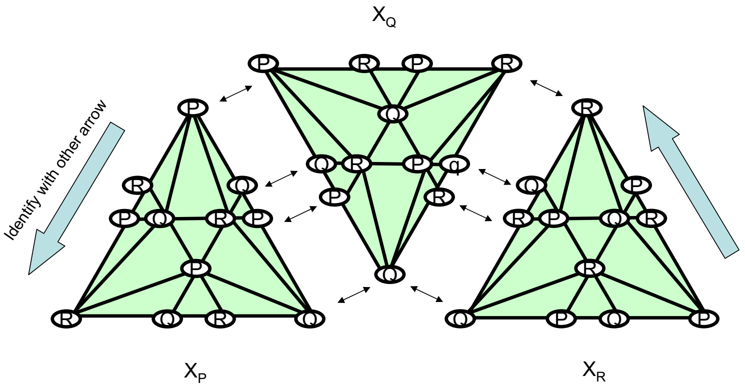


Fig. 4. One-round Moebius task protocol complex for 3 processes)

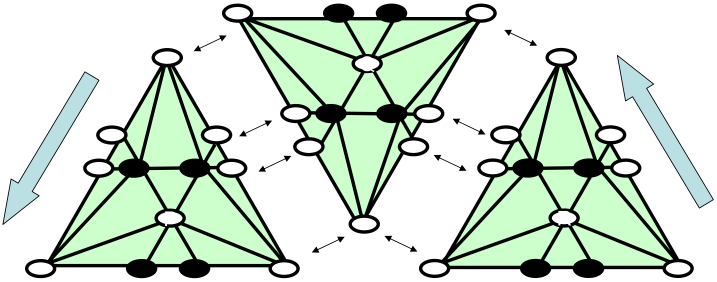


Fig. 5. Weak Symmetry-breaking from one-round Moebius task protocol

We call X_{ii} the *external face* of X_i (even though it is technically a complex), and the X_{ij}, for $i \neq j$, the *internal faces*.

Let U_j be the set of process ids in X_{ij}: 0 to $2N + 1$ except for j. Let $\pi_j : U_j \to U_j$ send the index with rank r in U_j to the index with rank $r + N \bmod 2N$.

Identify each internal face X_{ij} with $X_{\pi\ (i)j}$. Because $\pi_j(\pi_j(i)) = i$, each $(2N - 1)$-simplex in an external face lies in exactly two $(2N)$-simplexes, so the result is a manifold. (This also why the construction only works in even dimensions.)

The Moebius task itself is defined as follows. Let S^n be the input n-simplex. The Moebius complex $\mathcal{M}(S^n)$ is defined so that its boundary complex $\partial\mathcal{M}(S^n)$ is a subdivision of the boundary complex ∂S^n of the input simplex. For each proper face S^m of S^n, for $m < n$, the task specification map Δ carries S^m to the m-simplexes in the unique face of $\partial\mathcal{M}(S^n)$ with the same set of process ids. Finally, Δ maps S^n to all n-simplexes of $\mathcal{M}(S^n)$.

Theorem 2. *The Moebius task cannot solve Set Agreement.*

Proof. The one-round Moebius task is a manifold task, so composing the Moebius task with itself, with read/write rounds, or with any other manifold task yields a manifold task (Corollary 1). So by Corollary 1, the Moebius task cannot solve Set Agreement. □

To show this task solves weak symmetry breaking, we color the edges with black and white pebbles (that is, *true* or *false* values) so that no simplex is monochrome. For the central simplex of each X_i, color each node black except for the one labeled with P_i. For the central simplex of each external face X_{ii}, color the central $(2N - 2)$-simplex black. The rest are white.

Every $(2N - 1)$-simplex X in X_i intersects both a face, either internal or external, and a central $(2N - 1)$-simplex. If X intersects an internal face, then the vertexes on that face are white, and the vertexes on the central simplex are black. If X intersects the internal face, then it intersects the white node of the central simplex of X_i, and a black node of the central simplex of X_{ii}.

Theorem 3. *Set agreement implements renaming but not vice-versa.*

Proof. Set agreement solves weak symmetry-breaking (Corollary 3) which solves renaming (Corollary 2).

On the other hand, if renaming solves set agreement, then so does WSB, and so does the Moebius task, contradicting Theorem 1. □

References

1. Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
2. Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002.
3. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 1993 ACM Symposium on Theory of Computing*, pages 206–215, May 1993.
4. Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 41–51, New York, NY, USA, 1993. ACM Press.
5. Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 189–198, New York, NY, USA, 1997. ACM Press.
6. S. Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings Of The Ninth Annual ACM Symposium On Principles of Distributed Computing*, pages 311–234, August 1990.
7. M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.
8. Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152, New York, NY, USA, 1998. ACM Press.
9. Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 111–120, New York, NY, USA, 1993. ACM Press.
10. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
11. M.P. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, January 1991.
12. J.R. Munkres. *Elements Of Algebraic Topology*. Addison Wesley, Reading MA, 1984. ISBN 0-201-04586-9.
13. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the 1993 ACM Symposium on Theory of Computing*, pages 101–110, May 1993.
14. Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

Exact Distance Labelings Yield Additive-Stretch Compact Routing Schemes

Arthur Brady and Lenore Cowen

Department of Computer Science, Tufts University, Medford, MA, USA
{abrady, cowen}@cs.tufts.edu

Abstract. Distance labelings and compact routing schemes have both been active areas of recent research. It was already known that graphs with constant-sized recursive separators, such as trees, outerplanar graphs, series-parallel graphs and graphs of bounded treewidth, support both exact distance labelings and optimal (additive stretch 0, multiplicative stretch 1) compact routing schemes, but there are many classes of graphs known to admit exact distance labelings that do not have constant-sized separators. Our main result is to demonstrate that *every* unweighted, undirected n-vertex graph which supports an exact distance labeling with $l(n)$-sized labels also supports a compact routing scheme with $O(l(n) + \log^2 n/\log\log n)$-sized headers, $O(\sqrt{n}(l(n) + \log^2 n/\log\log n))$-sized routing tables, and an additive stretch of 6.

We then investigate two classes of graphs which support exact distance labelings (but do not guarantee constant-sized separators), where we can improve substantially on our general result. In the case of interval graphs, we present a compact routing scheme with $O(\log n)$-sized headers, $O(\log n)$-sized routing tables and additive stretch 1, improving headers and table sizes from a result of [1], which uses $O(\log^3 n/\log\log n)$-bit headers and tables. We also present a compact routing scheme for the related family of circular arc graphs which guarantees $O(\log^2 n)$-sized headers, $O(\log n)$-sized routing tables and an additive stretch of 1.

1 Introduction

According to the usual representation of a graph, vertices are assigned labels (say 1 to n for an n-vertex graph) in an arbitrary way, in the sense that the names are just $\log n$ bit placeholders for the rows and columns of the adjacency matrix (or alternatively, the list of edges), used to encode the structure of the graph. Breuer and Folkman [2] introduced the problem of determining which classes of graphs support the assignment of short vertex labels (i.e. $O(\log n)$ or $O(\log^c n)$ bits, c constant) so that, given only the labels of vertices i and j, it can be inferred whether or not i and j are adjacent. David Peleg [3] introduced the notion of distance labelings by generalizing this question: that is, he asked if can we assign short labels to the vertices of a graph so that the *distance* between vertices i and j can be computed just from the labels of i and j.

Definition 1.1. *Given an undirected graph $G = (V, E)$, a* ***distance labeling on*** *G (over some string alphabet Σ) is a function $DL : V \longmapsto \Sigma^*$ which assigns*

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 339–354, 2006.

*a string to each vertex $v \in V$ in such a way that for any two vertices $v,\ w \in V$, a function dist can be computed so that dist$(DL(v),\ DL(w))$ is the distance between v and w in G. We refer to the string $DL(v)$ assigned to a vertex v as v's **distance label**, and we refer to the computation of dist$(DL(v),\ DL(w))$ as a **distance query**.*

Notice that our definition bounds neither the size of the distance labels nor the time required to compute the distance function (once the labels have been generated). For these structures to be of any practical use, we would prefer to find a distance labeling which guarantees small labels (say $O(\log^c n)$ bits for some small constant c) and fast computation times. Unfortunately, not all classes of graphs enjoy distance labelings which have these properties. For example, it's known [4] that any distance labeling on general graphs must use $\Theta(n)$-bit labels, and any distance labeling on planar graphs must use $\Omega(n^{1/3})$-bit labels. On the other hand, distance labelings meeting these criteria have been discovered for several well-studied classes of graphs. The distance labeling on trees given in [3] uses $O(\log^2 n)$-bit labels and answers distance queries in polylogarithmic time. In fact, it was shown in [4] that the size of the labels in this result matches the lower bound for any distance labeling on trees.

Much of the existing work on distance labelings for particular graph families has relied on the existence of *separators* for these families. It was shown in [4] that any n-node graph with a recursive separator of size $f(n)$ supports a distance labeling which uses labels of size $O(f(n) \cdot \log^2 n)$. Trees, series-parallel graphs, and graphs of bounded treewidth all have constant-sized separators, yielding distance labelings on these graph families which use $O(\log^2 n)$-bit labels. This idea was extended in [5] to construct a distance labeling using $O(g(n) \cdot \log n)$-bit labels for any graph with an $f(n)$-sized separator. In this case, $f(n)$ may be large, but $g(n)$ – which is derived from certain structural properties of the separator – may be quite small (see [5] for full discussion and details). The authors went on to demonstrate that the families of *interval graphs* and *permutation graphs* exhibit separators for which $g(n) = O(\log n)$, yielding distance labelings for these graphs with $O(\log^2 n)$-bit labels. The result for interval graphs was improved in [6], which gives a labeling scheme which guarantees $O(\log n)$-bit labels (for both interval and circular arc graphs), and the scheme of [7] improves upon the previous result for permutation graphs, also using $O(\log n)$-bit labels.

1.1 Compact Routing

Consider a communications network modeled as a connected undirected graph $G = (V, E)$, $|V| = n$, with network nodes represented as vertices and direct communications links represented as edges.

A *routing scheme* R is a distributed algorithm defined on G which guarantees that any vertex v can send a message M to any other vertex w (along some (v, w)-path P in G), using metadata stored in M along with information stored locally at each vertex along P.

We refer to the metadata stored in a message M as M's *header,* and to the local information stored at a vertex v as v's *routing table.* Given an input graph

$G = (V, E)$, a routing scheme R must specify the construction of the routing table at each vertex $v \in V$, the construction of the header of any message M originating at a given source v and intended for a given destination w, and a deterministic forwarding function $F(\mathit{table}(x),\ \mathit{header}(M))$. This function, when given the information in the routing table of a vertex x and the information in M's header, possibly rewrites the header, then selects an edge adjacent to x along which to forward M.

F is known as R's *routing function.* Given a source vertex v, a destination vertex w and a message M, the sequence of vertices $\langle v = v_0,\ v_1,\ \ldots,\ v_k\rangle$ defined by successive applications of $F(\mathit{table}(v_i),\ \mathit{header}(M))$ must be such that k is finite and $v_k = w$. We refer to this (v, w)-path as the *route* P_{vw} from v to w with respect to R. If F is allowed to alter the header of a message at intermediate vertices between its source and destination (while maintaining bounds on the size of the header), we refer to R as supporting *writeable packet headers.* Our general result requires writeable packet headers; our schemes for interval graphs and circular arc graphs do not. All our results are in the *name-dependent* model of compact routing (see [8] for discussion), and hold for unweighted, undirected graphs.

Definition 1.2. *The* ***multiplicative stretch*** *of a routing scheme R with respect to a graph G is $\max\limits_{v,w \in V(G)} \frac{P}{d(v,w)}$, where P_{vw} is defined as above and $d(v, w)$ is the length of a shortest (v, w)-path in G. The* ***additive stretch*** *of a routing scheme R with respect to a graph G is $\max\limits_{v,w \in V(G)} |P_{vw} - d(v, w)|$.*

Additive stretch has been studied in the context of graph spanners (cf. [9,10,11]), and is referred to as "deviation" in the work of [1,12].

We assume that each vertex in G is arbitrarily assigned a unique $\log n$-bit **ID** $\in \{1 \ldots n\}$, and that these IDs are provided along with the input graph, although message headers will contain additional information, such as distance labels, so we are considering compact routing in the *name-dependent* model (cf. [8]). We consider the *fixed-port* routing model, in which each vertex v has locally assigned an arbitrary $\log n$-bit **port name** $\mathit{port}_v(w) \in \{1 \ldots d(v)\}$ to each of its adjacent edges (v, w), and that the retrieval of this port name is the only mechanism by which v can forward a message along edge (v, w).

The field of *compact routing* is concerned with creating routing schemes which minimize worst-case routing table size, header size and stretch. In particular, we would like header sizes to be polylogarithmic in n, table sizes to be sublinear, and stretch to be bounded by a constant.

Most work in compact routing to date has dealt only with *multiplicative* stretch. Our main result is the following theorem:

Theorem 1.3. *Let $G = (V, E)$ with $|V| = n$ have an exact distance labeling with $O(l(n))$-sized distance labels. Then there exists a compact routing scheme for G which uses $O(l(n) + \log^2 n / \log\log n)$-sized headers, $O(\sqrt{n} \cdot (l(n) + \log^2 n / \log\log n))$-sized routing tables, and has an* ***additive*** *stretch of 6.*

We prove this theorem in Sect. 2. This immediately implies compact routing schemes for interval graphs and the related family of circular arc graphs [6], as well as for permutation graphs [7], with $O(\log^2 n/ \log\log n)$-sized headers, $O(\sqrt{n}\cdot \log^2 n/ \log\log n)$-sized routing tables and an additive stretch of 6, by plugging the known exact distance labeling schemes for these graphs into Theorem 1.3. We note that for interval graphs and permutation graphs, better schemes were already known: Dragan, Yan and Corneil [12] present compact routing schemes for these graphs with $O(\log n)$-bit routing tables and message headers, and an additive stretch of 2, while Dragan and Lomonosov [1] present a compact routing scheme for interval graphs with $O(\log^3 n/ \log\log n)$-bit headers and tables, and an additive stretch of 1. In Sects. 3 and 4, we show that for interval graphs and circular arc graphs, we can exploit the structural properties of these graphs to do better. In Sect. 3, we introduce a compact routing scheme for interval graphs which guarantees an additive stretch of 1 and uses $O(\log n)$-bit headers and routing tables. In Sect. 4, we give a compact routing scheme for circular arc graphs which also guarantees an additive stretch of 1, using $O(\log^2 n)$-bit headers and $O(\log n)$-bit routing tables.

2 Proof of Theorem 1.3

Definition 2.1. *Given any graph $G = (V, E)$, the* ***square*** *$G^2 = (V, E')$ of G is the graph which places an edge between u and v if $d(u, v) \leq 2$ in G.*

Fact 2.2. [13,14] *There exists an exact (i.e. additive stretch 0) compact routing scheme for n-vertex* ***trees*** *which uses $O(\log^2 n/ \log\log n)$-bit headers and $O(\log^2 n/ \log\log n)$-bit routing tables.*

2.1 Algorithm

Given a connected, unweighted, undirected n-vertex graph $G = (V, E)$, with vertices assigned arbitrary IDs $\in \{1 \ldots n\}$, and a distance labeling $\langle DL,\ dist\rangle$ on G, which guarantees labels of size at most $l(n)$, greedily construct a set I by iteratively choosing vertices in G of degree $\geq \lceil\sqrt{n}\rceil$, in such a way that each chosen vertex v satisfies $d(v, i) \geq 3$ for each i already in I, until no more vertices can be chosen. Note that $0 \leq |I| \leq \sqrt{n}$, and that I is an independent set in G^2. Identify a single-source shortest-path tree T_i, spanning G, rooted at each vertex $i \in I$. Process each tree T_i according to the scheme referenced in Fact 2.2. We will refer to the header assigned to v by the scheme of Fact 2.2 in tree T_i as $treeHeader_i(v)$, and to the table assigned to v by this scheme as $treeTable_i(v)$.

For each vertex v and each tree T_i, store $treeTable_i(v)$ in v's routing table. We refer to the entire collection of these tree-routing tables at v as v's *local tree data*. We will refer to the fixed $\log n$-bit ID assigned to each vertex v in the input graph as $ID(v)$. All vertices store their own IDs and distance labels in their tables; in addition, for each vertex $v \in V$:

Case 1: $v \in I$. Distribute the storage of all pairs $\langle ID(w), treeHeader_v(w)\rangle$, for all vertices $v \neq w \in V$, across v's neighbors as follows: first, sort these n

pairs by $ID(w)$ and store the result in a sequence S_v. Next, choose an arbitrary sequence N_v of $\lceil\sqrt{n}\rceil$ of v's neighbors. Partition the ordered list S_v into $\lceil\sqrt{n}\rceil$ blocks, each of size $\lceil\sqrt{n}\rceil$ (except possibly the last block, which can be smaller if $\sqrt{n}$ is not an integer), and store the k^{th} such block at v's neighbor $N_v[k]$. Also at $N_v[k]$, store $port_{N\ [k]}(v)$. We refer to this collection of data stored at $N_v[k]$ as $N_v[k]$'s *dictionary data.* At v, store a triple $\langle port_v(N_v[k]), ID(w_{start}), ID(w_{end})\rangle$ for each of v's $\lceil\sqrt{n}\rceil$ neighbors in N_v, where w_{start} and w_{end} are the first and last entries in $N_v[k]$'s dictionary data, respectively. We refer to this collection of triples stored at v as v's *dictionary index.*

Case 2: $v \notin I$ and $deg(v) \geq \lceil\sqrt{n}\rceil$. In this case, select some $i \in I$ for which $d(v,i) \leq 2$. (Such an i always exists because I was constructed to be maximal in G^2.) At v, store at most two port names, containing exact shortest-path routing information from v to i. Mark v "SUBORDINATE TO i." We refer to this collection of data as v's *local link data.*

Case 3: $deg(v) < \lceil\sqrt{n}\rceil$. At v, store $\langle ID(x), DL(x), port_v(x)\rangle$, for each neighbor x of v. We refer to this collection as v's *local neighbor data.*

The header for a message destined for a vertex w will contain four elements: $ID(w)$, $DL(w)$, an integer IN_TREE $\in \{0\ \dots\ n\}$, and $treeHeader_v(w)$ for some tree T_v. Headers will initially contain only $ID(w)$ and $DL(w)$, with the other elements empty.

Routing from the current vertex v toward a destination vertex w proceeds iteratively according to several cases. If $deg(v) < \lceil\sqrt{n}\rceil$, we search v's local neighbor data for a neighbor v' of v such that $d(v,v') + d(v',w) = d(v,w)$, using the distance labels to compute this value, then route to v'. If $deg(v) \geq \lceil\sqrt{n}\rceil$ but $v \notin I$, we route (along at most two edges) to v's closest neighbor i in I using v's local link data. Lastly, if $v \in I$, we use v's dictionary index to determine which of v's neighbors has stored information for w in its block, route to that neighbor, retrieve $treeHeader_v(w)$, write it into w's message header, and route to w along T_v according to the scheme referenced in Fact 2.2.

2.2 Analysis

At any point in the iteration of the routing procedure, the header on a message destined for a vertex w contains at most $ID(w)$, $DL(w)$, one $\log n$-bit integer, and $treeHeader_v(w)$ for some $v \in I$, for a total header size of $O(l(n) + \log^2 n/\log\log n)$.

If $deg(v) < \lceil\sqrt{n}\rceil$, then v's local neighbor data contains $O(\sqrt{n})$ triples, each containing a $\log n$-bit ID, a $\log n$-bit port name, and an $O(l(n))$-bit distance label; v's local tree data contains $O(\sqrt{n})$ tree tables assigned to v by the scheme of Fact 2.2, each of size $O(\log^2 n/\log\log n)$; and lastly v also stores its own $\log n$-bit ID and its $O(l(n))$-bit distance label. If $deg(v) \geq \lceil\sqrt{n}\rceil$, then v may store a dictionary index containing $O(\sqrt{n})$ triples, each of size $O(\log n)$; local tree data of total size $O(\sqrt{n}\log^2 n/\log\log n)$; and v's own ID and distance label. Lastly, v (irrespective of $deg(v)$) may have been selected to store at most one block of dictionary data for some vertex in I, of size $O(\sqrt{n}\log^2 n/\log\log n)$.

Taken together, this gives a worst-case routing table size of $O(\sqrt{n} \cdot (l(n) + \log^2 n/\log\log n))$.

Lemma 2.3. ***(Stretch)*** *The algorithm routes messages with maximum stretch OPT+6.*

Proof. When routing from v to w, if all intermediate vertices encountered are of degree $< \lceil\sqrt{n}\rceil$ (except possibly w), then each step routes along a shortest path from the current vertex to w, yielding an optimal route overall. If an intermediate vertex h of degree $\geq \lceil\sqrt{n}\rceil$ is encountered, we consider two cases:

Case 1: The first such vertex h is in I. In this case, we look up w's header in T_h, using at most two edges to route to some neighbor of h which stores dictionary data for w and back again, then continue along a shortest path from h to w in T_h. Since h was the first vertex of degree $\geq \lceil\sqrt{n}\rceil$ to be encountered and since at each previous step we used distance labels to route optimally toward w, we have $d(v,w) = d(v,h) + d(h,w)$. Since in this case our algorithm uses a route of length at most $d(v,h) + 2 + d(h,w)$, this case yields an additive stretch of at most 2.

Case 2: The first such vertex h is not in I. In this case, we route along at most two edges to h's closest neighbor $i \in I$, route to one of i's neighbors j to obtain tree-routing information for w, and route back along at most three edges through h (possibly fewer, if j or i is closer to w), continuing along some shortest path in T_i from i to w. Since $d(v,w) = d(v,h) + d(h,w)$, since $d(i,w) \leq d(h,w) + 2$, and since T_i is a shortest-path spanning tree rooted at i, we have $P_{vw} \leq d(v,h) + d(h,i) + d(i,j) + d_T(j,w) \leq d(v,h) + 2 + 1 + [d(i,w) + 1] = d(v,h) + d(i,w) + 4 \leq d(v,h) + [d(h,w) + 2] + 4 = d(v,h) + d(h,w) + 6 = d(v,w) + 6$, for a total of at most 6 more edges than an optimal route. □

3 Compact Routing on Interval Graphs

Our general result implies the existence of a compact routing scheme for interval graphs with $O(\log^2 n/\log\log n)$-sized headers, $O(\sqrt{n}\log^2 n/\log\log n)$-sized routing tables, and an additive stretch of 6. We now show that we can do substantially better for this family of graphs, proving the following:

Theorem 3.1. *Any (unweighted, undirected) interval graph supports a compact routing scheme with $O(\log n)$-bit headers, $O(\log n)$-bit tables and an additive stretch of 1.*

Definition 3.2. *Given a finite set H of closed intervals $v = [L(v), R(v)]$ on the real line, $|H| = n$, the* ***interval graph*** *$G = (V,E)$ of H is defined by assigning one vertex to each interval v, with an edge between every pair of vertices whose corresponding intervals intersect. For a vertex $v \in V$, we define* ***L(v) (R(v))*** *to be the left (right) endpoint of the interval in H which corresponds to v, which we will refer to as v's* ***reference interval****. For a set $J \subseteq V$, we define* ***L(J)***

to be $\min_{v \in J} L(v)$ *and* ***R(J)*** *to be* $\max_{v \in J} R(v)$*. The* ***reference set*** *of a set* $J \subseteq V$ *consists of the union on the real line of all the reference intervals of vertices in* J*, with overlapping intervals combined: that is, two elements in* J*'s reference set (which correspond to two subsets of* J*) are distinct only if there is no edge between those two subsets in* G*. We say a vertex* v *(respectively, a set of vertices* J*)* ***spans*** *a given interval* i *if the reference interval of* v *(resp., the reference set of* J*) contains* i*.*

We will use the same variable to describe both a vertex in an interval graph and its reference interval on the real line, and we will similarly overload a single variable to represent both a vertex set in G and its reference set. We also only consider the case where there is no single vertex i whose reference interval spans the reference set for the whole graph, since compact routing with additive stretch 1 is otherwise simple: always route through i, since it is adjacent to every other vertex; the routing table at v contains only $port_v(i)$, and the header for w contains only $port_i(w)$.

Definition 3.3. *Given an interval graph* $G = (V, E)$*, choose a vertex* $a \in V$ *so that* $L(a) = L(V)$ *and* $R(a)$ *is maximum, and choose a vertex* $b \in V$ *so that* $R(b) = R(V)$ *and* $L(b)$ *is minimum. (Note that* $a \neq b$ *since we are assuming that no single vertex spans the reference set of* V*.) A* ***spine*** *of* G *is a sequence of vertices* $S = \langle a = s_1, \ldots, s_k = b \rangle$ *chosen so that* $R(s_i)$ *is maximum over all neighbors of* s_{i-1}*, for all* $1 < i \leq k$*, choosing* $s_k = b$ *as soon as it becomes available. Given an interval graph* $G = (V, E)$*, a spine* S *of* G *and a vertex* $v \notin S$*, we refer to the set of vertices in* S *adjacent to* v *as* v*'s* ***landmarks****.*

Note that every $v \notin S$ has at least one and at most three landmarks; this follows from the fact that the reference set of S is equal to that of V, combined with the fact that S is a shortest path from a to b (cf. Lemma 3.5).

Given an interval graph $G = (V, E)$ with a spine S, consider a pair of vertices a and b where $R(a) < L(b)$, i.e. a is to the left of b on the real line. Consider a sequence $P = \langle a = p_0, p_1, \ldots, p_l = b \rangle$ describing a shortest path from a to b in G. Since $R(a) < L(b)$, there must be some vertex $p \in P$ for which $R(p) > R(a)$. Notice that p_1 is the first such vertex (if it weren't, we could delete vertices from P to form a shorter path). The following lemma is then immediate by induction:

Lemma 3.4. *For all* $p_i \in P$*,* $0 \leq i < l$*,* $R(p_{i+1}) > R(p_i)$*.* □

Lemma 3.5. *Let* $T = \langle a = t_0, t_1, \ldots, t_k = b \rangle$ *be a sequence constructed iteratively by setting* t_0 *to* a*, setting an index* i *to 0, and while* t_i *is not adjacent to* b*, setting* t_{i+1} *to be a neighbor of* t_i *for which* $R(t_i)$ *is maximum, incrementing* i *at each step. Then* T *is a sequence describing a shortest path from* a *to* b*.*

Proof. Let $P = \langle a = p_0, p_1, \ldots, p_l = b \rangle$ describe a shortest path from a to b. If $P = T$, there is nothing to prove. So assume $P \neq T$, and let p_i be the first vertex of P where $p_i \neq t_i$. Observe that $L(t_i) \leq R(p_{i-1})$, since t_i is adjacent to p_{i-1} and $R(p_{i-1}) \leq R(t_i)$ by construction. Since P describes a shortest path,

p_{i-1} cannot be adjacent to p_{i+1}, so $R(p_{i-1}) < L(p_{i+1})$. Since (by Lemma 3.4) $R(p_i) < R(p_{i+1})$, and since p_i is adjacent to p_{i+1}, we have $L(p_{i+1}) \leq R(p_i)$. Since t_i was chosen from among the neighbors of p_{i-1} so that $R(t_i)$ was maximum, we have $R(p_i) \leq R(t_i)$. Taken together, this gives $L(t_i) \leq R(p_{i-1}) < L(p_{i+1}) \leq R(p_i) \leq R(t_i)$, meaning t_i is adjacent to p_{i+1}. We can thus substitute t_i for p_i in P to form a new sequence P' which also describes a shortest path from a to b; we can then perform such a substitution for every intermediate vertex $a \neq p_i \neq b$ of P, and since p_{l-1} is adjacent to b, we have that $|T| = k = l = |P|$. □

Lemma 3.6. *If $v, w \notin S$ do not share a landmark, they are not adjacent.* □

3.1 Algorithm

The header for each vertex $w \in S$ contains $L(w)$ and $R(w)$. The header for each vertex $w \notin S$ contains $L(w), R(w)$, and for each landmark x of w, $L(x), R(x)$, and $port_x(w)$. The routing table for any vertex v is constructed as follows: if $v \in S$, let a and b be its neighbors in S to the left and right respectively (if they both exist). Store $L(v), R(v), L(a), R(a), port_v(a), L(b), R(b)$, and $port_v(b)$. If $v \notin S$, store $L(v)$ and $R(v)$, and for each landmark x of v, store $L(x), R(x)$, and $port_v(x)$. Both v's table and header contain the (fixed) ID assigned in the input graph, plus an extra bit, signaling whether or not v is a member of S.
To route from v to w:

Case 1: $v, w \in S$. If v and w are adjacent, then use the information in v's routing table to route along $port_v(w)$ to w. Otherwise, compare the information in w's header (containing w's endpoints) with the information in v's table (containing v's endpoints) to determine whether w lies to the left or to the right of v on the real line, and route along S in the appropriate direction, using the adjacency information in v's routing table and in the tables of all intermediate vertices in S.

Case 2: $v, w \notin S$. If v and w share a landmark a, retrieve $port_v(a)$ from v's table and $port_a(w)$ from w's header, and route to w along those two edges. If v and w do not share a landmark, then they are not adjacent by Lemma 3.6. Say v is to the left of w on the real line. Let x be v's rightmost landmark, and let y be w's leftmost landmark. Retrieve $port_v(x)$ from v's routing table and route along that edge to x, then from x to y along S, and finally retrieve $port_y(w)$ from w's header, and route along that edge from y to w. If v is to the right of w on the real line, route analogously, reversing all directions.

Case 3: $v \in S, w \notin S$. If v is one of w's landmarks, retrieve $port_v(w)$ from w's header and route along that edge. If v is not one of w's landmarks, but is adjacent to one of w's landmarks x, route to x using the information in v's table, then to w using the information in w's header. Finally, if v is adjacent neither to w nor to any of w's landmarks, say that v lies to the left of w's landmarks' reference set. Route from v to w's leftmost landmark a, along S, using the routing tables of v and all intermediate vertices in S, then retrieve $port_a(w)$ from w's header and route along that edge. If v lies to the right of w's landmarks' reference set, route analogously, reversing all directions.

Case 4: $v \notin S, w \in S$. If w is one of v's landmarks, retrieve $port_v(w)$ from v's table and route along that edge. If w is not one of v's landmarks, but is adjacent to one of v's landmarks x, route to x using the information in v's table, then to w using the adjacency information in x's table. Finally, if w is adjacent neither to v nor to any of v's landmarks, say that w lies to the right of v's landmarks' reference set. Let x be v's rightmost landmark; route to x using the information in v's table, then along S to w using the routing tables of x and all intermediate vertices in S. If w lies to the left of v's landmarks' reference set, route analogously, reversing all directions.

3.2 Analysis

Despite the fact that we have defined interval graphs for sets of intervals on the real line, we assume for the sake of convenience that the endpoints of each reference interval are $O(\log n)$-bit integers; since we're dealing with finite sets, a straightforward scaling process can ensure that this is the case. Each header contains at most 12 $O(\log n)$-bit integers plus one bit, for a total asymptotic header size of $O(\log n)$. Each routing table also contains at most 12 $O(\log n)$-bit integers plus one bit, for a total asymptotic table size of $O(\log n)$.

Lemma 3.7. *Given an interval graph $G = (V, E)$, let I be the connected interval on the real line which is the reference set of V. Given any interval C which is a subset of I, let A be the set of vertices in V whose reference intervals contain $L(C)$, and let B be the set of vertices in V whose reference intervals contain $R(C)$. Let l denote the length of a shortest path between any vertex in A and any vertex in B. Then given the subsequence of S defined by $\langle s_0, s_1, \ldots, s_k\rangle$, where s_0 is the vertex of S containing $L(C)$ for which $R(s_0)$ is maximum, and s_k is the vertex of S containing $R(C)$ for which $R(s_k)$ is minimum, $k \leq l + 1$.*

Proof. Let I be defined as above, and let C be any closed interval contained in I, with $a = L(C)$ and $b = R(C)$. Let $P = \langle p_0, p_1, \ldots, p_l\rangle$ be a path from some vertex p_0 containing a to some vertex p_l containing b such that $|P| = l$ is minimized. Let s_0 be the vertex of S containing a for which $R(s_0)$ is maximum, and let s_k be the leftmost vertex of S containing b, where $k = d(s_0, s_k)$.

Case 1: s_0 is adjacent to p_1, and p_{l-1} is adjacent to s_k. In this case, the result is immediate: the sequence $\langle s_0, p_1, , \ldots, p_{l-1}, s_k\rangle$ represents a path from s_0 to s_k of length l, and the subsequence $\langle s_0, \ldots, s_k\rangle$ of S represents a shortest path from s_0 to s_k, so $k \leq l$.

Case 2: s_0 is not adjacent to p_1, but p_{l-1} is adjacent to s_k. In this case, notice that s_0 is adjacent to p_0 since they both contain a. The path represented by the sequence $\langle s_0, p_0, p_1, \ldots, p_{l-1}, s_k\rangle$ is of length $l + 1$, and since $d(s_0, s_k) = k$, we have that $k \leq l + 1$.

Case 3: s_0 is adjacent to p_1, but p_{l-1} is not adjacent to s_k. Using an argument analogous to that of Case 2, we again have $k \leq l+1$.

Case 4: s_0 is not adjacent to p_1, and p_{l-1} is not adjacent to s_k. Since s_1 was chosen from among the neighbors of s_0 such that $R(s_1)$ was maximum, we know that since p_0 is a neighbor of s_0, $R(s_1) \geq R(p_0)$, hence s_1 is adjacent to p_1. Similarly, s_i is adjacent to p_i for all $0 \leq i \leq \min\{k,l\}$, and $\min\{k,l\} = l$, since l was chosen to be minimum. Consider s_l, which is adjacent to p_l. If $l = k$, there is nothing to prove. Otherwise, since s_{l+1} was chosen from among the neighbors of s_l so that $R(s_{l+1})$ was maximum, and since p_l is a neighbor of s_l, we have that $R(s_{l+1}) \geq R(p_l)$, so s_{l+1} must contain b, hence $k \leq l+1$. □

Lemma 3.8. ***(Stretch)*** *Given any connected interval graph $G = (V,E)$, our algorithm routes messages between any two vertices $v, w \in V$ along a path P_{vw} of length $\leq d(v,w)+1$, where $d(v,w)$ is the distance between v and w in G.*

Proof

Case 1: $v, w \in S$. Since we are routing along a shortest path between v and w, $|P_{vw}| = d(v,w)$.

Case 2: $v, w \notin S$. If v and w share a landmark, then $|P_{vw}| = 2$, and since $d(v,w) \geq 1$, $|P_{vw}| \leq d(v,w)+1$. If v and w do not share a landmark, then assume without loss of generality that v is to the left of w on the real line. Let x be v's rightmost landmark, and let y be w's leftmost landmark. Since v is not adjacent to x's right-hand neighbor in S, $R(x) > R(v)$, and similarly $L(y) < L(w)$. Consider the interval $C = [R(x)+\epsilon, L(y)-\epsilon]$ on the real line, where ϵ is chosen small enough so that $R(x)+\epsilon$ lies between $R(x)$ and the next (greater) endpoint of any interval in S to the right of $R(x)$, and $L(y)-\epsilon$ lies between $L(y)$ and the next (lesser) endpoint of any interval in S to the left of $L(y)$. Let $I \subseteq V$ be the set of vertices of G which intersect C, and notice that I is a vertex cut of G separating v and w, so that $d(v,w) \geq d(v,I) + D_I + d(I,w)$, where D_I is the minimum length of a path from a vertex containing $L(C)$ to a vertex containing $R(C)$. Let $\langle s_0, s_1, \ldots, s_k \rangle$ be the sequence of vertices in S traversed by our algorithm between x and y, not including x and y. Since s_0 is the only vertex in S containing $R(x)+\epsilon$, s_0 must be the vertex in S containing $L(C)$ where $R(s_0)$ is maximum, and similarly s_k is the vertex in S containing $R(C)$ where $L(s_k)$ is minimum. By Lemma 3.7, then, $k \leq D_I + 1$. Since v is not adjacent to any vertex containing $R(x)+\epsilon$, $d(v,I) \geq 2$, and since w is not adjacent to any vertex containing $L(y)-\epsilon$, $d(I,w) \geq 2$. So we have $|P_{vw}| = 2+k+2 \leq d(v,I) + (D_I+1) + d(I,w) \leq d(v,w)+1$.

Case 3: $v \in S, w \notin S$. If v is one of w's landmarks, then $|P_{vw}| = d(v,w)$. If v is not one of w's landmarks, but is adjacent to one of w's landmarks, then $d(v,w) = 2$, and $|P_{vw}| = 2 = d(v,w)$. If neither of the above is the case, then we have $|P_{vw}| \leq d(v,w)+1$, by an analogous argument to that of Case 2, with $C = [R(v)+\epsilon, L(y)-\epsilon]$, where w's leftmost landmark y is assumed without loss of generality to be to the right of v on the real line.

Case 4: $v \notin S, w \in S$. This case is proved by an identical argument to that of Case 3, with the interval C of the final part defined to be $[R(x) + \epsilon, L(w) - \epsilon]$, where v's rightmost landmark x is assumed without loss of generality to be to the left of w on the real line. □

4 Compact Routing on Circular Arc Graphs

Definition 4.1. *Given a circle C and a set A of n closed arcs which are subsets of C, the* ***circular arc graph*** *corresponding to A is the graph $G = (V, E)$ constructed by creating one vertex $v_a \in V$ for each arc a in A, with an edge $(v_a, v_b) \in E$ if the arcs a and b corresponding to v_a and v_b intersect. We refer to C as the* ***reference circle*** *of G, to A as the* ***arc set*** *of G, and to a as the* ***reference arc*** *of v_a. For any arc $a \in A$ which is a strict subset of C, we define* $\boldsymbol{L(a) = L(v_a)}$ *to be the counterclockwise endpoint of a, and similarly we define* $\boldsymbol{R(a) = R(v_a)}$ *to be the clockwise endpoint of a. For any set J of vertices of G, the* ***reference set*** *of J consists of the union (on the reference circle) of all the reference arcs of vertices in J, with overlapping arcs combined: that is, two elements in J's reference set (which correspond to two subsets of J) are distinct only if there is no edge between those two subsets in G. Note in particular that the reference set of V is not necessarily equal to the arc set of G. If the reference set of some subset $X \subseteq V$ of vertices of G is a single connected arc which is not equal to the entire reference circle C, we define* $\boldsymbol{L(X)}$ *to be the counterclockwise endpoint of the reference set of X, and* $\boldsymbol{R(X)}$ *to be its clockwise endpoint. We say a vertex v (respectively, a set of vertices J)* ***spans*** *a given closed arc a on C if the reference arc of v (resp., the reference set of J) contains a.*

We will use the same variable to describe both a vertex in a circular arc graph G and its reference arc on G's reference circle C, and we will similarly overload a single variable to represent both a vertex set in G and its reference set on C. We can consider only the case where there is no single vertex a whose reference arc spans the entire reference circle, since compact routing with additive stretch 1 in the other case is straightforward: always route through a, since it is adjacent to every other vertex; the routing table at v contains only $port_v(a)$, and the header for w contains only $port_a(w)$. Since each component of a circular arc graph whose vertex set does not span its entire reference circle is also an interval graph, and we already have a compact routing scheme for interval graphs, we will only consider circular arc graphs whose vertex sets completely span their reference circles.

Definition 4.2. *Given a circular arc graph G with reference circle C and arc set A, and given any point x on C, the* ***plug of G corresponding to x*** *is the subset of vertices of G whose reference arcs contain x.*

Lemma 4.3. *Given a circular arc graph G with reference circle C, if there exists a plug P of G (corresponding to a point x on C) such that the reference set of P is C itself, then there exists a set of exactly two vertices in P whose reference arcs also span all of C.*

Proof. Since we have ruled out the case in which one arc may span the entire reference circle, and the reference set of P is the entire reference circle C, there must be a set of *at least* two arcs in P which spans C. To show that there is such a set containing *exactly* two arcs, first note that every arc in P must contain x by definition. Let a be the arc in P which covers the largest portion of C in the direction clockwise from x, and let b be the arc in P which covers the largest portion of C in the direction counterclockwise from x. If the reference set of $\{a, b\}$ is not equal to C, then P cannot span C, since both a and b were chosen as maximal. Hence a and b must together span C. □

Note that since all vertices in a given plug intersect at x, a plug is also a clique. According to Lemma 4.3, if the reference set of some plug P is the entire reference circle C, then there exists a pair of vertices $a, b \in P$ which together span all of C, and so each vertex in G must be adjacent to a, b, or both. Finally, we use the fact that there exists an exact distance labeling scheme for interval graphs which uses $O(\log n)$-bit labels [6].

4.1 Algorithm

Let $G = (V, E)$ be an n-vertex circular arc graph, with reference circle C and arc set A.

Case 1: There exists some plug P of G for which the reference set of P equals all of C. In this case, let a and b be a pair of arcs of P spanning C whose existence is guaranteed by Lemma 4.3. The routing table of every vertex $v \in V$ will contain at most five elements: the ID assigned to v in the input graph, one bit indicating whether or not v is adjacent to a (and, if so, $port_v(a)$), and one bit indicating whether or not v is adjacent to b (and, if so, $port_v(b)$). The header of every vertex $w \in V$ will similarly contain at most five elements: the ID assigned to w in the input graph, a bit indicating whether or not w is adjacent to a (and, if so, $port_a(w)$), and a bit indicating whether or not w is adjacent to b (and, if so, $port_b(w)$).

Case 2: There is no plug of G whose reference set equals all of C. In this case, we select an arbitrary point x on C, and let P be the plug of G corresponding to x. Since the reference set of P does not equal C, and since P is connected in G, the reference set of P is a single closed arc which is a proper subset of C. Because of this, $G[V \setminus P]$ is an interval graph. Let a and b be chosen from P such that $L(a) = L(P)$ and $R(b) = R(P)$. (If one vertex satisfies both requirements, we choose that vertex; for clarity, our language throughout the following will assume that a and b are distinct.) Compute distance labels for all vertices in $G[V \setminus P]$ using the scheme given in [6]. The header and routing table for each vertex v in $V \setminus P$ will contain the distance label assigned to v by this scheme; the header and routing table for each vertex p in P will contain a bit marking p as a member of P. Next, identify two shortest-path trees T_a and T_b spanning $G[V \setminus P]$, rooted at a and b respectively. Preprocess these trees according to the compact tree-routing scheme referenced in Fact 2.2, and append the headers and routing tables generated by this scheme, for both trees, to the headers and routing tables,

respectively, of every vertex in each tree. For each vertex $v \in V \setminus P$, add $d(v, a)$ and $d(v, b)$ to both the header and the routing table of v. Also for each vertex $v \in V \setminus P$, add the tree-routing headers of a and b, in T_a and T_b respectively, to v's local routing table. Preprocess $G[V \setminus P]$ according to the compact routing scheme for interval graphs given in Sect. 3, and append the headers and tables generated by that scheme to the headers and tables, respectively, for all vertices in $V \setminus P$. For each vertex p in P, record $port_p(a)$, $port_p(b)$, $port_a(p)$ and $port_b(p)$ in p's header and routing table. Finally, both the header and routing table of every vertex v will contain the ID assigned to it in the input graph.
Routing now proceeds as follows:

Case 1: To route from v to w, first extract adjacency information for a and b from the routing table of v and the header of w. If both v and w are adjacent to one of these (say a), route from v to a using $port_v(a)$ from v's routing table, then from a to w using $port_a(w)$ in w's header. If this is not the case (say v is adjacent only to a, and w is adjacent only to b), route from v to a using $port_v(a)$ in v's routing table, then from a to b using $port_a(b)$ in a's routing table, then from b to w using $port_b(w)$ from w's header.

Case 2: If neither v nor w is in the plug P, compute d_1, the distance between v and w in $G[V \setminus P]$, using the distance labels in v's routing table and w's header. Next extract $d(v, a)$ and $d(v, b)$ from v's routing table, extract $d(w, a)$ and $d(w, b)$ from w's header, and compute $d_2 = \min\{d(v, a), d(v, b)\} + \min\{d(w, a), d(w, b)\} + 1$. If $d_1 < d_2$, route from v to w through $G[V \setminus P]$ using the scheme of Sect. 3. Otherwise, say v is closest to a and w is closest to b. We route from v to a along T_a, using the tree-routing header for a stored in v's routing table (and the tree-routing tables of all intermediate vertices), then from a to b using $port_a(b)$ in a's routing table, and finally from b to w along T_b using the information in w's tree-routing header for T_b (and the tree-routing tables at all intermediate vertices). (If the closer of a and b to v and w is the same (say a), then route as above from v to a and from a to w.) If only v is in P, use the information in w's header to compute the closer of a or b to w (say a), route to a using $port_v(a)$ in v's routing table, then to w along T_a using the information in w's tree-routing header (and the tree-routing tables at all intermediate vertices). If only w is in P, use the information in v's routing table to compute the closer of a or b to v (say a), route to a along T_a using the tree-routing header for a in v's routing table (and the tree-routing table at each intermediate vertex), then route to w using $port_a(w)$, stored in w's header. If both v and w are in P, use $port_v(a)$ in v's routing table to route from v to a, then use $port_a(w)$ from w's header to route from a to w.

4.2 Analysis

If Case 1 above holds, the header for each vertex contains at most two bits and two port names, for a header size of $O(\log n)$. If Case 2 above holds, then in the worst case, the header for a vertex v contains a $\log n$-bit ID, an $O(\log n)$-bit distance label in $G[V \setminus P]$, two $O(\log^2 n/ \log\log n)$-bit tree-routing headers for

T_a and T_b, two $O(\log n)$-bit integers representing $d(v,a)$ and $d(v,b)$, and one $O(\log n)$-bit header from the interval graph scheme of Sect. 3, yielding a total header size of $O(\log^2 n/\log\log n)$.

If Case 1 above holds, then the routing table for each vertex contains at most one $\log n$-bit ID, two bits and two edge labels, for a table size of $O(\log n)$. If Case 2 above holds, then in the worst case, the routing table for a vertex v contains a $\log n$-bit ID, an $O(\log n)$-bit distance label in $G[V \setminus P]$, two $O(\log^2 n/\log\log n)$-bit tree-routing tables for T_a and T_b, two $O(\log n)$-bit integers representing $d(v,a)$ and $d(v,b)$, and one $O(\log n)$-bit table from the interval graph scheme of Sect. 3, yielding a total routing table size of $O(\log^2 n/\log\log n)$.

Lemma 4.4. *If Case 2 above holds and a vertex v is not in the plug P, then $d(v,P) = \min\{d(v,a), d(v,b)\}$.*

Proof. $L(a) = L(P)$ and $R(b) = R(P)$ by definition, and $v \notin P$. Observe that the reference set of any shortest path from v to any vertex in P must be an arc containing $L(P)$ or $R(P)$, say $L(P)$. If p is the vertex of P on this path containing $L(P)$ and $p \neq a$, clearly we can substitute a for p to obtain a path of equal length. The case for the crossing of $R(P)$ is analogous. □

Lemma 4.5. ***(Stretch)*** *Our algorithm routes messages from any source v to any destination w with maximum stretch OPT+1.*

Proof. If Case 1 above holds and if v and w are both adjacent to one of a or b, we route along a path of length 2, and since $d(v,w) \geq 1$, this is $\leq OPT + 1$. If Case 1 above holds and v and w are not both adjacent to a or to b, then say for example that v is adjacent to a and w to b. Then since the reference arcs of a and b span the entire reference circle C, observe that the reference arcs for v and w can't intersect, and so v and w are not adjacent in G, so $d(v,w) \geq 2$. Since we then route along a path of length 3, we have a route of length $\leq OPT + 1$. If Case 2 above holds and v and w are both in the plug P, then $d(v,w) = 1$, and we route along a path of length $\leq 2 = OPT + 1$. If Case 2 above holds and if one of v and w is in the plug P (say w), then $d(v,w) \geq d(v,P)$. Since we route from v to the closer of a or b and then along at most one edge to w, by Lemma 4.4, we route along a path of length at most $d(v,P) + 1 \leq OPT + 1$. Finally, if Case 2 above holds and if neither v nor w is in P, we consider two sub-cases:

Case 2A: Every shortest path from v to w crosses P. In this case, $d(v,w) \geq d(v,P) + d(w,P)$. Since $d_2 = d(v,P) + d(w,P) + 1$ by Lemma 4.4, we have that $d_2 \leq d(v,w) + 1$. Since every shortest path between v and w crosses P, we also know that $d_1 > d(v,w)$, since it represents the length of a path which does not contain any vertices in P. If $d_1 < d_2$, then we have $d(v,w) < d_1 < d_2$, implying (since the quantities involved are integers) that $d_2 \geq d(v,w) + 2$, a contradiction. So it must be the case that $d_2 \leq d_1$, and according to our routing algorithm, we therefore route along a path of length at most $d_2 \leq OPT + 1$.

Case 2B: No shortest path between v and w crosses P. Then $d_1 = d(v,w)$, so d_2 can't be less than d_1, since d_1 is the length of a path between v and w and it's minimum. Also, $d_2 \neq d_1$, since d_1 is minimum, d_2 is the length of a path through P, and no shortest path crosses P by assumption. So $d_1 < d_2$, and we route through $G[V \setminus P]$ using the scheme given in Sect. 3, which uses a route of length at most $OPT + 1$. □

We have now proved the following: any circular arc graph supports a compact routing scheme with $O(\log^2 n/\log\log n)$-bit headers, $O(\log^2 n/\log\log n)$-bit tables and an additive stretch of 1.

We note that substituting alternate tree-routing schemes for the one mentioned in Fact 2.2 will produce different tradeoffs in table and header size. Substituting the alternate tree-routing scheme given in [14], which guarantees $O(\log^2 n)$-bit headers and $O(\log n)$-bit routing tables, achieves the bounds stated in the abstract and Sect. 1. Substituting the tree-routing scheme of [15], which guarantees $O(\log n)$-bit headers and $O(\min\{deg(v), \sqrt{n}\}\cdot\log n)$-bit routing tables at each vertex v, yields a compact routing scheme for circular arc graphs with a header size of $O(\log n)$ and a (much larger) table size of $O(\min\{deg(v), \sqrt{n}\} \cdot \log n)$ at each vertex v.

The authors gratefully acknowledge support from NSF grant CCR-0208629.

References

1. Dragan, F.F., Lomonosov, I.: On compact and efficient routing in certain graph classes. In: Proc. 15th Annual Symp. on Algorithms and Computation (ISAAC). (2004) 240–414
2. Breuer, M.A., Folkman, J.: An unexpected result on coding the vertices of a graph. J. Mathematical Analysis and Applications **20**(3) (1967) 583–600
3. Peleg, D.: Proximity-preserving labeling schemes and their applications. In: Proc. 25th Intl. Wkshp. on Graph-Theoretic Concepts in Computer Science (WG). (1999) 30–41
4. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. J. Algorithms **53**(1) (2004) 85–112
5. Katz, M., Katz, N.A., Peleg, D.: Distance labeling schemes for well-separated graph classes. Discrete Applied Mathematics **145**(3) (2005) 384–402
6. Gavoille, C., Paul, C.: Optimal distance labeling for interval and circular-arc graphs. In: Proc. 11th Annual European Symp. on Algorithms. (2003) 254–265
7. Bazzaro, F., Gavoille, C.: Localized and compact data-structure for comparability graphs. In: Proc. 16th Annual Symp. on Algorithms and Computation (ISAAC). (2005) 1122–1131
8. Arias, M., Cowen, L., Laing, K., Rajaraman, R., Taka, O.: Compact routing with name independence. In: Proc. 15th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA). (2003) 184–192
9. Dor, D., Halperin, S., Zwick, U.: All pairs almost shortest paths. In: Proc. 37th Annual Symp. on Foundations of Computer Science (FOCS). (1996) 452–461
10. Elkin, M., Peleg, D.: (1+ϵ, β)-spanner constructions for general graphs. SIAM J. Comput. **33**(3) (2004) 608–631
11. Baswana, S., Kavitha, T., Mehlhorn, K., Pettie, S.: New constructions of (α, β)-spanners and purely additive spanners. In: Proc. 16th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA). (2005) 672–681

12. Dragan, F.F., Yan, C., Corneil, D.G.: Collective tree spanners and routing in at-free related graphs. In: Proc. 30th Intl. Wkshp. on Graph-Theoretic Concepts in Computer Science (WG). (2004) 68–80
13. Fraigniaud, P., Gavoille, C.: Routing in trees. In: Proc. 28th Intl. Colloq. on Automata, Languages and Programming (ICALP). (2001) 757–772
14. Thorup, M., Zwick, U.: Compact routing schemes. In: Proc. 13th Annual ACM Symp. on Parallel Algorithms and Architectures, ACM (2001) 1–10
15. Cowen, L.: Compact routing with minimum stretch. J. Algorithms **38**(1) (2001) 170–183

A Fast Distributed Approximation Algorithm for Minimum Spanning Trees

Maleq Khan and Gopal Pandurangan

Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA
{mmkhan, gopal}@cs.purdue.edu

Abstract. We give a distributed algorithm that constructs a $O(\log n)$-approximate minimum spanning tree (MST) in arbitrary networks. Our algorithm runs in time $\tilde{O}(D(G) + L(G, w))$ where $L(G, w)$ is a parameter called the *local shortest path diameter* and $D(G)$ is the (unweighted) diameter of the graph. Our algorithm is existentially optimal (up to polylogarithmic factors), i.e., there exists graphs which need $\Omega(D(G) + L(G, w))$ time to compute an H-approximation to the MST for any $H \in [1, \Theta(\log n)]$. Our result also shows that there can be a significant time gap between exact and approximate MST computation: there exists graphs in which the running time of our approximation algorithm is exponentially faster than the *time-optimal* distributed algorithm that computes the MST. Finally, we show that our algorithm can be used to find an approximate MST in wireless networks and in random weighted networks in almost optimal $\tilde{O}(D(G))$ time.

Keywords: Distributed Approximation Algorithm, Minimum Spanning Tree.

1 Introduction

1.1 Background and Previous Work

The distributed minimum spanning tree (MST) problem is one of the most important problems in the area of distributed computing. There has been a long line of research to develop efficient distributed algorithms for the MST problem starting with the seminal paper of Gallager et al [1] that constructs the MST in $O(n \log n)$ time and $O(|E| + n \log n)$ messages. The communication (message) complexity of Gallager et al. is optimal, but its time complexity is not. Hence further research concentrated on improving the time complexity. The time complexity was first improved to $O(n \log \log n)$ by Chin and Ting [2], further improved to $O(n \log^* n)$ by Gafni [3], and then improved to *existentially optimal* running time of $O(n)$ by Awerbuch [4]. The $O(n)$ bound is existentially optimal because there exists graphs where no distributed MST algorithm can do better than $\Omega(n)$ time. This was the state of art till the mid-nineties when Garay, Kutten, and Peleg [5] raised the question of identifying graph parameters that can better capture the complexity (motivated by "universal" complexity) of distributed MST computation. For many existing networks G, their diameter $D(G)$ (or D for short) is significantly smaller than the number of vertices n and therefore is a good candidate to design protocols whose running time is bounded in terms of $D(G)$ rather than

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 355–369, 2006.

n. Garay, Kutten, and Peleg [5] gave the first such distributed algorithm for the MST problem that ran in time $O(D(G) + n^{0.61})$ which was later improved by Kutten and Peleg [6] to $O(D(G) + \sqrt{n}\log^* n)$. Elkin [7] refined this result further and argued that the parameter called "MST-radius" captures the complexity of distributed MST better. He devised a distributed protocol that constructs the MST in $\tilde{O}(\mu(G, w) + \sqrt{n})$ time, where $\mu(G, w)$ is the "MST-radius" of the graph [7] (is a function of the graph topology as well as the edge weights). The ratio between diameter and MST-radius can be as large as $\Theta(n)$, and consequently on some inputs this protocol is faster than the protocol of [6] by a factor of $\Omega(\sqrt{n})$. However, a drawback of this protocol (unlike previous MST protocols [6,5,2,3,1]) is that it cannot detect termination in this much time (unless $\mu(G, w)$ is given as part of the input). Finally, we note that the time-efficient algorithms of [6,7,5] are not message-optimal (i.e., they take asymptotically much more than $O(|E| + n\log n)$, e.g., the protocol of [6] takes $O(|E| + n^{1.5})$ messages.

The lack of progress in improving the result of [6], and in particular breaking the $\sqrt{n}$ barrier, led to work on lower bounds for distributed MST problem. Peleg and Rabinovich [8] showed that $\tilde{\Omega}(\sqrt{n})$ is required for constructing MST even on graphs of small diameter and showed that this result establishes the asymptotic near-tight (existential) optimality of the protocol of [6].

While the previous distributed protocols deal with computing the exact MST, the next important question addressed in the literature concerns the study of distributed *approximation* of MST, i.e., constructing a spanning tree whose total weight is near-minimum. From a practical perspective, given that MST construction can take as much as $\tilde{\Omega}(\sqrt{n})$ time, it is worth investigating whether one can design distributed algorithms that run faster and output a near-minimum spanning tree. Peleg and Rabinovich [8] was one of the first to raise the question of devising faster algorithms that construct an approximation to the MST and left it open for further study. To quote their paper: "To the best of our knowledge nothing nontrivial is known about this problem...". Since then, the most important result known till date is the *hardness* results shown by Elkin [9]. This result showed that *approximating* the MST problem on graphs of small diameter (e.g., $O(\log n)$) within a ratio H requires essentially $\Omega(\sqrt{n/HB})$ time (assuming B bits can be sent through each edge in one round), i.e., this gives a time-approximation trade-off for the distributed MST problem: $T^2H = \Omega(\sqrt{n/B})$. However, not much progress has been made on designing time-efficient distributed approximation algorithms for MST. To quote Elkin's survey paper [10]: "There is no satisfactory approximation algorithm known for the MST problem". To the best of our knowledge, the only known distributed approximation algorithm for MST is given by Elkin in [9]. This algorithm gives an H-approximation protocol for the MST with running time $O(D(G) + \frac{\omega}{H-1} \cdot \log^* n)$, where ω_{max} is the ratio between the maximal and minimal weight of an edge in the input graph G. Thus this algorithm is not independent of the edge weights and its running time can be quite large.

1.2 Distributed Computing Model and Our Results

We present a fast distributed approximation algorithm for the MST problem. We will first briefly describe the distributed computing model that is used by our algorithm (as well as the previous MST algorithms [2,1,5,6,4,3,7] mentioned above) which is now standard in distributed computing literature (see e.g., the book by Peleg [11]).

Distributed computing model. We are given a network modeled as an undirected weighted graph $G = (V, E, w)$ where V is the set of nodes (vertices) and E is the set of communication links between them and $w(e)$ gives the weight of the edge $e \in E$. Without loss of generality, we will assume that G is connected. Each node hosts a processor with limited initial knowledge. Specifically, we make the common assumption that nodes have unique identity numbers (this is not really essential, but simplifies presentation) and at the beginning of the computation each vertex v accepts as input its own identity number, the identity numbers of its neighbors in G (i.e., nodes that share an edge with v), and the weights of the edges that are adjacent to v. Thus a node has only *local* knowledge limited to itself and its neighbors. The vertices are allowed to communicate through the edges of the graph G. We assume that the communication is synchronous and occurs in discrete pulses (time steps). (This assumption is not essential for our time complexity analysis. One can use a *synchronizer* to obtain the same time bound in an asynchronous network at the cost of some increase in the message (communication) complexity [11].) During each time step, each node v is allowed to send an arbitrary message of size $O(\log n)$ through each edge $e = (v, u)$ that is adjacent to v, and the message will arrive at u at the end of the current pulse. (We note that if unbounded-size messages are allowed, then MST problem can be trivially solved in $O(D(G))$ time[11].) The weights of the edges are at most polynomial in the number of vertices n, and therefore the weight of a single edge can be communicated in one time step. This model of distributed computation is called the $\mathcal{CONGEST}(\log n)$ model or simply the $\mathcal{CONGEST}$ model [11] (the previous results on distributed MST cited in Sect. 1.1 are for this model). We note that, more generally, $\mathcal{CONGEST}(B)$ allows messages of size at most $O(B)$ to be transmitted in a single time step across an edge. Our algorithm can straightforwardly be applied to this model also. We will assume $B = \log n$ throughout this paper.

Overview of results. Our main contribution is an almost existentially optimal (in both time and communication complexity) distributed approximation algorithm that constructs a $O(\log n)$-approximate minimum spanning tree, i.e., whose cost is within a $O(\log n)$ factor of the MST. The running time[1] of our algorithm is $\tilde{O}(D(G) + L(G, w))$ where $L(G, w)$ is a parameter called as the *local shortest path diameter* (we defer the definition of $L(G, w)$ to Sect. 2.2). Like the MST-radius, $L(G, w)$ depends both on the graph topology as well as on the edge weights. $L(G, w)$ always lies between 1 and n. $L(G, w)$ can be smaller or larger than the diameter and typically it can be much smaller than n or even $\sqrt{n}$ (recall that this is essentially a lower bound on distributed (exact) MST computation). In fact, we show that there exists graphs for which any distributed algorithm for computing the MST will take $\tilde{\Omega}(\sqrt{n})$ time, while our algorithm will compute a near-optimal approximation in $\tilde{O}(1)$ time, since $L(G, w) = \tilde{O}(1)$ and $D = \tilde{O}(1)$ for these graphs. Thus there exists an exponential gap between exact MST and $O(\log n)$-approximate MST computation. However, in some graphs $L(G, w)$ can

[1] We use the notations $\tilde{O}(f(n))$ and $\tilde{\Omega}(f(n))$ to denote $O(f(n) \cdot \mathit{polylog}(f(n)))$ and $\Omega(f(n)/\mathit{polylog}(f(n)))$, respectively.

be asymptotically much larger than both the diameter as well as $\sqrt{n}$. By combining the MST algorithm of Kutten and Peleg [6] with our algorithm in an obvious way we can obtain an algorithm with the same approximation guarantee but with running time $\tilde{O}(D(G) + \min(L(G,w), \sqrt{n}))$.

The parameter $L(G,w)$ is not arbitrary. We show that it captures the hardness of distributed approximation quite precisely: there exists a family of n-vertex graphs where $\Omega(L(G,w))$ time is needed by any distributed approximation algorithm to approximate MST within a H-factor ($1 \leq H \leq O(\log n)$) (cf. Theorem 5). This implies that our algorithm is existentially optimal (upto polylogarithmic factors) and in general no other algorithm can do better. We note that the existential optimality our algorithm is with respect to $L(G,w)$ instead of n as in the case of Awerbuch's algorithm [4]. Our algorithm is also existentially optimal (upto polylogarithmic factors) with respect to communication (message) complexity — takes $\tilde{O}(|E|)$ messages, since $\Omega(|E|)$ messages is clearly needed in some graphs to construct any spanning tree[12,13].

One of our motivations for this work is to investigate whether fast distributed algorithms that construct (near-optimal) MST can be given for special classes of networks. An important consequence of our result is that networks with low $L(G,w)$ value (compared to $O(D(G))$ admit a $\tilde{O}(D(G))$ time $O(\log n)$-approximate distributed algorithm. In particular unit disk graphs have $L(G,w) = 1$. Unit disk graphs are commonly used models in wireless networks. We also show $L(G,w) = O(\log n)$ with high probability in any arbitrary network whose edge weights are chosen independently at random from an arbitrary distribution (cf. Theorem 7).

2 Distributed Approximate MST Algorithm

2.1 Nearest Neighbor Tree Scheme

The main idea of our approach is to construct a spanning tree called as the *Nearest Neighbor Tree (NNT)* efficiently in a distributed fashion. In our previous work [14], we introduced the Nearest Neighbor Tree and showed that its cost is within a $O(\log n)$ factor of the MST. The scheme used to construct a NNT (henceforth called *NNT scheme*) is as follows: (1) each node first chooses a unique identity number called *rank* and (2) each node (except the one with the highest rank) connects (via the *shortest path*) to the *nearest* node of higher rank. We showed that the NNT scheme constructs a spanning subgraph in any weighted graph whose cost is at most $O(\log n)$ times that of the MST, irrespective of how the ranks are selected (as long as they are distinct) [14]. Note that cycles can be introduced in step 2, and hence to get a spanning tree we need to remove some edges to break the cycles. Our NNT scheme is based on the approximation algorithm for the *traveling salesman problem* (coincidentally called Nearest Neighbor algorithm) analyzed in a classic paper of Rosenkrantz, Lewis, and Stearns [15]. Imase and Waxman [16] also used a scheme based on [15] (their algorithm can also be considered a variant of NNT scheme) to show that it can maintain a $O(\log n)$-approximate Steiner tree dynamically (assuming only node additions, but not deletions.) However, their algorithm will not work in a distributed setting (unlike our NNT scheme) because one cannot connect to the shortest node (they can do that since the nodes are added one by one) as this can introduce cycles. The approach needed for distributed implementation is very different (cf. Sect. 2.3).

The advantage of NNT scheme is this: each node, individually, has the task of finding its own node to connect, and hence no explicit coordination is needed between nodes. However, despite the simplicity of the NNT scheme, it is not clear how to efficiently implement the scheme in a general weighted graph. In our previous work [14], we showed NNT scheme can be implemented in a *complete metric* graph G (i.e., $D(G) = 1$). Our algorithm takes only $O(n \log n)$ messages to construct a $O(\log n)$-approximate MST as opposed to $\Omega(n^2)$ lower bound (shown by Korach et al [17]) needed by any distributed MST algorithm in this model. If time complexity needs to be optimized, then NNT scheme can be easily implemented in $O(1)$ time (using $O(n^2)$ messges), as opposed to the best known time bound of $O(\log \log n)$ for (exact) MST [18]. These results suggest that NNT scheme can yield faster and communication-efficient algorithms compared to the algorithm that compute the exact MST. However, efficient implementation in a general weighted graph is non-trivial and was left open in [14]. Thus, a main contribution of this paper is giving an efficient implementation of the scheme in a general network. The main difficulty is in efficiently finding the nearest node of higher rank in a distributed fashion because of congestion (since many nodes are trying to search at the same time) and avoiding cycle formation. We use a technique of "incremental" neighborhood exploration that avoids congestion and cycle formation and is explained in detail in Sect. 2.3.

2.2 Preliminaries

We use the following definitions and notations concerning an undirected weighted graph $G = (V, E, w)$. We say that u and v are neighbors of each other if $(u, v) \in E$.

Notations:

$|Q(u, v)|$ or simply $|Q|$ — is the number of edges in path Q from u to v.

$w(Q(u, v))$ or $w(Q)$ — is the weight of the path Q, which is defined as the sum of the weights of the edges in path Q, i.e., $w(Q) = \sum_{(x,y)\in Q} w(x, y)$. $P(u, v)$ — is a shortest path (in the weighted sense) from u to v.

$d(u, v)$ — is the (weighted) distance between u and v, and defined by $d(u, v) = w(P(u, v))$.

$N_\rho(v)$ — set of all *neighbors* u such that $w(u, v) \le \rho$, i.e., $N_\rho(v) = \{u \mid (u, v) \in E \wedge w(u, v) \le \rho\}$.

$W(v)$ — is the weight of the largest edge adjacent to v. $W(v) = \max_{(v,x)\in E} w(v, x)$

$l(u, v)$ — is the minimum length (number of the edges) shortest path from u to v. Note that there may be more than one shortest path from u to v. Thus $l(u, v)$ is the number of edges of the shortest path having the least number of edges, i.e, $l(u, v) = \min\{|P(u, v)| \mid P(u, v)$ is a shortest path from u to $v\}$.

Definition 1. ***ρ-neighborhood.*** *ρ-neighborhood of a node v, denoted by $\Gamma_\rho(v)$, is the set of the nodes that are within distance ρ from v. $\Gamma_\rho(v) = \{u \mid d(u, v) \le \rho\}$.*

Definition 2. (ρ, λ)***-neighborhood.*** *(ρ, λ)-neighborhood of a node v, denoted by $\Gamma_{\rho,\lambda}(v)$, is the set of all nodes u such that there is a path $Q(v, u)$ such that $w(Q) \le \rho$ and $|Q| \le \lambda$. Clearly, $\Gamma_{\rho,\lambda}(v) \subseteq \Gamma_\rho(v)$.*

Definition 3. ***Shortest Path Diameter (SPD).*** *SPD is denoted by $S(G, w)$ (or S for short) and defined by $S = \max_{u,v \in V} l(u, v)$.*

Definition 4. ***Local Shortest Path Diameter (LSPD).*** *LSPD is denoted by $L(G, w)$ (or L for short) and defined by $L = \max_{v \in V} L(v)$, where $L(v) = \max_{u \in \Gamma_{\ \ (\ )}(v)} l(u, v)$.*

Notice that $L \leq S \leq n$ in any graph. However, there exists graphs, where L is significantly smaller than both S and the (unweighted) diameter of the graph, D. For example, in a chain of n nodes (all edges with weight 1), $S = n$, $D = n$, and $L = 1$.

2.3 Distributed NNT Algorithm

We recall that the basic NNT scheme is as follows. Each node v selects a unique rank $r(v)$. Then each node finds the nearest node of higher rank and connects to it via the shortest path.

Rank selection. The nodes select unique ranks as follows. First a leader is elected by a leader election algorithm. Let s be the leader node. The leader picks a number $p(s)$ from the range $[b - 1, b]$, where b is a number arbitrarily chosen by s, and sends this number $p(s)$ along with its ID (identity number) to its neighbors. A neighbor v of the leader s, after receiving $p(s)$, picks a number $p(v)$ from the open interval $[p(s) - 1, p(s))$, thus $p(v)$ is less than $p(s)$, and then transmits $p(v)$ and $ID(v)$ to all of its neighbors. This process is repeated by every node in the graph. Notice that at some point, every node in the graph will receive a message from at least one of its neighbors since the given graph is connected; some nodes may receive more than one message. As soon as a node u receives the first message from a neighbor v, it picks a number $p(u)$ from $[p(v) - 1, p(v))$, so that it is smaller than $p(v)$, and transmits $p(u)$ and $ID(u)$ to its neighbors. If u receives another message later from another neighbor v', u simply stores $p(v')$ and $ID(v')$, and does nothing else. $p(u)$ and $ID(u)$ constitute u's rank $r(u)$ as follows. For any two nodes u and v, $r(u) < r(v)$ iff i) $p(u) < p(v)$, or ii) $p(u) = p(v)$ and $ID(u) < ID(v)$.

At the end of execution of the above procedure of rank selection, it is easy to make the following observations.

Observation 1

1. *Each node knows the ranks of all of its neighbors.*

2. *The leader s has the highest rank among all nodes in the graph.*

3. *Each node v, except the leader, has one neighbor u, i.e. $(u, v) \in E$, such that $r(u) > r(v)$.*

Connecting to a higher-ranked node. Each node v (except the leader s) executes the following algorithm simultaneously to find the nearest node of higher rank and connect to it. Each node v needs to explore only the nodes in $\Gamma_{W(v)}(v)$ to find a node of higher rank.

Each node v executes the algorithm in *phases*. In the first phase, v sets $\rho = 1$. In the subsequent phases, it doubles the value of ρ; that is, in the ith phase, $\rho = 2^{i-1}$. In a phase of the algorithm, v explores the nodes in $\Gamma_\rho(v)$ to find a node u (if any) such that $r(u) > r(v)$. If such a node with higher rank is not found, v continues to the next phase

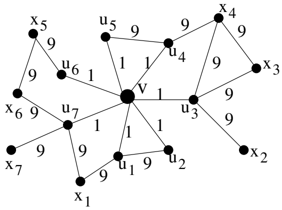


Fig. 1. A network with possible congestion in the edges adjacent to v. Weight of the edges (v, u_i) is 1 for all i, and 9 for the rest of the edges. Assume $r(v) < r(u_i)$ for all i.

with ρ doubled. By Observation 3 of 1, v needs to increase ρ to at most $W(v)$. Each phase of the algorithm consists of one or more *rounds*. In the first round, v sets $\lambda = 1$. In subsequent rounds, values for λ are doubled. In a particular round, v explores all nodes in $\Gamma_{\rho,\lambda}(v)$. At the end of each round, v counts the number of nodes it has explored. If the number of nodes remain the same in two successive rounds of the same phase (that is, v already explored all nodes in $\Gamma_\rho(v)$), v doubles ρ and starts the next phase. If at any point of time v finds a node of higher rank, it then terminates its exploration.

Since all of the nodes explore their neighborhoods simultaneously, many nodes may have overlapping ρ-neighborhoods. This might create congestion of the messages in some edges that may result in increased running time of the algorithm, in some cases by a factor of $\Theta(n)$. Consider the network given in Fig. 1. If $r(v) < r(u_i)$ for all i, when $\rho \geq 2$ and $\lambda \geq 2$, an exploration message sent to v by any u_i will be forwarded to all other u_is. Note that values for ρ and λ for all u_is will not necessarily be the same at a particular time. Thus congestion at any edge (v, u_i) can be as much as the number of such nodes u_i, which can be, in fact, $\Theta(n)$ in some graphs. However, to improve the running time of the algorithm, we keep congestions on all edges bounded by $O(1)$ by sacrificing the quality of the NNT, but only by a constant factor. To do so, v decides that some lower ranked u_is can connect to some higher ranked u_is and informs them instead of forwarding their message to the other nodes (details are given below). Thus v forwards messages from only *one* u_i and this avoids congestion. As a result, a node may not connect to the nearest node of higher rank. However, our algorithm guarantees that distance to the connecting node is not larger than four times the distance to the nearest node of higher rank. The detailed description is given below.

1. Exploration of ρ-neighborhood to find a node of higher rank:

Initiating exploration. Initially, each node v sets radius $\rho \leftarrow 1$ and $\lambda \leftarrow 1$. v explores the nodes in $\Gamma_{\rho,\lambda}(v)$ in a BFS-like manner to find if there is a node $x \in \Gamma_{\rho,\lambda}(v)$ such that $r(v) < r(x)$. v sends *explore* messages $< explore, v, r(v), \rho, \lambda, pd, l >$ to all $u \in N_\rho(v)$. In the message $< explore, v, r(v), \rho, \lambda, pd, l >$, v is the originator of the *explore* message; $r(v)$ is its rank, ρ is its current phase value; λ is its current round number in this phase; pd is the weight of the path traveled by this message so far (from v to the current node), and l is the number of links that the message can travel further. Before v sends the message to its neighbor u, v sets $pd \leftarrow w(v, u)$ and $l \leftarrow \lambda - 1$.

Forwarding *explore* messages. Any node y may receive more than one *explore* message from the same originator v via different paths for the same round. Any subsequent message is forwarded only if the later message arrived through a shorter path than the previous one. Any node y, after receiving the message $< explore, v, r(v), \rho, \lambda, pd, l >$ from one of its neighbors, say z, checks if it previously received another message $< explore, v, r(v), \rho, \lambda, pd', l' >$ from z' with the same originator v such that $pd' \leq pd$. If so, y sends back a *count* message to z' with count = 0. The purpose of the *count* messages is to determine the number of nodes explored by v in this round. Otherwise, if $r(v) < r(y)$, y sends back a *found* message to v containing y's rank. Otherwise, If $N_{\rho-pd}(y) - \{z\} = \phi$ or $l = 0$, y sends back a *count* message with count = 1 and sets a marker $counted(v, \rho, \lambda) \leftarrow TRUE$. The purpose of the marker $counted(v, \rho, \lambda)$ is to make sure that y is counted only once for the same source v and in the same phase and round of the algorithm. If $r(v) > r(y)$, $l > 0$, and $N_{\rho-pd}(y) - \{z\} \neq \phi$, y forwards the message to all of its neighbors $u \in N_{\rho-pd}(y) - \{z\}$ after setting $pd \leftarrow pd + w(y, u)$ and $l \leftarrow l - 1$.

Controlling Congestion. If at any time step, a node v receives more than one, say $k > 1$, *explore* messages from different originators u_i, $1 \leq i \leq k$, it takes the following actions. Let $< explore, u_i, r(u_i), \rho_i, \lambda_i, pd_i, l_i >$ be the *explore* message from originator u_i. If there is a u_j such that $r(u_i) < r(u_j)$ and $pd_j \leq \rho_i$, v sends back a *found* message to u_i telling that u_i can connect to u_j where weight of the connecting path $w(Q(v, v')) = pd_i + pd_j \leq 2\rho_i$. In this way, some of the u_is will be replied back a *found* message and their *explore* messages will not be forwarded by v.

Let u_s be the node with lowest rank among the rest of u_is (i.e., those u_is which were not sent a *found* message by v), and u_t be an arbitrary node among the rest of u_is and let $t \neq s$. Now it must be the case that ρ_s is strictly smaller than ρ_t , i.e., u_s is in an earlier phase than u_t. This can happen if, in some previous phase, u_t exhausted its ρ-value with smaller λ-value leading to a smaller number of rounds in that phase and quick transition to the next phase. In such a case, we keep u_t waiting for at least one round without affecting the overall running time of the algorithm. To do this, v forwards *explore* message of u_s only and sends back *wait* messages to all u_t.

Each *explore* message triggers exactly one reply (either *found*, *wait*, or *count* message). These reply-back messages move in similar fashion as of *explore* messages but in reverse direction and they are aggregated (convergecast) on the way back as described next. Thus those reply messages also do not create any congestion in any edge.

Convergecast of the Replies of the *explore* Messages. If any node y forwards the *explore* message $< explore, v, r(v), \rho, \lambda, pd, l >$ received from z for the originator v to its neighbors in $N_{\rho-pd}(y) - \{z\}$, eventually, at some point later, y will receive replies to these *explore* messages, which y forwarded to $N_{\rho-pd}(y) - \{z\}$. Each of these replies is either a *count* message, *wait* message, or a *found* message. Once y receives replies from all nodes in $N_{\rho-pd}(y) - \{z\}$, it takes the following actions. If at least one of the replies is a *found* message or a *wait* message, y ignores all of the *count* messages and sends the *found* message or the *wait* message to z towards the destination v. If all of the replies are *count* messages, y adds the count values of these messages and sends a single *count* message to v with the aggregated count. Also, y adds itself to the count if the marker $counted(v, \rho, \lambda) = FALSE$ and sets $counted(v, \rho, \lambda) \leftarrow TRUE$. At the very beginning, y initializes $counted(v, \rho, \lambda) \leftarrow FALSE$. The *count* messages (also the *wait* and *found* messages) travel in the opposite direction of the *explore* messages using the same paths toward v. Thus the *count* messages form a convergecast as opposed to the (controlled) broadcast of the *explore* messages.

Actions of the Originator after Receiving the Replies of the *explore* messages. At some time step, v receives replies of the *explore* messages originated by itself from all nodes in $N_\rho(v)$. Each of these replies is either a *count* message, *wait* message, or a *found* message. If at least one of the replies is a *found* message, v is done with exploration and makes the connection as described in Item 2 below. Otherwise, if there is a *wait* message, v again initiates exploration with same ρ and λ. If all of them are *count* messages: (a) if $\lambda = 1$, v initiates exploration with $\lambda \leftarrow 2$ and the same ρ; (b) if $\lambda > 1$ and count-value for this round is larger than that of the previous round, v initiates exploration with $\lambda \leftarrow 2\lambda$ and the same ρ; (c) otherwise v initiates exploration with $\lambda \leftarrow 2\lambda$ and $\rho \leftarrow 2\rho$.

2. Making Connection:

Let u be a node with higher rank that v found by exploration. If v finds more than one node with rank higher than itself, it selects the nearest one among them. Let $Q(v, u)$ be the path from v to u. The path $Q(v, u)$ is discovered when u is found in the exploration process initiated by v. The edges in $Q(v, u)$ are added in the resulting spanning tree as follows. To add the edges in $Q(v, u)$, v sends a *connect* message to u along this path. During the exploration process, the intermediate nodes in the path simply keeps tracks of the predecessor and successor nodes for this originator v. Let $Q(v, u) =< v, \ldots, x, y, \ldots, u >$. By our choice of u, note that all the intermediate nodes will have rank lower than $r(v)$. When the connect message passes through the edge (x, y), node x uses (x, y) as its connecting edge regardless of x's rank. If x did not find its connecting node yet, x stops searching for such nodes as the edge (x, y) serves as x's connection. If x is already connected using a path, say $< x, x_1, x_2, \ldots, x_k >$, the edge (x, x_1) is removed from the tree, but the rest of the edges in this path still remains in the tree. All nodes in path $Q(v, u)$ including v upgrade their ranks to $r(u)$; i.e., they assumes a new rank which is equal to u's rank. It might happen that in between exploration and connection, some node x in path $Q(v, u)$ changed its rank due to a connection by some origin other than v. In such a case, when the connect message travels through x, if x's current rank is larger than $r(v)$, x accepts the connections as the last node in the path and returns a *rank-update* message toward v instead of forwarding the connect message to the next node (i.e., y) toward u. This is necessary to avoid cycle creation.

2.4 Analysis of Algorithm

In this section, we analyze the correctness and performance of the distributed NNT algorithm. The following lemmas and theorems show our results.

Lemma 1. *Let, during exploration, v found a higher ranked node u and the path $Q(v, u)$. If v's nearest node of higher rank is u', then $w(Q) \leq 4d(v, u')$.*

Proof. Assume that u is found when v explored a (ρ, λ)-neighborhood for some ρ and λ. Then $d(v, u') > \rho/2$, otherwise, v would find u' as a node of higher rank in the previous phase and would not explore the ρ-neighborhood. Now, u could be found by v in two ways. i) The *explore* message originated by v reached u and u sent back a *found* message. In this case, $w(Q) \leq \rho$. ii) Some node y received two *explore* messages originated by v and u via the paths $R(v, y)$ and $S(u, y)$ respectively, where $r(v) < r(u)$ and $w(S) \leq \rho$; and y sent a *found* message to v (see "Controlling Congestion" in Item 1). In this case, $w(Q) = w(R) + w(S) \leq 2\rho$, since $w(R) \leq \rho$. Thus for both cases, we have $w(Q) \leq 4d(v, u')$.

Lemma 2. *The algorithm adds exactly* $n - 1$ *edges to the NNT.*

Proof. Let a node v connect to another node u using the path $Q(v, u) = < v, \ldots, x, y, z, \ldots, u >$. When a connect message goes through an edge, say (x, y) (from x to y), in this path, the edge (x, y) is added to the tree. We say the edge (x, y) is associated to node x (not to y) based on the direction of the flow of the connect message. If, previously, x was associated to some other edge, say (x, y'), the edge (x, y') was removed from the tree. Thus each node is associated to at most one edge.

Except the leader s, each node x must make a connection and thus at least one connect message must go through or from x. Then, each node, except s, is associated to some edge in the tree.

Thus each node, except s, is associated to exactly one edge in NNT; and s cannot be associated to any node since a connect message cannot be originated by or go through s.

Now, to complete the proof, we need to show that no two nodes are associated to the same edge. Let x be associated to edge (x, y). When the connect message went through (x, y) from x to y, $r(x)$ and $r(y)$ became equal. Later if another connect message increased $r(x)$, then either $r(y)$ also increased to the same value or x became associated to some edge other than (x, y). Thus, while keeping (x, y) associated to x, it must be true that $r(x) \leq r(y)$. Then any new connect message that might make (x, y) associated to y by passing the connect message from y to x, must pass through x toward some node with rank higher than $r(y)$ (i.e., the connect message cannot terminate at x). This will make x associated to some other edge than (x, y). Therefore, no two nodes are associated to the same edge.

Lemma 3. *The edges in the NNT added by the given distributed algorithm does not create any cycle.*

Proof. Each node has a unique rank and it can connect only to a node with higher rank. Thus if each node can connect to a node of higher rank using a direct edge (as in a complete graph), it is easy to see that there cannot be any cycle. However, in the above algorithm, a node u connects to a node of higher rank, v, $r(u) < r(v)$, using shortest path $P(u, v)$, which may contain more than one edge and in such a path, ranks of the intermediate nodes are smaller than $r(u)$. Thus the only possibility of creating a cycle is when some other connecting shortest path goes though these intermediate nodes. For example, in Fig. 2, the paths $P(u, v)$ and $P(p, q)$ both go through a lower ranked node x.

In Fig. 2, if p connects to q using path $< p, x, q >$ before u makes its connection, x gets a new rank which is equal to $r(q)$. Thus u finds a higher ranked node, x, at a closer distance than v and connects to x instead of v. Note that if x is already connected to some node, it releases such connection and takes $< x, q >$ as its new connection, i.e., q is x's new parent. Now y_2 uses either (y_2, x) or (y_2, v), but not both, for its connection. Thus there is no cycle in the resulting graph.

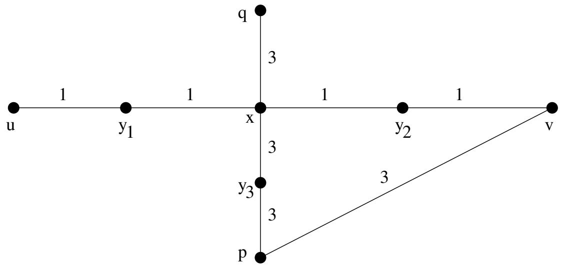


Fig. 2. A possible scenario of creating cycle and avoiding it. Nodes are marked with letters. Edge weights are given in the figure. Let $r(u) = 11, r(v) = 12, r(p) = 13, r(q) = 14$, and ranks of the rest of the nodes are smaller than 11. u connects to v, v connects to p, and p connects to q.

Now, assume that u already made its connection to v, but p is not connected yet. At this moment, x's rank is upgraded to $r(v)$ which is still smaller than $r(p)$. Thus p finds q as its nearest node of higher rank and connects using path $< p, x, q >$. In this connection process, x removes its old connecting edge (x, y_2) and gets (x, q) as its new connecting edge. Again, there cannot be any cycle in the resulting graph.

If x receives the connection request messages from both u (toward v) and p (toward q) at the same time, x only forwards the message for the destination with highest rank; here it is q. u's connection only goes up to x. Note that x already knows the ranks of both q and v from previous exploration steps.

From Lemmas 2 and 3 we have the following theorem.

Theorem 1. *The above algorithm produces a tree spanning all nodes in the graph.*

We next show that the spanning tree found is an $O(\log n)$-approximation to the MST (Theorem 2).

Theorem 2. *Let the NNT be the spanning tree produced by the above algorithm. Then the cost of the tree $c(NNT) \leq 4\lceil \log n \rceil c(MST)$.*

Proof. Let $H = (V_H, E_H)$ be a *complete* graph constructed from $G = (V, E)$ as follows. $V_H = V$ and weight of the edge $(u, v) \in E_H$ is the weight of the shortest path $P(u, v)$ in G. Now, the weights of the edges in H satisfy triangle inequality. Let NNT_H be a nearest neighbor tree and MST_H be a minimum spanning tree on H. We can show that $c(NNT_H) \leq \lceil \log n \rceil c(MST_H)$ [14].

Let NNT' be a spanning tree on G, where each node connects to the nearest node of higher rank. Then it is easy to show that $c(NNT') \leq c(NNT_H)$ and $c(MST_H) \leq c(MST)$.

By Lemma 1, we have $c(NNT) \leq 4c(NNT')$. Thus we get,

$$c(NNT) \leq 4c(NNT_H) \leq 4\lceil \log n \rceil c(MST_H) \leq 4\lceil \log n \rceil c(MST).$$

Theorem 3. *The running time of the above algorithm is $O(D + L \log n)$.*

Proof. Time to elect leader is $O(D)$. The rank choosing scheme takes also $O(D)$ time.

In the exploration process, ρ can increase to at most $2W$; because, within distance W, it is guaranteed that there is a node of higher rank (Observation 3 of 1). Thus, the number of phases in the algorithm is at most $O(\log W) = O(\log n)$.

In each phase, λ can grow to at most $4 * L$. When $L \leq \lambda < 2L$ and $2L \leq \lambda < 4L$, in both rounds, the count of the number of nodes explored will be the same. As a result, the node will move to the next phase.

Now, in each round, a node takes at most $O(\lambda)$ time; because the messages travel at most λ edges back and forth and at any time the congestion in any edge is $O(1)$. Thus any round takes time at most

$$\sum_{\lambda=1}^{\log(4L)} O(\lambda) = O(L).$$

Thus time for the exploration process is $O(L \log W)$. Total time of the algorithm for leader election, rank selection, and exploration is $O(D + D + L \log n) = O(D + L \log n)$.

Theorem 4. *The message complexity of the algorithm is* $O(|E| \log L \log n) = O(|E| \log^2 n)$.

Proof. The number of phases in the algorithm is at most $O(\log L)$. In each phase, each node executes at most $O(\log W) = O(\log n)$ rounds. In each round, each edge carries $O(1)$ messages. That is, number of messages in each round is $O(|E|)$. Thus total messages is $O(|E| \log L \log n)$.

3 Exact vs. Approximate MST and Near-Optimality of NNT Algorithm

Comparison with Distributed Algorithms for (Exact) MST. There can be a large gap between the local shortest path diameter L and $\tilde{\Omega}(\sqrt{n})$, which is the lower bound for exact MST computation. In particular, we can show that there exists a family of graphs where NNT algorithm takes $\tilde{O}(1)$ time, but *any* distributed algorithm for computing (exact) MST will take $\tilde{\Omega}(\sqrt{n})$ time. To show this we consider the parameterized (weighted) family of graphs called $\mathcal{J}_m^K$ defined in Peleg and Rabinovich [8]. (One can also show a similar result using the family of graphs defined by Elkin [9].) The size of $\mathcal{J}_m^K$ is $n = \Theta(m^{2K})$ and its diameter $\Theta(Km) = \Theta(Kn^{1/(2K)})$. For every $K \geq 2$, Peleg and Rabinovich show that any distributed algorithm for the MST problem will take $\Omega(\sqrt{n}/BK)$ time on some graphs belonging to the family. The graphs of this family have $L = \Theta(m^K) = \sqrt{n}$. We modify this construction as follows: the weights on all the highway edges except the first highway (H^1) is changed to 0.5 (originally they were all zero); all other weights remain the same. This makes $L = \Theta(Km)$, i.e., same order as the diameter. One can check that the proof of Peleg and Rabinovich is still valid, i.e., the lower bound for MST will take $\Omega(\sqrt{n}/BK)$ time on some graphs of this family, but NNT algorithm will take only $\tilde{\Omega}(L)$ time. Thus we can state:

Theorem 5. *For every $K \geq 2$, there exists a family of $n-$vertex graphs in which NNT algorithm takes $O(Kn^{1/(2K)})$ time while any distributed algorithm for computing the exact MST requires $\tilde{\Omega}(\sqrt{n})$ time. In particular, for every $n \geq 2$, there exists a family of graphs in which NNT algorithm takes $\tilde{O}(1)$ time whereas* any *distributed MST algorithm will take $\tilde{\Omega}(\sqrt{n})$ time.*

Such a large gap between NNT and any distributed MST algorithm can be also shown for constant diameter graphs, using a similar modification of a lower bound construction given in Elkin [9] (which generalizes and improves the results of Lotker et al [19]).

Near (existential) optimality of NNT algorithm. We show that there exists a family of graphs such that any distributed algorithm to find a $H(\leq \log n)$-approximate MST takes $\Omega(L)$ time (where L is the local shortest path diameter) on some of these graphs. Since NNT algorithm takes $\tilde{O}(D+L)$, this shows the near-tight optimality of NNT (i.e., tight up to a $polylog(n)$ factor). This type of optimality is called *existential optimality* which shows that our algorithm cannot be improved in general.

To show our lower bound we look closely at the hardness of distributed approximation of MST shown by Elkin [9]. Elkin constructed a family of weighted graphs $\mathcal{G}^{\omega}$ to show a lower bound on the time complexity of any $H-$approximation distributed MST algorithm (whether deterministic or randomized). We briefly describe this result and show that this lower bound is *precisely the local shortest path diameter L of the graph*. The graph family $\mathcal{G}^{\omega}(\tau, m, p)$ is parameterized by 3 integers τ, m, and p, where $p \leq \log n$. The size of the graph $n = \Theta(\tau m)$, the diameter is $D = \Theta(p)$ and the local shortest path diameter can be easily checked to be $L = \Theta(m)$. Note that graphs of different size, diameter, and $LSPD$ can be obtained by varying the parameters τ, m, and p. (We refer to [9] for the detailed description of the graph family and the assignment of weights.) We now slightly restate the results of [9] (assuming the $\mathcal{CONGEST}(\mathcal{B})$ model):

Theorem 6 ([9]). *1. There exists graphs belonging to the family $\mathcal{G}^{\omega}(\tau, m, p)$ having diameter at most D for $D \in 4, 6, 8, \ldots$ and LPSD $L = \Theta(m)$ such that any randomized H-approximation algorithm for the MST problem on these graphs takes $T = \Theta(L) = \Omega((\frac{n}{H \cdot D \cdot B})^{1/2-1/(2(D-1))}$ distributed time.*
2. If $D = O(\log n)$ then the lower bound can be strengthened to $\Theta(L) = \Omega(\sqrt{\frac{n}{H \cdot B \cdot \log n}})$.

Using a slightly different weighted family $\tilde{\mathcal{G}^{\omega}}(\tau, m)$ parameterized by two parameters τ and m, where size $n = \tau m^2$, diameter $D = \Omega(m)$ and LSPD $L = \Theta(m^2)$, one can strengthen the lower bound of the above theorem by a factor of $\sqrt{\log n}$ for graphs of diameter $\Omega(n^{\delta})$.

The above results show the following two important facts:

1. There are graphs having diameter $D << L$ where any H-approximation algorithm requires $\Omega(L)$ time.

2. More importantly, for graphs with very different diameters — varying from a constant (including 1, i.e., exact MST) to logarithmic to polynomial in the size of n — the lower bound of distributed approximate-MST is captured by the local shortest path parameter. In conjunction with our upper bound given by the NNT algorithm which takes $\tilde{O}(D + L)$ time, this implies that the LPSD L captures in a better fashion the complexity of distributed $O(\log n)$-approximate-MST computation.

4 Special Classes of Graphs

We show that in unit disk graphs (a commonly used model for wireless networks) $L = 1$, and in random weighted graphs, $L = O((\log n))$ with high probability. Thus our algorithm will run in near-optimal time of $\tilde{O}(D(G))$ on these graphs.

Unit Disk Graph (UDG). Unit disk graph is an euclidian graph where there is an edge between two nodes u and v if and only if general $dist(u, v) \leq R$ for some R (R is typically taken to be 1). Here $dist(u, v)$ is the euclidian distance between u and v which is the weight of the edge (u, v). For any node v, $W(v) \leq R$. Now if there is node u such that $d(u, v) \leq R$, then $dist(u, v) \leq R$ by triangle inequality. Thus $(u, v) \in E$ and the edge (u, v) is the shortest path from u to v. As a result, for any UDG, $L = 1$. For a 2-dimensional UDG, diameter can be as large as $\Theta(\sqrt{(}n))$.

Graph with Random Edge Weights. Consider any graph G (topology can be arbitrary) with edge weights chosen randomly from an arbitrary distribution (i.e., each edge weight is chosen i.i.d from the distribution). The following theorem shows that L and S is small compared to the diameter for such a graph.

Theorem 7. *Consider a graph G where the edge weights are chosen randomly from a (arbitrary) distribution with a constant (independent of n) mean. Then: (1) $L = O(\log n)$ with high probability (whp), i.e., probability at least $1 - 1/n^{\Omega(1)}$; and (2) the shortest path diameter $S = O(\log n)$ if $D < \log n$ and $S = O(D)$ if $D \geq \log n$ whp.*

Proof. Without loss of generality, we can assume that edge weights are randomly drawn from $[0, 1]$ with mean μ. Otherwise the edge weights can be normalized to this range without affecting the desired result. For any node v, $W(v) \leq 1$. Consider any path with $m = k \log n$ edges, for some constant k. Let the weights of the edges in this path be $w_1, w_2, \cdots, w_m$. For any i, $E[w_i] = \mu$. Since $\frac{1}{2}\mu k \log n \geq 1$ for sufficiently large k, we have

$$\Pr\{\sum_{i=1}^{m} w_i \leq 1\} \leq \Pr\{\sum_{i=1}^{m} w_i \leq \frac{1}{2}\mu k \log n\} = \Pr\{\mu - \frac{1}{m}\sum_{i=1}^{m} w_i \geq \frac{1}{2}\mu\}.$$

Using Hoeffding bound [20] and putting $k = \frac{6}{\mu^2}$,

$$\Pr\{\mu - \frac{1}{m}\sum_{i=1}^{m} w_i \geq \frac{1}{2}\mu\} \leq e^{-m\mu^2/2} = \frac{1}{n^3}.$$

Thus if it is given that the weight of a path is at most 1, then the probability that the number of edges $\leq \frac{6}{\mu^2} \log n$ is at most $\frac{1}{n^3}$. Now consider all nodes u such that $d(v, u) \leq W(v)$. There are at most $n - 1$ such nodes and thus there are at most $n - 1$ shortest paths leading to those nodes from v.

Thus using union bound, $\Pr\{L(v) \geq \frac{6}{\mu^2} \log n\} \leq n \times \frac{1}{n^3} = \frac{1}{n^2}$.

Using $L = \max\{L(v)\}$ and union bound, $\Pr\{L \geq \frac{6}{\mu^2} \log n\} \leq n \times \frac{1}{n^2} = \frac{1}{n}$.

Therefore, with probability at least $1 - \frac{1}{n}$, L is smaller than or equal to $\frac{6}{\mu^2} \log n$.

Proof of part 2 is similar.

References

1. Gallager, R., Humblet, P., Spira, P.: A distributed algorithm for minimum-weight spanning trees. ACM Transactions on Programming Languages and Systems **5**(1) (1983) 66–77
2. Chin, F., Ting, H.: An almost linear time and $O(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees. In: Proc. 26th IEEE Symp. Foundations of Computer Science. (1985) 257–266
3. Gafni, E.: Improvements in the time complexity of two message-optimal election algorithms. In: Proc. of the 4th Symp. on Principles of Distributed Computing. (1985) 175–185
4. Awerbuch, B.: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In: Proc. 19th ACM Symp. on Theory of Computing. (1987) 230–240
5. Garay, J., Kutten, S., Peleg, D.: A sublinear time distributed algorithm for minimum-weight spanning trees. SIAM J. Comput. **27** (1998) 302–316
6. Kutten, S., Peleg, D.: Fast distributed construction of k-dominating sets and applications. J. Algorithms **28** (1998) 40–66
7. Elkin, M.: A faster distributed protocol for constructing minimum spanning tree. In: Proc. of the ACM-SIAM Symp. on Discrete Algorithms. (2004) 352–361
8. Peleg, D., Rabinovich, V.: A near-tight lower bound on the time complexity of distributed mst construction. In: Proc. of the 40th IEEE Symp. on Foundations of Computer Science. (1999) 253–261
9. Elkin, M.: Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In: Proc. of the ACM Symposium on Theory of Computing. (2004) 331 – 340
10. Elkin, M.: An overview of distributed approximation. ACM SIGACT News Distributed Computing Column **35**(4) (2004) 40–57
11. Peleg, D.: Distributed Computing: A Locality Sensitive Approach. SIAM (2000)
12. Korach, E., Moran, S., Zaks, S.: Optimal lower bounds for some distributed algorithms for a complete network of processors. Theoretical Computer Science **64** (1989) 125–132
13. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (1994)
14. Khan, M., Kumar, V.A., Pandurangan, G.: A simple randomized scheme for constructing low-cost spanning subgraphs with applications to distributed algorithms. In: Technical Report, Dept. of Computer Science, Purdue University. (2005. http://www.cs.purdue.edu/homes/gopal/localmst.pdf)
15. Rosenkrantz, D., Stearns, R., Lewis, P.: An analysis of several heuristics for the traveling salesman problem. SIAM J. Comput. **6(3)** (1977) 563–581
16. Imase, M., Waxman, B.: Dynamic steiner tree problem. Siam J. Discrete Math **4(3)** (1991) 369–384
17. Korach, E., Moran, S., Zaks, S.: The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processors. SIAM Journal of Computing **16(2)** (1987) 231–236
18. Lotker, Z., Patt-Shamir, B., Pavlov, E., Peleg, D.: Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. SIAM J. Comput. **35**(1) (2005) 120–131
19. Lotker, Z., Patt-Shamir, B., Peleg, D.: Distributed mst for constant diameter graphs. In: Proc. of the 20th ACM Symp. on Principles of Distributed Computing. (2001) 63–72
20. Hoeffding, W.: Probability for sums of bounded random variables. J. of the American Statistical Association **58** (1963) 13–30

On Randomized Broadcasting in Power Law Networks

Robert Elsässer[*]

University of Paderborn
Institute for Computer Science
33102 Paderborn, Germany
elsa@upb.de

Abstract. Broadcasting algorithms have various range of applications in different fields of computer science. In this paper we consider randomized broadcasting algorithms in power law graphs which are often used to model large scale real world networks such as the Internet. We prove that for certain (truncated) power law networks there exists a time efficient randomized broadcasting algorithm whose communication complexity is bounded by an asymptotically optimal value.

In order to describe these power law graphs, we first consider the generalized random graph model $G(\mathbf{d}) = (V, E)$, where $\mathbf{d} = (d_1, \ldots, d_n)$ is a given sequence of expected degrees, and two nodes $v_i, v_j \in V$ share an edge in $G(\mathbf{d})$ with probability $p_{i,j} = d_i d_j / \sum_{k=1}^{n} d_k$, independently [7]. We show for these graphs that if the expected minimal degree $d_{\min}$ is larger than $\log^{\delta} n$, $\delta > 2$, and the number of nodes with expected degree d_i is proportional to $(d_i - d_{\min} + 1)^{-\beta}$, where $\beta > 2$ is a constant, then a simple randomized broadcasting protocol exists, which spreads any information r to all nodes of a graph $G(\mathbf{d})$ within $O(\log n)$ steps by using at most $O(n \max\{\log \log n, \log n / \log d_{\min}\})$ transmissions. Furthermore, we discuss the applicability of our methods in more general power law graph models. Please note that our results hold with probability $1 - 1/n^{\Omega(1)}$, even if n and $\mathbf{d}$ are completely unknown to the nodes of the graph.

The algorithm we present in this paper uses a very simple communication rule, and can efficiently handle restricted node failures or dynamical changes in the size of the network. In addition, our methods might be useful for further research in this field.

1 Introduction

Randomized broadcasting algorithms have extensively been studied in various network topologies. Such algorithms naturally provide robustness, simplicity and scalability. As an example, consider the so-called *push model* [11]: In a graph $G = (V, E)$ we place at some time t an information r on one of the nodes. Then, in each succeeding round, any *informed* vertex forwards the information

[*] The research was performed while the author visited the Department of Mathematics at the University of California, San Diego. Partly supported by the German Research Foundation under contract EL-399/1-1.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 370–384, 2006.

to a communication partner over an incident edge selected independently and uniformly at random. It is known that the push algorithm spreads an information within $O(\log n)$ rounds to all nodes of a random graph $G(\mathbf{d})$, with probability $1 - o(1/n)$, whenever $d_1 = \cdots = d_n > (2 + \Omega(1)) \log n$ [16], and this result can easily be generalized to any random graph $G(\mathbf{d})$ with $d_{\min} > (2 + \Omega(1)) \log n$. However, this algorithm generates $\Omega(n \log n)$ transmissions of r. Therefore, some modification of this scheme is needed in order to improve its efficiency on the random graphs mentioned before.

1.1 Models and Motivation

The study of information spreading in large networks has various fields of application in distributed computing. Consider for example the maintenance of replicated databases on name servers in a large network [11]. There are updates injected at various nodes, and these updates must be propagated to all the nodes in the network. In each step, a processor and its neighbor check whether their copies of the database agree, and if not, they perform the necessary updates. In order to be able to let all copies of the database converge to the same content, efficient broadcasting algorithms have to be developed.

Another well known example occurs in the analysis of epidemic disease. Often, mathematical studies about infection propagation make the assumption that an infected person spreads the infection equally likely to any member of the population [25] which leads to a complete graph for the underlying network. Whenever the question is, how fast the disease infects the whole community, the problem reduces to the broadcasting problem in the push model. However, in most of these papers, spreaders are only active in a certain time window, and the question of interest is, whether on certain networks modelling personal contacts an epidemic outbreak occurs. Several threshold theorems involving the basic reproduction number, contact number, and the replacement number have been stated. See e.g. [21] for a collection of results concerning the mathematics of infectious diseases. Notably, the analysis of epidemic diseases has recently been extended in [30] to generalized random graphs with arbitrary degree distributions.

There is an enormous amount of experimental and theoretical study of broadcasting algorithms in various models and on different network topologies. Several (deterministic and randomized) algorithms have been developed and analyzed. In this paper we only concentrate on the efficiency of randomized algorithms, and study their time and communication complexity using a simple communication model. The advantage of randomized broadcasting is its inherent robustness against several kinds of failures and dynamical changes compared to deterministic schemes that either need substantially more time [17] or can tolerate only a relatively small number of faults [27]. Our intention is to develop randomized broadcasting algorithms with the following properties:

- They can successfully handle restricted communication failures.
- They are fully adaptive and work correctly if the size or the topology of the network changes slightly during the execution of the algorithm.
- Their runtime and communication complexity is asymptotically minimal.

When using the push algorithm, the effects of node failures are very limited and dynamical changes in the size of the network do not really affect its efficiency. However, as described above, the push algorithm produces a large amount of transmissions.

Several termination mechanisms noticing when a specific information becomes available to all nodes so that its transmission can be stopped were investigated (e.g. [11,23]). Using simple mechanisms for the push model, it is possible to bound the number of transmissions in a random graph $G(\mathbf{d})$ with $d_{\min} = (2 + \Omega(1)) \log n$ to $O(n \log n)$.

An idea introduced in [11] consists of so called *pull transmissions*, i.e., any (informed or uninformed) node is allowed to call a randomly chosen neighbor, and whenever this neighbor is informed, the information is sent from the called to the calling node. However, these kind of transmission makes only sense if new or updated informations occur frequently in the network. It was observed in complete graphs that after a constant fraction of the nodes has been informed, then within $O(\log \log n)$ additional steps every node of the graph becomes informed as well [11,23]. This implies that in such graphs at most $O(n \log \log n)$ transmissions are needed if the distribution of the information is stopped at the right time.

Of particular interest is the study of the behavior of randomized broadcasting in large scale real world networks. In [1,2,15,22] it has been observed that in many real-world networks, including the Web, the Internet, telephone call graphs, and various social and biological networks, the degrees of the nodes have a so called power law distribution, i.e., the fraction of vertices with degree d is proportional to $d^{-\beta}$, where $\beta > 2$ is a fixed constant (for most of the previously mentioned networks it holds that $\beta \in (2, 3)$). In [2], Barabási and Albert suggested to model complex real world networks by the following dynamic random graph process: We start with a complete graph consisting of n_0 vertices, and then new nodes are added to the graph one at a time and joined to a fixed number $m < n_0$ of earlier vertices, selected with probabilities proportional to their degrees. In [4] it has been shown that the graphs constructed by a similar method is close to a power law with $\beta = 3$, i.e., the fraction of vertices with degree d is proportional to d^{-3} for any $d \in \{m, \ldots, n^{1/15}\}$.

Several other dynamic random graph models have been proposed and analyzed recently (e.g. [1,5,26]) Another approach is to model power law networks by a static random graph process. Clearly, the classical random graph model of [14,18] is ill suited to model such networks. Therefore, Chung and Lu proposed a more general random graph model with arbitrary degree distributions: For a sequence $\mathbf{d} = (d_1, \ldots, d_n)$ let $G(\mathbf{d})$ be the graph in which edges are independently assigned to each pair of vertices (v_i, v_j) with probability $d_i d_j / \sum_{k=1}^{n} d_k$ [7]. If now the degree distribution $\mathbf{d}$ obeys a power law, then the resulting graph is well suited for modelling power law graphs.

We should mention here that the real world networks described before also possess other properties (e.g. exhibit high levels of clustering, cf. [29,20]) which are unspecific for the random graphs considered above. Although we do not deal with graphs which have the other properties, we hope the techniques and results stated in this paper might provide insights at a more general level, too.

1.2 Related Work

Most of papers dealing with randomized broadcasting analyze the runtime of the push algorithm in different graph classes. Pittel [31] proved that it is possible to broadcast an information within $\log_2(n)+\ln(n)+O(1)$ steps in a complete graph, by using the push algorithm. Feige et al. determined asymptotically optimal upper bounds for the runtime of this algorithm in the classical random graph, bounded degree graphs and the hypercube [16]. Kempe et al. showed that in certain geometric networks any information is spread to nodes at distance t in $O(\ln^{1+\epsilon} t)$ steps [24].

In [23], Karp et al. combined the push and pull models, and presented a termination mechanism in order to bound the number of total transmissions by $O(n \log\log n)$ in complete graphs. It has also been shown that this result is asymptotically optimal among these kind of algorithms. They also considered communication failures and analyzed the performance of the algorithm in the case when the random connections established in each round follow an arbitrary probability distribution. The algorithm works fully distributed, whereby the nodes are supposed to have an estimation on the size of the network. We should note that we could not use the mechanisms of [23] for the graph $G(\mathbf{d})$, especially not in the case when $d_{\min}$ is below some threshold and whenever it is assumed that the nodes do not have any global information about the the graph. Therefore, we developed a new termination mechanism for the graphs we consider in this paper.

It is worth mentioning that the termination mechanism we present in Sections 2 and 3 can also be used for efficient broadcasting in the classical random graph model. We show in [12] that, with probability $1 - 1/n^{\Omega(1)}$, we can inform each node of a graph $G(\mathbf{d})$, in which it holds that $d_1 = \cdots = d_n > \log^2 n$, within $O(\log n)$ steps and by using at most $O(\max\{n \log\log n, n \log n / \log d_i\})$ transmissions.

As mentioned before, we mainly consider the general random graph model of [7] in this paper. Recently, several properties of these graphs have been analyzed. In [7], Chung and Lu determined the size and the volume of the connected components. In [8] they analyzed the distances between two nodes in such graphs. The largest eigenvalues of the adjacency matrix have been studied in [9], and the second smallest eigenvalue of the normalized Laplacian is approximated in [10], whenever $d_{\min} \gg \sqrt{\sum_{k=1}^{n} d_k/n} \cdot \log^3 n$. In the same paper, Chung et al. proved a very nice result concerning the distribution of the eigenvalues of the normalized Laplacian, if $d_{\min}$ is large enough.

1.3 Our Results

We present an adaptive randomized broadcasting algorithm which is able to distribute an information r, placed initially on a node of a random graph $G(\mathbf{d})$

of size n, to all nodes in the network within $O(\log n)$ steps by producing at most $O(n \max\{\log\log n, \frac{\log n}{\log d_{\min}}\})$ transmissions of r. Here $d_{\min}$ is assumed to be larger than $\log^{\delta} n$, where $\delta > 2$ is a constant, and the number of vertices with expected degree d_i is proportional to $(d_i - d_{\min} + 1)^{-\beta}$, $\beta > 2$. Then, we discuss the applicability of our methods in some variants of the dynamical random power law graph models of [2] and [1].

Although the description of the algorithm is not simple, it uses a very simple communication structure, and is robust against limited communication failures or restricted short time changes in the size of the network. Moreover, since we do not require any previous knowledge about the size of the network or about the degrees of the nodes, our algorithm is robust against any kind of long time changes in the size of the network. In our proofs, we assume a fully synchronized scenario, however, the algorithm supports restricted asynchronicity as well. Please note that the results can also be shown for $d_{\min} \geq \delta' \log n/n$, where δ' is a large constant, however, the proofs would be much more complicated, and we concentrate therefore on the weaker case in this paper.

The rest of the paper is organized in three sections. In Section 2 we present the algorithm with the properties mentioned above, whereby it is assumed that the nodes have an estimation of $\log n/\log d_{\min}$. In Section 3 we improve our algorithm by using an additional trick that enables us to solve the broadcasting problem efficiently even if nothing is known to the nodes about the graph. Finally, the last section contains our conclusions and points to some open problems.

2 Broadcasting Algorithm with Partial Knowledge

In this section, we analyze the behavior of a modified push & pull algorithm on certain (truncated) random power law graphs. As described in the introduction, the random graph $G(\mathbf{d}) = (V, E)$ is defined in the following way: Given n and the sequence $\mathbf{d} = (d_1, \ldots, d_n)$, generate graph $G(\mathbf{d})$ with n vertices by letting each pair $(v_i, v_j) \in V^2$ be an edge with probability $p_{i,j} = d_i d_j / \sum_{k=1}^{n} d_k$, independently (cf. [7]). We assume that $d_{\min} > \log^{\delta} n$, where $\delta > 2$ is a constant, which implies that that the graph is connected with high probability[1] (e.g. [3]). Moreover, each node v_i has $d_i(1 \pm o(1))$ neighbors in $G(\mathbf{d})$, w.h.p. In this paper, we also assume that the degree distribution satisfies a certain power law, i.e., the number of nodes with degree d_i is proportional to $(d_i - d_{\min} + 1)^{-\beta}$, where $\beta > 2$ is a constant, and $d_{\max} = \Theta(n^{1/\beta})$.

As mentioned in the introduction, pull transmissions make only sense if new informations occur frequently in the network so that almost every node places a call in each round anyway. Now even though we consider applications in which informations are constantly generated by almost every node, we focus on the distribution and lifetime of a single information.

First, we describe an algorithm for the case in which we assume that every node has an estimation of $\tau = \log n/\log d_{\min}$ (n, $\mathbf{d}$, and β however, are still

[1] When we write “with high probability” or “w.h.p.”, we mean with probability at least $1 - o(1/n)$.

unknown). After we show that this algorithm works correctly, we focus on the general case, in which the nodes do not have any knowledge about the network.

The algorithm we describe in the following paragraphs contains several rounds. Before the first round, some information r is placed on one of the nodes. In each succeeding round, every node u chooses a communication partner v from the set of its neighbors, independently and uniformly at random, and establishes a communication channel with it. In the current step, any information can be exchanged in both directions along a communication channel between two communication partners. Whenever a channel is established between two nodes, each one of them has to decide whether to transmit the specific information to the other node, without knowing if the vertex at the other end of the edge has already received the information prior to this step. If a node decides to send r, then it also transmits its node ID and a constant number of other messages related to r. The rules, when r is transmitted and when not, are described below. Concerning the flow of information we distinguish between push and pull transmissions. The size of information exchanged in any way is not limited and each information exchange between two neighbors in a round is counted as a single transmission.

At the beginning, we initialize at each node u an array $T[c_{\max}]$ (where $c_{\max}$ is some constant) for storing at most $c_{\max}$ different node IDs, the integer *age*, which describes the age of the information, *itime*, representing the last time step (known to u) in which a node was newly informed in the system, and a counter *ctr* which is set initially to 0. The age is incremented in every succeeding round, by each informed node, and distributed together with the information and the integer *itime*. It should be noted that *itime* and *ctr* are local variables and may differ from node to node.

During the execution of the algorithm, each node can be in one of the states U (uninformed), A (active), G (going down), or S (sleeping). If a node is in state U, it means that it has not received the information r yet. In all other states the node knows r. Now, an arbitrary node u performs in any step t the following procedure.

1. Choose a neighbor, uniformly at random, and call this node to establish a communication channel with it. Furthermore, establish a communication channel with all nodes which call u in this step.
2. If u is in state A or G, then send to all nodes which have established a communication channel with u the message $(r, itime, age, ID(u))$.
3. Receive messages from all neighbors which have established a communication channel with u. Let these messages be denoted by $(r, itime_1, age_1, ID(v_1))$, $\ldots$, $(r, itime_k, age_k, ID(v_k))$ (if any). Then, close all communication channels.
4. Perform the following local computations:
   - 4.1. If u is in state A,G, or S, and *itime* is smaller than $\max_{1 \le i \le k} itime_i$, then set $itime = \max_{1 \le i \le k} itime_i$. If u is not in state U, then increment *age* by 1.
   - 4.2. If u is in state U and there is a neighbor v_i, which transmitted r to u, then switch state of u to A, and set *itime* and *age* to $age_i + 1$.

4.3. If u is in state A, then:
 - if u does not receive r in this step, then set $ctr = 0$ and $T[j] = 0$ for any $j = 1, \dots, c_{\max}$.
 - if u receives r in this step and there are some $i \in \{1, \dots, k\}$ such that $ID(v_i) \notin T$, then choose such an i (e.g. uniformly at random), set $T[ctr + 1] = ID(v_i)$, and increment ctr by 1. If $ctr = c_{\max}$, then switch state of u to G.

4.4. If u is in state G and age is equal to $itime + \alpha \max\{\log itime, \tau\}$, where α is a large constant, then switch state of u to S.

4.5. If u is in state S and age is smaller than $itime + \alpha \max\{\log itime, \tau\}$, then switch back to state G.

Please note that the nodes are aware of an estimate of the value $\tau = \log n / \log d_{\min}$ (the modified algorithm for the case in which nothing is known to the nodes is given in the next section).

In order to show the runtime and communication efficiency of the algorithm described above, we first state a combinatorial result w.r.t. the random graph $G(\mathbf{d})$. For some u, v let $A_{u,v}$ denote the event that u and v are connected by an edge, and let $A_{u,v,l}$ denote the event that u and v share an edge **and** u chooses v in step l (according to the random phone call model described above). In the next lemma, we deal with the distribution of the neighbors of a node u in a graph $G(\mathbf{d})$, after it has chosen t neighbors, uniformly at random, in $t = O(\log n)$ consecutive steps. In particular, we show that the probability of u being connected with some node v, not chosen within these t steps, is not substantially modified after $O(\log n)$ steps.

Lemma 1. *Let $V = \{v_1, \dots, v_n\}$ be a set of n nodes and let every pair of nodes v_i, v_j be connected with probability $p_{i,j}$, independently, where $p_{i,j}$ and $\mathbf{d}$ satisfy the conditions described at the beginning of this section. If $t = O(\log n)$, $u, v \in V$, and $A(U_0, U_1, U_2) = \bigwedge_{\substack{0 \quad \le \\ (\; , \quad ,)\in \; 0}} A_{v \; ,v \; ,l} \bigwedge_{(v \, ', v \, ')\in U_1} A_{v \, ',v \, '} \bigwedge_{(v \, '', v \, '')\in U_2} \overline{A_{(v \, '',v \, '')}}$, for some $U_0 \subset V \times V \times \{0, \dots, t\}$ and $U_1, U_2 \subset V \times V$, then it holds that*

$$\Pr\left[(u, v) \in E \mid A(U_0, U_1, U_2)\right] = p_{u,v}(1 \pm O(t/d_{\min})),$$

for any U_0, U_1, U_2 satisfying the following properties:

- $|U_0 \cap \{(v_i, v_j, l) | v_j \in V\}| = 1$ *for any* $v_i \in V$ *and* $l \in \{0, \dots, t\}$,
- $|U_1 \cap \{(u, u') | u' \in V\}| = \Omega(d_u)$ *and* $|U_1 \cap \{(v, v') | v' \in V\}| = \Omega(d_v)$,
- $(u, v) \notin U_1 \cup U_2$, *and* $(u, v, i) \notin U_0$ *for any* i.

The proof of this lemma is similar to the proof of Lemma 1 of [12], and is omitted due to space limitations. Lemma 1 implies that the probability of an edge's presence, given certain other edges and non-edges, is not substantially modified after $O(\log n)$ steps. Therefore, even if the occurrences of the edges are not necessarily independent after $t = O(\log n)$ steps, in certain cases (as in the lemmas below) we still can apply some known results which require independency (like the Chernoff bounds [6,19]) if the probabilities $p_{u,v}$ are properly approximated by $p_{u,v}(1 - o(1))$ or $p_{u,v}(1 + o(1))$.

Now, by using Lemma 1 we show the desired bounds on the runtime and communication complexity in six phases. In a first phase we prove that if r is placed at some node u at time 0, then within $O(\log n)$ rounds, at least $\log^q n$ nodes become informed, w.h.p., where q is a large constant. The communication complexity is bounded after this phase by $O(\log^{q+1} n)$.

In the second, third, and fourth phase we study the cases $|I(t)| \in [\log^q n, n^{1-\epsilon}]$, $|I(t)| \in [n^{1-\epsilon}, \frac{n}{2^{4\max\{\ ,\log\log\ \}}}]$, and $|I(t)| \in [\frac{n}{2^{4\max\{\ ,\log\log\ \}}}, \frac{n}{2}]$, respectively, where $I(t)$ denotes the set of informed nodes at time t and $\epsilon \leq 1/\beta$ is a constant. We show that in all these cases $|I(t+1)| = |I(t)|(1 + \Theta(1))$. This implies that after further $O(\log n)$ rounds the number of informed nodes becomes larger than $n/2$. Additionally, after these rounds the communication complexity is bounded by $O(n)$. We should note that these cases require different proof techniques, and hence, we have to consider these cases in different phases.

In the fifth phase, we show that if $|H(t)| \in [n/\sqrt{d_{\min}}, n/2]$, where $H(t) = V \setminus I(t)$, then after $O(\log\log n)$ rounds the number of uninformed nodes drops below $n/\sqrt{d_{\min}}$. In the sixth phase, we prove that within additional $O(\log n/\log d_{\min})$ rounds the algorithm informs all vertices.

According to the phases described before, we consider the following lemmas.

Lemma 2. *Let $I(t)$ be the set of informed nodes in $G(\mathbf{d})$ at time t, and let $|I(t)| \leq \log^q n$, where q is a properly chosen constant value. Then, within $O(\log n)$ steps the number of informed nodes becomes larger than $\log^q n$, w.h.p.*

Proof. Let u be the node at which the information r is placed at time 0. Let the tree $T_t(u) = (V', E')$ be defined in the following way: V' contains the nodes informed until time t, and there is an edge between two nodes $u', u'' \in V'$ in $T_t(u)$ if u'' is informed by u' before step $t+1$. If some node gets the information from several nodes simultaneously, then only one of them (chosen randomly) is considered to share an edge in $T_t(u)$ with this node.

Let u' be a node informed within the first $\log^q n$ nodes. Since $d_{\max} = \Theta(n^{1/\beta})$ and $\beta > 2$, it holds that $d_{u'} < \sqrt{n}$. Let $I_{u'}(t)$ denote the set of nodes which have been informed by u' before time step t. Then, since $vol(I(t)) = o(n^{1/\beta} \log^{q+1} n)$, where $vol(I(t)) = \sum_{v\ \in I(t)} d_j$ denotes the *volume* of $I(t)$, applying Lemma 1 together with the Chernoff bounds[2] [6,19] we conclude that, with probability $1 - o(1/n^2)$, there can be at most κ edges between u' and some other nodes of $I(t) \setminus I_{u'}(t)$ (as long as $|I(t)| \leq \log^q n$), where κ is a constant. Therefore, as long as $|I(t)| \leq \log^q n$, the probability that the node u' with $|I_{u'}(t)| \leq c_{\max} - \kappa - 1$ will be forced by the algorithm to switch to state G is $o(1/n^2)$.

We ignore now the probability that a node with less than $c_{\max} - \kappa - 1$ neighbors in $T_t(u)$ will switch to state G. Clearly, $T_t(u)$ has less than $|I(t)|/(c_{\max} - \kappa - 2)$ nodes with more than $c_{\max} - \kappa - 1$ neighbors in $T_t(u)$. Since a node u' with $|I_{u'}(t)| \leq c_{\max} - \kappa - 1$ propagates the information to some uninformed node

[2] Due to space limitations, we do not explain the mathematical details behind the application of Chernoff bounds here.

with probability $1 - O(1/\log^{\delta} n)$, applying Lemma 1 together with the Chernoff bounds [6,19] (similarly to [16]), we conclude that the number of informed nodes becomes larger than $\log^q n$ within $O(\log n)$ steps. □

Lemma 3. *Let $G(\mathbf{d})$ be a random graph and let $\mathbf{d}$ satisfy the conditions described at the beginning of this section. Let $|I(t)| \in \{\ln^q n, \ldots, n^{1-\epsilon}\}$, where $\epsilon < 1/\beta$ is a small constant, and assume that $|I(t)|(1 - O(t/\log^2 n))$ nodes are in state A. Furthermore, we assume that at most $O(|I(t)|/\log^{j(\beta-2)} n)$ active vertices have ctr $= j$. Then a constant c exists such that $|I(t+1)| \geq (1+c)|I(t)|$, $|I_a(t+1)| \geq |I(t+1)|(1-O((t+1)/\log^2 n))$, and at most $O(|I(t+1)|/\log^{j(\beta-2)} n)$ active vertices have ctr $= j$ after step $t + 1$, with probability $1 - o(1/n^2)$, where $I_a(t + 1)$ represents the set of nodes in state A at time $t + 1$.*

Proof. We assume for simplicity that the information is only transmitted by push transmissions. Due to the definition of $G(\mathbf{d})$ there are $n(1-O(1/\log^{2(\beta-1)}))$ nodes with expected degree less than $d_{\min} + \log^2 n$ [7]. Now, since $|I(t)| \leq n^{1-\epsilon}$, a constant ϵ' exists such that at most a fraction of $1/n^{\epsilon'}$ of the nodes having less than $d_{\min} + \log^2 n$ neighbors in $G(\mathbf{d})$ are informed. On the other hand, each node v_j of expected degree d_j has more than $d_j(1 - O(1/\log^{2(\beta-2)} n + \sqrt{\log n/d_j}))$ uninformed neighbors among the nodes which have expected degree at most $d_{\min} + \log^2 n$ in $G(\mathbf{d})$, w.h.p. [7]. Similarly, with probability $1 - o(1/n^2)$, a node v_j has at most $O(d_j/\log^{2(\beta-2)} n + \log n)$ neighbors, which are either informed or have expected degree at least $d_{\min} + \log^2 n$.

For simplicity we assume that $d_j/\log^{2(\beta-2)} n \geq \log n$. Then, by using the Chernoff bounds [6,19] we conclude that if q is large enough, then at least $|I_a(t)|(1 - O(1/\log^{2(\beta-2)} n))$ informed active vertices choose an uninformed vertex in step $t + 1$, with probability $1 - o(1/n^2)$. Similarly, there can be at most $O(|I(t)|/\log^{j(\beta-2)} n \cdot 1/\log^{2(\beta-2)} n) \leq O(|I(t)|/\log^{(j+1)(\beta-2)} n)$ vertices which have *ctr* $= j + 1$ after the t + 1st step. Now, if $c_{\max}$ is large enough, the second and third claim of the lemma follow.

In order to show the first statement, we apply Lemma 1 together with the results of [32], and conclude that, with probability $1 - o(1/n^2)$, at most $O(1)$ vertices of $I_a(t)$ choose the same uninformed node v_j of expected degree $d_j \leq d_{\min} + \log^2 n$. □

We assume in the sequel that $d_{\min} = n^{o(1)}$.

Lemma 4. *Let $I(t)$ be the set of informed nodes in $G(\mathbf{d})$ at time $t = O(\log n)$. We assume that $n^{1-\epsilon} \leq |I(t)| \leq n/2^{4\max\{\tau,\log\log n\}}$, where $\epsilon < 1/\beta$ is a constant. We also assume that $|I(t)|(1 - O(t/\log^2 n))$ nodes are in state A, and at most $O(|I(t)|/\log^{j(\beta-2)} n)$ active vertices have ctr $= j$. Then, a constant c exists such that $|I(t + 1)| \geq |I(t)|(1 + c)$, $|I_a(t + 1)| \geq |I(t + 1)|(1 - O((t + 1)/\log^2 n))$, and at most $O(|I(t + 1)|/\log^{j(\beta-2)} n)$ active vertices have ctr $= j$ after step $t + 1$, with probability $1 - o(1/n^2)$.*

Proof. We only discuss the case $\beta < 3$ here. If $\beta \geq 3$, then the proof is much simpler. Since $|I(t)| \leq n/2^{4\max\{\tau,\log\log n\}}$, there are at least $|V_{\min}|(1 -$

$O(1/2^{4\max\{\tau,\log\log n\}}))$ uninformed vertices of expected degree less than $d_{\min} + \log^2 n$, with probability $1 - o(1/n^2)$, where $V_{\min} = \{v_j \mid d_j \leq d_{\min} + \log^2 n\}$. Lemma 1 implies that every node v_j has at least $d_j(1 - O(1/\log^{2(\beta-2)} n))$ uninformed neighbors in $V_{\min}$, with probability $1 - o(1/n^2)$. Similarly, v_j has at most $O(d_j/\log^{2(\beta-2)} n + \log n)$ neighbors, which are either informed or have expected degree at least $d_{\min} + \log^2 n$.

for simplicity we assume that $d_j/\log^{2(\beta-2)} n \geq \log n$. Then, an active informed node chooses an uninformed neighbor with probability at least $1 - O(1/\log^{2(\beta-2)} n)$. If $c_{\max}$ is large enough, then we apply the same arguments as in the proof of Lemma 3, and obtain the second and third statement of the lemma.

By using Lemma 1 together with the results of [32], we conclude that $\Theta(|I(t)|)$ uninformed nodes of $V_{\min}$ are chosen by $O(1)$ informed nodes, each, and the lemma follows. □

Lemma 5. *Let $I(t)$ be the set of informed nodes in $G(\mathbf{d})$ at time $t = O(\log n)$. Assume that $|I(t)| \in [n/2^{4\max\{\tau,\log\log n\}}, n/2]$. Then, there exists a constant c such that $|I(t+1)| \geq |I(t)|(1+c)$ with probability $1 - o(1/n^2)$. Moreover, $|I(t+1)|(1 - O(1/\log n))$ vertices transmit r for at least $\frac{\alpha}{2}\max\{\tau, \log\log n\}$ further steps.*

Proof. Lemma 4 implies that if $|I(t)| \leq n/2^{4\max\{\tau,\log\log n\}}$, then $|I(t)|(1 - o(1/\log n))$ nodes are in state A. Since more than $|I(t)| - n^{1-\Omega(1)}$ informed vertices have *itime* $= \Omega(\log n)$, it holds that when $|I(t)| > n/2^{4\max\{\tau,\log\log n\}}$ for the first time, then $|I(t)|(1 - o(1/\log n))$ informed vertices are in state A, and have some proper *itime* $= \Theta(\log n)$. Therefore, if α is large enough, then all these vertices will be transmitting for $\alpha/2 \cdot \max\{\log\log n, \tau\}$ further steps, and the second statement of the lemma follows.

In order to show the first statement, we denote by $I_{a,g}(t)$ the set of informed nodes which are either in state A, or in state G with some proper *itime* $= \Theta(\log n)$. Since $|I(t)| \leq n/2$, there are at least $n(1 - o(1))/2$ uninformed nodes in $V_{\min}$ at time t, with probability $1 - o(1/n^2)$. We also know that any vertex v_j has $d_j(1 - o(1))$ neighbors among the nodes which have at most $d_{\min} + \log^2 n$ neighbors in $G(\mathbf{d})$. Applying now Lemma 1 together with the Chernoff bounds [6,19], we conclude that $\Theta(|I(t)|)$ informed vertices choose an uninformed neighbor in step $t+1$. Applying the same arguments as in the proof of Lemma 4, we obtain that $|I(t+1)| \geq |I(t)|(1+c)$. □

After we informed more than $n/2$ nodes, we only consider the vertices informed by pull transmissions as newly informed vertices. Then, we can state the following lemma.

Lemma 6. *Let $|H(t)| \in [n/\sqrt{d_{\min}}, n/2]$ be the number of uninformed nodes in $G(\mathbf{d})$ at some time $t = O(\log n)$. Let $t'' = \epsilon'' \log n$ denote the first step in which $|I(t'')| \geq n/2^{4\max\{\log n/\log d, \log\log n\}})$, where ϵ'' is a constant, and assume that there are at most $|H(t)|(1 + O(\frac{t}{\log^2 n}))$ informed nodes in state S, or in state G with itime $\leq \epsilon'' \log n$. Then, after $k = O(\log\log n)$ additional steps, the number* *of uninformed nodes $|H(t+k)|$ is less than $n/\sqrt{d_{\min}}$, and the number of nodes which are either in state S or have itime $\leq \epsilon'' \log n$ is at most $|H(t+k)|(1+o(1))$.*

Proof. Let t_0 be the first time step such that $|I(t_0)| \geq n/2$ and let D_{t_0} denote the set of vertices which already have r, but are either in state S or in state G with *itime* $< \epsilon'' \log n$. From Lemma 4 and 5 we know that $|D_{t_0}| = o(n/2^{4\max\{\log n/\log d, \log\log n\}})$. We may therefore assume that at time t_0 any node informed before step t'' is either in state S or in state G with *itime* $< \epsilon'' \log n$.

Now we only consider the vertices v_i associated with some $d_i = n^{1/\beta - \Omega(1)}$. We know that for all these i there are $n^{\Omega(1)}$ vertices with degrees in the range $[d_i - \log^2 n, d_i + \log^2 n]$, w.h.p.

Let V_i denote the set of vertices which have their expected degree in the range $[d_i - \log^2 n, d_i + \log^2 n]$, where $d_i \in \{d_{\min}, d_{\min} + 2\log^2 n, d_{\min} + 4\log^2 n, \dots\}$. Since any v_j has $d_j(1-o(1))$ neighbors in $V_{\min}$, and $|V_{\min} \cap (H(t_0) \cup D_{t_0})| \in (|V_{\min}|/4, 3|V_{\min}|/4)$, we can apply Lemma 1 together with the Chernoff bounds [6,19] to conclude that $|V_i \cap I_{a,g}(t_0 + c)| \geq |V_i|/2$ for some constant c.

Similarly, we can show that if for some t the set D_t contains the informed nodes being in state S or in state G with *itime* $< \epsilon'' \log n$, and $|D_t \cap V_{\min}| \leq |H(t) \cap V_{\min}|(1+o(1))$, then there exists a c such that $|H(t+c-1) \cap V_i|/|V_i| \leq \max\{|H(t) \cap V_{\min}|/|V_{\min}|, O(\log n)/|V_i|\}$ and the inequality $|D_{t+c-1} \cap V_i|/|V_i| \leq \max\{|D_t \cap V_{\min}|/|V_{\min}|, O(\log n)/|V_i|\}$ is also fulfilled. Combining Lemma 1 with the Chernoff bounds [6,19] we obtain that at most $|(H(t) \cup D_t) \cap V_{\min}| \cdot |H(t) \cap V_{\min}|(1+o(1))/|V_{\min}|$ nodes of $V_{\min}$ remain uninformed after step $t+c$. This yields $|H(t+c) \cap V_{\min}| \leq |H(t) \cap V_{\min}|^2(2+o(1))/n$. Similarly, $|D_{t+c} \cap V_{\min}| \leq |D_t \cap V_{\min}|^2(2+o(1))/n$. Since we assumed that $d_{\min} = n^{o(1)}$, it holds that $O(n^{1/\beta} \log^2 n) \ll n/\sqrt{d_{\min}}$. Therefore, we can ignore the vertices with expected degree $n^{1/\beta - o(1)}$, and the lemma follows. □

Lemma 7. *Let $|H(t)| \leq n/\sqrt{d_{\min}}$ be the number of uninformed nodes in $G(\mathbf{d})$ at some time $t = O(\log n)$. Then, an arbitrary uninformed node remains uninformed after step $t+1+c$, for some constant c, with probability $O(\log n/\sqrt{d_{\min}})$.*

Proof. Lemma 6 implies that if t is the first time step in which $|H(t)| \leq n/\sqrt{d_{\min}}$, then $|D_t| \leq n(1+o(1))/\sqrt{d_{\min}}$. The arguments of the proof of Lemma 6 imply that, with probability $1 - o(1/n^2)$, a constant c exists such that $|H(t+c) \cap V_i| \leq \max\{\frac{|V_i|}{\sqrt{d_{\min}}}, O(\log n)\}$ and $|D_{t+c} \cap V_i| \leq \max\{\frac{|V_i|}{\sqrt{d_{\min}}}, O(\log n)\}$ for any i such that $d_i \in \{d_{\min}, d_{\min} + 2\log^2 n, \dots\}$. Lemma 1 implies together with the results of [32] that $|H(t+c) \cap V_i|(1-o(1)) - O(\log n)$ uninformed vertices of each V_i have at least $d_i - O(\sqrt{d_i \log n} + d_i/\sqrt{d_{\min}} + \log^2 n)$ neighbors in $I_{a,g}(t+c)$, and the lemma follows. □

Summarizing the results of Lemmas 2 - 7, we obtain the following theorem.

Theorem 1. *Let $G(\mathbf{d}) = (V, E)$ be a random graph with the properties described at the beginning of this section, and assume that every node knows the value $\tau = \log n/\log d_{\min}$. Then, the algorithm described at the beginning of this section spreads any information r to all nodes of the graph within $O(\log n)$ steps, and by using $O(n \max\{\log\log n, \tau\})$ transmissions, w.h.p.*

We assumed in this section that during the algorithm proceeds the nodes are aware of an estimate of τ. In the next section we present a method which allows the nodes to determine the desired estimate while the algorithm is executed.

3 Fully Adaptive Broadcasting Algorithm

In the following paragraphs we describe the fully adaptive broadcasting algorithm. Let any node choose a neighbor, uniformly at random, in round 0 and 1, respectively. Then, we let each node compare the neighbors ID, chosen in round 1, with the ID of the neighbor selected in round 0. If the two IDs at some node w are the same, then w sends out a special information r_w. Additionally, each node of the graph sends out a special message r'_w. These special messages (of type r_w and r'_w) perform random walks in the system and some node $w' \in V$ checks how many of these messages are lying on it at some time $t = q' time_{w'}(r)$, where q' is a large constant and $time_{w'}(r)$ denotes the time when w' has got r. Now, if $time_{w'}(r)$ is large enough (i.e., $time_{w'}(r) = \Omega(\log n)$), then any such message lies on w' with probability $d_{w'} / \sum_{k+1}^{n} d_k (1 \pm o(1/n^2))$ [3,13]. Combining the results of [13] with [28], we conclude that if $time_{w'}(r)$ is large enough, then some nodes of $G(\mathbf{d})$, which do not possess any r'_w message at this time, have $\Theta(\log n / \log d_{\min})$ messages of type r_w. Moreover, all nodes (for which $time_{w'}(r)$ is large enough) with no r'_w's lying on them, will have at most $O(\log n / \log d_{\min})$ messages of type r_w. Please note that in each round there are $O(n)$ transmissions based on the random walk of the special messages. Since every node chooses a communication partner in each round anyway, the overall communication complexity does not increase substantially in the system. Now we modify the algorithm described in the previous section so that some nodes compute an estimate on $\log n / \log d_{\min}$ during the algorithm proceeds and broadcast the information to the other nodes. Then, almost all nodes will use this value while being in state G.

Now, we describe informally the fully adaptive algorithm. A formal description of a similar algorithm can be found in [12]. In this algorithm, every node performs three phases, each of them consisting of several rounds. In the first phase, when a node u gets r, then it sets $time_u(r)$ to the current age of r, and the value $time_u(r)$ will never be updated again. Each node u executes the algorithm described at the beginning of this section, whereby we introduce the following small modification: When u switches to state G, then $itime$ will never be updated at u in the first phase again. This modification implies that u cannot switch from state S to state G in this first phase.

For the second phase, define $s_i = 2^i$, $i \in \{0, \ldots, q' \log\log n\}$. We call two numbers, j_1 and j_2, $s2$-equivalent, and denote $j_1 \sim_{s_2} j_2$, if an i exists such that $s_i \leq j_1, j_2 \leq s_{i+1}$. Two nodes w' and w'' are $s2$-equivalent if $time_{w'}(r)$ and $time_{w''}(r)$ are $s2$-equivalent. Now, each node u checks while being in states A and G (in the first phase), whether $c_{\max}$ different nodes exist, which are $s2$-equivalent with u, have not been informed by u but are contacted by u, and transmit r to u. If u has seen $c_{\max}$ such nodes before it switches to state S, then u checks after $\alpha \cdot itime$ additional steps the number of r_w's and the number of r'_w's on it. If there are no r'_w's on u, then the local variable τ'' is set to the number of r_w's on u. If there is any r'_w on u, then τ'' is set to 0. If $\tau'' > \log itime$, then the node switches in step $8\alpha\lfloor time_u(r)/4\rfloor + h$ to the special state R (if it is not already in this state) and to state A (even if it was in state R before), updates *itime*, sets $\tau' = \tau''$, and starts to transmit r as in the first phase (along with *age* and *itime*), together with τ'. Here, h depends on τ'' and will be defined later. We should mention that a node in the special state R is, apart from this special state, in one of the states A, G, or S.

If a node u, which is not in state R, receives r and some value τ' from a node which is in state R, then u switches to R and A, sets its own τ' to the received τ', and *itime* is updated to the actual time. As *itime* and *ctr*, the variables τ' and τ'' are local variables, which may be different from node to node.

If a node in state R and A switches to state G, then it transmits r as long as $age < itime + \alpha\tau'$. During the second phase, the received τ' is always compared to the own τ' and, if necessary (i.e., the received τ' is larger than the own τ'), the own τ' is updated. As in the first phase, *itime* is not updated after a node switches to state G, excepting the following case. If τ' has to be reset, and the own τ' is not $s2$-equivalent with the received τ', then the own *itime* is reset to the actual time.

As mentioned above, the value of h depends on τ''. Define $h_1 = \lfloor time_u(r)/4\rfloor$ for a node u. If $\tau'' \in [\log h_1 + 1, 2\log h_1]$ then $h = 0$. For $\tau'' \in [2\log h_1 + 1, 4\log h_1]$ we set h to $\alpha h_1/\log h_1$, and generally, if $\tau'' \in [2^i \log h_1 + 1, 2^{i+1}\log h_1]$, then $h = i\alpha h_1/\log h_1$. We further assume that the value h_1 is also transmitted in state R and every node keeps only the largest h_1 value ever transmitted to it.

The third phase begins at time $16\alpha h_1$ for every node being in state R. At this time, the nodes being in state R switch to the special state R' and A, and run the algorithm as described at the beginning of this section, i.e., *itime* is updated in state G, if the received τ' is larger than and not $s2$-equivalent with the own τ', **or** the τ's are $s2$-equivalent and the received *itime* is larger than the own *itime*. Now, a node is in state G as long as $age < itime + \alpha\tau'$. Here, τ' represents the own τ' value of the node which may be updated according to the rules described above.

Please note that the algorithm described above is (apart from the existence of special messages r'_w) the same as the fully adaptive algorithm of [12]. Now we can state the following theorem.

Theorem 2. *Let $G(\mathbf{d})$ be a random graph with the properties described at the beginning of Section 2. The three-phase algorithm described in this section informs all nodes of the graph within $O(\log n)$ steps, whereby its communication complexity is bounded by $O(n \max\{\log\log n, \tau\})$, with probability $1 - 1/n^{\Omega(1)}$.*

The proof of this theorem is omitted due to space limitations. Combining the techniques of [23] with the methods used in our proofs, it can be shown that the bound of Theorem 2 is asymptotically tight. The algorithms presented in this paper can also be used to in a more natural random power law graph model. If we consider the graph $G(\mathbf{d})$, where $d_{\min} \geq \log^{\delta} n$ and the number of vertices with degree d_i is proportional to $d_i^{-\beta}$, $\beta > 2$ constant, then the result of Theorem 2 holds for this graph, too. We omit the details due to space limitations.

Using a more sophisticated analysis, similar results can be obtained on the dynamic random graph model D of [1]. If we denote by G_t the graph obtained after t steps, then two low degree nodes will be connected with probability $\Theta(d_{\min}(t)/n)$, where $d_{\min}(t)$ is the minimum degree in G_t, whenever the number of edges, which are drawn from the node being assigned to the graph at some time t', is large enough ($\geq \log^{\delta} n$). Although the edges do not occur independently from each other, similar techniques to those used in our proofs can be applied to generalize our results to G_t. Again, we omit the details due to space limitations.

4 Conclusion

In this paper, we analyzed the performance of randomized broadcasting algorithms in certain (truncated) random power law graphs. First, we have shown that, whenever **d** satisfies a certain power law distribution, we are able to broadcast a message to all nodes of a random graph $G(\mathbf{d})$ within $O(\log n)$ steps, by using $O\left(n \max\left\{\log\log n, \frac{\log n}{\log d_{\min}}\right\}\right)$ transmissions, with probability $1 - 1/n^{\Omega(1)}$. Inspecting our proofs, we observe that the assumption $d_{\min} \geq \log^{\delta} n$ with $\delta > 2$ is strictly needed (e.g. Lemmas 1 and 7). A careful revision of our lemmas would also yield the main result for a graph $G(\mathbf{d})$ with $d_{\min} \geq \delta' \log n$, where δ' is a very large constant. In the case when $d_{\min} = (2 + \Theta(1)) \log n$ but $d_{\min} \leq \delta' \log n$, the graph is still connected with high probability, however we cannot apply the techniques of this paper to obtain the desired result.

References

1. W. Aiello, F.R.K. Chung, and L. Lu. Random evolution in massive graphs. In *Proc. of FOCS'01*, pages 510–519, 2001.
2. A. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.
3. B. Bollobás. *Random Graphs.* Academic Press, 1985.
4. B. Bollobás, O. Riordan, J. Spencer, and G. Tusnády. The degree sequence of a scale-free random graph process. *Rand. Struct. Alg.*, 18:279–290, 2001.
5. D. Callaway, J. Hopcroft, J. Kleinberg, M. Newman, and S. Strogatz. Are randomly grown graphs really random? *Physical Review E*, 64, 051902, 2001.
6. H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Stat.*, 23(1):115–126, 1952.
7. F.R.K. Chung and L. Lu. Connected components in random graphs with given expected degre sequences. *Annals of Combinatorics*, 6:125–145, 2002.
8. F.R.K. Chung and L. Lu. The average distances in random graphs with given expected degrees. *Internet Mathematics*, 1:91–114, 2003.
9. F.R.K. Chung, L. Lu, and V. Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7:21–33, 2003.
10. F.R.K. Chung, L. Lu, and V. Vu. The spectra of random graphs with given expected degrees. *Proc. National Academy of Sciences*, 100:6313–6318, 2003.

11. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of PODC'87*, pages 1–12, 1987.
12. R. Elsässer. Bounding the communication complexity of randomized broadcasting in random-like graphs. *Proc. of SPAA*, 2006.
13. R. Elsässer, B. Monien, and S. Schamberger. Load balancing of indivisible unit size tokens in dynamic and heterogeneous networks. *Proc. of ESA'04*, pages 640–651, 2004.
14. P. Erdős and A. Rényi. On random graphs I. *Publ. Math. Debrecen 6*, pages 290–297, 1959.
15. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *Comput. Commun. Rev.*, 29:251–263, 1999.
16. U. Feige, D. Peleg, P. Raghavan, and E. Upfal. Randomized broadcast in networks. *Random Structures and Algorithms*, 1(4):447–460, 1990.
17. L. Gasieniec and A. Pelc. Adaptive broadcasting with faulty nodes. *Parallel Computing*, 22:903–912, 1996.
18. E.N. Gilbert. Random graphs. *Ann. Math. Statist. 30*, pages 1141–1144, 1959.
19. T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 36(6):305–308, 1990.
20. M. Henzinger. Algorithmic challenges in web search engines. *Internet Mathematics*, 1:493–507, 2003.
21. H.W. Hethcore. Mathematics of infectious diseases. *SIAM Review 42*, pages 599–653, 2000.
22. H. Jeong, B. Tomber, R. Albert, Z. Oltvai, and A.L. Barabási. The large-scale organization of metabolic networks. *Nature*, 407:378–382, 2000.
23. R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized rumor spreading. *Proc. of FOCS'00*, pages 565–574, 2000.
24. D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. *Proc. of STOC'01*, pages 163–172, 2001.
25. W.O. Kermack and A.G. McKendrick. Contributions to the mathematical theory of epidemics. *Proc. Roy. Soc.*, pages 700–721, 1927.
26. J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagoplan, and A. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. of COCOON'99*, pages 1–17, 1999.
27. T. Leighton, B. Maggs, and R. Sitamaran. On the fault tolerance of some popular bounded-degree networks. In *Proc. of FOCS'92*, pages 542–552, 1992.
28. M.D. Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing.* PhD thesis, University of California at Berkeley, 1996.
29. M. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.
30. M. E. J. Newman. The spread of epidemic disease on networks. *Phys. Rev. E 66, 016128*, 2002.
31. B. Pittel. On spreading rumor. *SIAM J. Appl. Math.*, 47(1):213–223, 1987.
32. M. Raab and A. Steger. "Balls into bins" a simple and tight analysis. In *Proc. of RANDOM'98*, pages 159–170, 1998.

Distributed Approximation Algorithms in Unit-Disk Graphs

A. Czygrinow[1] and M. Hańćkowiak[2]

[1] Department of Mathematics and Statistics
Arizona State University
Tempe, AZ 85287-1804, USA
andrzej@math.la.asu.edu

[2] Faculty of Mathematics and Computer Science
Adam Mickiewicz University
Poznań, Poland
mhanckow@amu.edu.pl

Abstract. We will give distributed approximation schemes for the maximum matching problem and the minimum connected dominating set problem in unit-disk graphs. The algorithms are deterministic, run in a poly-logarithmic number of rounds in the message passing model and the approximation error can be made $O(1/\log^k |G|)$ where $|G|$ is the order of the graph and k is a positive integer.

1 Introduction

In this paper we will give efficient distributed approximation algorithms for two important graph-theoretic problems, the Maximum Matching (MM) Problem and the Minimum Connected-Dominating Set (MCDS) Problem. Our algorithms work in the message passing model and assume that the underlying network has a unit-disk graph topology. Both problems are classical problems in graph theory with many important practical applications. For example, efficient solutions for the MCDS Problem are particularly interesting because of their applications to routing in mobile ad-hoc networks (see for example [DW04]).

Studying distributed message passing approximation algorithms for unit-disk graphs was initiated by Kuhn et. al. in [KMNW05b] where efficient approximation schemes for the Maximum Independent Set (MaxIS) Problem and the Minimum Dominating Set (MDS) Problem are given. Research described in the series of papers [KMW05], [KMNW05b], and [KMNW05a] is the main motivation for our work here. We give efficient distributed approximation schemes for two additional classical graph-theoretic problems. In addition, both of our procedures are based on a rather general clustering framework that almost immediately gives efficient solutions to MaxIS and MDS Problems with potential for further applications to unit-disk graphs. Second motivation for our study comes from a recent work [CH06] and [CHS06] in which distributed approximations for MaxIS Problem, MM Problem, and MDS Problem in planar graphs

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 385–398, 2006.

are given. Unit-disk graphs form another important class of graphs where simple clustering methods give efficient distributed approximations. Since unit-disk graphs are commonly used to model mobile ad-hoc networks or static radio networks this line of research is also of clear practical importance. As in the case of [KMNW05b] or [KMW05], we assume that there is a lower level protocol that resolves transmission conflicts (MAC) and communication is done in the message passing model.

1.1 Model and Notation

We consider a standard, distributed, synchronous message-passing model described for example in Linial [L92] or in [P00]. In this model a network is represented by an undirected graph where vertices correspond to processors and edges to communication links between them. We assume that vertices have unique identifiers and that communication is synchronized. In a single round of an algorithm, each vertex of the graph can send messages to its neighbors, can receive messages from its neighbors, and can perform local computations. Although the above model allows unlimited local computations, we will pay attention not only to the distributed complexity of a problem but will also indicate the sequential running time at each processor. We will focus entirely on the time complexity analysis leaving message complexity issues for future research. Although the above model is certainly an oversimplified version of a real-life system, it perfectly captures the most fundamental challenge faced by a distributed algorithm; the problem of finding a global solution in a network based on local information about the topology of the graph which is available to each node.

In this paper we will consider graphs with the unit-disk graph topology. A graph $G = (V, E)$ is called a unit-disk graph if there is a function $f : V \to R^2$ such that $uv \in E$ if and only if $||f(u) - f(v)||_2 \leq 1$. Although the fact that G is a unit-disk graph is critical to our analysis, we will assume that no information about geometrical representation of G is available to nodes. In fact, if such information is available, distributed running time of our algorithms can be reduced significantly as it is possible to find a maximal independent set in a much faster way [KMW05].

Finally, we will use standard graph-theoretic notation and terminology. In particular, following the convention from [D97], we will denote by $|G|$ the number of vertices in G and by $||G||$ the number of edges.

1.2 Results

We will describe efficient distributed algorithms for the MM and MCDS problems in unit-disk graphs. All algorithms are deterministic and the running time is poly-logarithmic in the order of the graph. Depending on the desired approximation ratio the time is a polynomial in $\log |G|$ of higher or smaller order. In the same way, the amount of local computations can be made more or less time consuming based on the desired approximation error. For the MM Problem, we give a deterministic distributed algorithm which given a positive integer k and real number $\alpha \geq 0$ finds in a unit-disk graph $G = (V, E)$ a set $M \subset E$ such that M is a matching in G and $|M| \geq (1 - \alpha - O(1/\log^k |G|))\beta(G)$ where $\beta(G)$ is the size of the maximum matching in G (Theorem 2). For the MCDS Problem, we give a deterministic distributed algorithm which given a positive integer k and real number $\alpha \geq 0$, finds in a connected unit-disk graph G finds a set $D \subset V(G)$ such that D induces a connected subgraph of G, D is a dominating set in G, and $|D| \leq (1 + \alpha + O(1/\log^k |G|))\gamma_c(G)$ where $\gamma_c(G)$ is the size of the smallest connected dominating set in G (Theorem 3). Both algorithms run in a polylogarithmic (in $|G|$) number of rounds and the amount of local computations is at most $T^M_\alpha(|G|)$ or $T^C_\alpha(|G|)$, where $T^M_\alpha(|G|)$ is the sequential complexity of finding a $(1 - \alpha)$-approximation of a maximum matching in unit-disk graphs and $T^C_\alpha(|G|)$ is the sequential complexity of finding a $(1 + \alpha)$-approximation of a minimum connected dominating set in unit-disk graphs. In the case of maximum matching $\alpha = 0$ can be accomplished in polynomial time and although finding a minimum connected dominating set in unit-disk graphs is NP-complete (see [L82]), there are very fast constant error approximations and in the case a local geometric representation of the graph is available, $(1 + 1/s)$-approximation can be found in $O(|G|^{O((s \log s)^2)})$ time (see [CHLWD03]). In addition, our methods yield a sequential polynomial time approximation scheme (PTAS) for the MCDS Problem extending the work of Nieberg and Hurink from [NH05] where a PTAS for the MDS Problem is given.

1.3 Related Work

There has been recently an explosive growth of interest in complexity issues of algorithms for geometric graphs. This line of research has roots in computational geometry (see for example [GGHZZ01]) as well as in wireless networks. Distributed algorithms in the message passing model for unit-disk graphs have been recently studied by Kuhn et. al. in [KMW05], [KMNW05a], and [KMNW05b]. In particular, [KMNW05a] contains a $O(\log \Delta(G) \log^* |G|)$-time distributed algorithm for the maximal independent set problem. In [KMNW05b] in turn, authors gave distributed approximation schemes for the MaxIS Problem and the MDS Problem. All of the results, as well methods described in this work, exploit the bounded growth property of unit-disk graphs (see [KMW05] for an example of formalization or Section 2 for a different approach). Although methods use the same properties which are inherent to unit-disk graphs, the approaches differ significantly. In particular, algorithms from [KMNW05b] work entirely in an underlying network which has the unit-disk graph property. In contrast, the first phase of our algorithms works in an auxiliary graph which arises from a maximal independent set in a graph and it is this auxiliary graph which is further clustered to obtain partition of the original graph.

1.4 Organization

In the rest of the paper we shall first give our clustering algorithm (Section 2) and then discuss algorithms for the MM and MCDS problems (Section 3). Finally we conclude with brief remarks on how the methods can be used to give a PTAS for the MCDS Problem in unit-disk graphs.

2 Clustering Algorithm

In this section, we will give a distributed algorithm that finds a clustering of a unit-disk graph. By clustering we mean a partition of the vertex set of a graph with the property that each set in the partition induces a connected subgraph. Of course, additional properties of the clustering will be important to obtain approximation algorithms. The clustering procedure works in two phases. In the first phase we use the $O(\log \Delta(G) \log^* |G|)$-time algorithm from [KMNW05a] to find a maximal independent set I in graph G. In the second phase an auxiliary graph is constructed from I and we invoke a clustering algorithm to find a a partition of it. The clustering of the auxiliary graph gives the final a clustering of G. The analysis of our algorithms heavily exploits the so-called bounded growth property of an auxiliary graph arising from a unit-disk graphs. Consequently the method described here is limited to unit-disk graphs or some natural generalizations. The following fact about unit-disk graphs will be key to our analysis.

Lemma 1. *Let G be a unit-disk graph and let k be a positive integer. For any independent set I in G and any geometrical representation of G, the number of vertices from I which are contained in a ball in R^2 of radius k is at most $4(k+0.5)^2$.*

Although the constant 4 is certainly not best possible it is sufficient for out considerations and is easy to prove. Indeed, any packing of balls with radius 0.5 into a ball of radius $(k+0.5)$ can have at most $4(k+0.5)^2$ members. The second phase of clustering works entirely in the auxiliary graph obtained in the first phase and only after partition of the auxiliary graph is found we return to the original unit-disk graph G. We will first describe the second phase and then give the main clustering algorithm.

2.1 Second Phase of Clustering Algorithm

In this section, we give a clustering algorithm of a C-bounded growth graph (see the definition below). It is important to mention that our notion of a C-bounded growth graph is different and maybe less standard than a similar concept considered in [KMNW05a].

Definition 1. *A graph H has a C-bounded growth if for every vertex v from H and every nonnegative integer k the number of vertices within distance (in H) k of v is at most Ck^2+1.*

Our algorithm for graphs with C-bounded growth will use the ruling set method described in [AGLP89]. Let $G=(V,E)$ be a graph with identifiers of vertices from the set $\{1,\ldots,m\}$ where m is globally known and satisfies $|G| \le m \le poly(|G|)$ for some polynomial $poly(|G|)$. A *D-ruling set* in G is a subset S of V with two properties:

- For any two distinct vertices s, s' from S, the distance (in G) between s and s' is at least D.
- For any vertex $v \in V \setminus S$ there is a vertex $s \in S$ such that the distance between s and v is at most $D \log |G|$.

There is an easy distributed algorithm which finds a D-ruling set in any graph.

Theorem 1 ([AGLP89]). *There is a distributed algorithm which in any graph G finds a D-ruling set in $O(D \log |G|)$ rounds.*

Our algorithm uses parameters ϵ, D, and F (used in CLUSTERING) which can be set to specific values yielding different running times and approximation factors. For $0 < \epsilon < 1$, and fixed C, let l_* be the smallest positive integer with the property

$$(1+\epsilon)^{l_*} \geq C l_*^2 + 1. \tag{1}$$

It is easy to check that

$$l_* = O(1/\epsilon^2). \tag{2}$$

In addition, let D be such that

$$D > 2 l_*. \tag{3}$$

In the next two procedures we will find a clustering of a graph which has C-bounded growth. First procedure is essentially one iteration of the main algorithm. We will denote the set of vertices which have a neighbor in U by $N(U)$.

```
ClusterSet
Input: Constant $C$. Graph $H = (V, E)$ which has $C$-bounded growth and such
that identifiers of $V$ are bounded by $m$. Parameters: $0 < \epsilon < 1$ (arbitrary) and
$D$ (must satisfy (3)).
Output: A family of subsets of $V$.

(1) Find a $D$-ruling set, $\{v_1, v_2, \ldots, v_s\}$ in $H$.
(2) For every $v_i$ in parallel:
    (a) Let $U_i := \{v_i\}$, $N_i := N(U_i) \setminus U_i$.
    (b) while $|N_i| \geq \epsilon |U_i|$
          • $U_i := U_i \cup N_i$. $N_i := N(U_i) \setminus U_i$.
(3) Return $U_1, U_2, \ldots, U_s$.
```

The analysis of CLUSERSET can be divided into a few lemmas.

Lemma 2. *The number of vertices in the D-ruling set obtained in step one of* CLUSTERSET *is at least*

$$|H|/(CD^2 \log^2 |H| + 1).$$

Proof. For every vertex v_i, where $i = 1, \ldots, s$ from the ruling set, consider the set W_i of vertices in H which are within distance $D \log |H|$ of v_i. From Definition 1, $|W_i| \leq CD^2 \log^2 |H| + 1$. On the other hand, since v_i's form a D-ruling set, $|H| \leq |\bigcup_{i=s}^{l} W_i|$ and so

$$|H| \leq s(CD^2 \log^2 |H| + 1).$$

Next two lemmas show that U_i's have small diameter and more importantly the total number of edges that intersect two different U_i's is small.

Lemma 3. *Let l_* be such that inequality (1) holds and let $r(U_i)$ denote the radius of $H[U_i]$. Then*

$$r(U_i) \leq l_*.$$

Proof. Let $U_i^{(l)}$ denote the set U_i in the lth iteration of the while loop from step 2(b). Then $|U_i^{(0)}| = 1$ and in the lth iteration $|U_i^{(l)}| \geq (1+\epsilon)^l$. On the other hand, by Definition 1, $|U_i^{(l)}| \leq Cl^2 + 1$. Consequently, if l_* is the smallest positive integer such that $(1+\epsilon)^{l_*} \geq Cl_*^2 + 1$ then $l \leq l_*$. Since the radius of $G[U_i]$ is at most l, we have $r(U_i) \leq l_*$.

Lemma 4. *Sets U_i returned by* ClusterSet *are pair-wise disjoint. In addition, if $e(U_i, V \setminus U_i)$ denotes the number of edges between U_i and $V \setminus U_i$ then*

$$\sum_{i=1}^{s} e(U_i, V \setminus U_i) \leq C\epsilon |\bigcup_{i=s}^{l} U_i|.$$

Proof of Lemma 4. From Lemma 3 every U_i is such that $r(U_i) \leq l_*$. and so if $U_i \cap U_j$ is non-empty then the distance between v_i and v_j is at most $2l_*$ which contradicts the fact that v_i, v_j are in the D-ruling set where $D > 2l_*$ by (3). To prove the second part, note that for every U_i returned in step 3, the set $N_i = N(U_i) \setminus U_i$ is such that $|N_i| < \epsilon |U_i|$. Since H has the maximum degree of at most C, the number of edges between U_i and N_i is at most $C\epsilon|U_i|$.
Finally, we note that the running time of ClusterSet is $O(D \log m + 1/\epsilon^2)$.

Lemma 5. *The number of rounds of* ClusterSet *is $O(D \log m + 1/\epsilon^2)$.*

Proof. There are $O(D \log m)$ rounds to find the D-ruling set in step 1. This is followed by $l_* = O(1/\epsilon^2)$ iterations in step 2.

Our main clustering procedure will call ClusterSet a repeated number of times. In each call, sets $U_1, \ldots, U_l$ are obtained and vertices from $\bigcup U_i$ are deleted from the graph H. Finally, after trimming H with repeated application of ClusterSet, the remaining vertices will form one-element clusters.

Clustering
Input: Constant C. Graph $H = (V, E)$ which has C-bounded growth and such that the identifiers of V are less than or equal to m. Parameters: $0 < \epsilon < 1$ (arbitrary), D (must satisfy (3)), F (arbitrary).
Output: A partition $\mathcal{P}$ of V.

```
(1) Repeat $F$ times:
    (a) Call ClusterSet in $H$. Add all sets $U_i$ obtained from ClusterSet to family $\mathcal{P}$.
    (b) Delete from $H$ vertices from $\bigcup U_i$ and edges incident to these vertices.
(2) For every vertex left in $H$ create a set which contains only this vertex and add it to $\mathcal{P}$. Return $\mathcal{P}$.
```

We note the following property of the partition obtained by Clustering.

Lemma 6. *Let $\mathcal{P} = (V_1, \ldots, V_t)$ be a partition of V returned by* Clustering. *The number of edges of H connecting vertices from different V_i's is*

$$O\left(\left(\left(1 - \frac{1}{CD^2 \log^2 |H| + 1}\right)^F + \epsilon\right)|H|\right).$$

Proof. First note that since H is a graph with a constant maximum degree, $||H|| \leq C|H|/2$. Consider sets added to $\mathcal{P}$ in iterations from step 1. Edges which have exactly one endpoint in these sets are deleted in step 1(b) and by Lemma 4, the number of them is at most $C\epsilon|H|$. The remaining edges which must be counted are the edges of H from step two. To estimate these, we note that by Lemma 2, the number of vertices in this graph is $O\left(\left(1 - \frac{1}{CD^2 \log^2 |H|+1}\right)^F |H|\right)$. Consequently, the number of edges of H connecting vertices from different V_i's is $O\left(\left(\left(1 - \frac{1}{CD^2 \log^2 |H|+1}\right)^F + \epsilon\right)|H|\right)$.

Lemma 7. Clustering *runs in $O\left(F\left(D \log m + 1/\epsilon^2\right)\right)$ rounds.*

Proof. There are F iterations of step 1, in which, by Lemma 5, sets are found in $O\left(D \log m + 1/\epsilon^2\right)$ rounds.

Corollary 1. *Let C be a fixed constant. For a C-bounded growth graph H with identifiers that are less than or equal to m, there is a distributed algorithm which finds in $O(1/\epsilon^6 \cdot \log^3 m \log 1/\epsilon)$ rounds a partition of V such that the number of edges between different partition classes is $O(\epsilon|H|)$.*

Proof. Let $D := 2l_* + 1 = O(1/\epsilon^2)$ and let $F := \lceil (\ln 1/\epsilon)(CD^2 \log^2 H + 1) \rceil = O((\log 1/\epsilon) D^2 \log^2 m)$. Then the number of rounds is $O(F \log m/\epsilon^2) = O(1/\epsilon^6 \cdot \log^3 m \log 1/\epsilon)$ (Lemma 7). In addition, the number of edges which connect different clusters is $O(\epsilon|H|)$ by Lemma 6 as $\left(1 - \frac{1}{CD^2 \log^2 |H|+1}\right)^F = O(\epsilon)$.

2.2 Main Clustering Algorithm

We will now give the main clustering algorithm for unit-disk graphs. We first find a maximal independent set I using an algorithm from [KMNW05a] and then apply Clustering in the auxiliary graph that arises from I. This gives clusters in the original graph.

Definition 2 (Auxiliary graph). *Let $I = \{v_1, \ldots, v_l\}$ be a maximal independent set in graph $G = (V, E)$. Let V_i be the set of neighbors of v_i such that if $w \in V_i$ then v_i is the neighbor of w in I with the least identifier, i.e. $V_i = \{w \in N(v_i) | ID(v_i) = \min\{ID(a) | a \in N(w) \cap I\}\}$, and let $\bar{V_i} = V_i \cup \{v_i\}$. Let $Aux(G)$ be the graph $(\mathcal{W}, \mathcal{E})$ with $\mathcal{W} = \{\bar{V_1}, \ldots, \bar{V_l}\}$ and $\{\bar{V_i}, \bar{V_j}\} \in \mathcal{E}$ whenever $i \neq j$ and there is an edge in G between a vertex from $\bar{V_i}$ and a vertex from $\bar{V_j}$.*

From Lemma 1, we see that $Aux(G)$ has C-bounded growth with $C = 48$ as if there are p vertices within distance k of some vertex v in $Aux(G)$ then there are is an independent set of size p in a ball of radius $3k$. Consequently $p \leq 4(3k + 0.5)^2 \leq 48k^2 + 1$.

Lemma 8. *$Aux(G)$ has 48-bounded growth.*

In particular the maximum degree of $Aux(G)$ is at most 48.

ClusteringUDG
Input: Unit disk graph $G = (V, E)$, $0 < \epsilon < 1$ (arbitrary).
Output: Partition $\mathcal{Q}$ of V.

(1) Call Kuhn et. al. algorithm from [KMNW05a] to find a maximal independent set I. Consider the auxiliary graph $Aux(G)$.
(2) Call Clustering with constants as in Corollary 1 to find a partition $\mathcal{P}$ of $Aux(G)$.
(3) As vertices in $Aux(G)$ correspond to pair-wise disjoint subsets of vertices in G, for every partition class from $\mathcal{P}$ add the union of the subsets of vertices from G that are contained in this class to $\mathcal{Q}$.

Clusters obtained by ClusteringUDG have additional properties which are very useful in our analysis. These attributes, however, must be stated in terms of clusters in $Aux(G)$ rather than clusters in G. In particular, it is not true that the number of edges connecting different clusters in G is "small" with respect to the total number of edges but the property holds in $Aux(G)$. Intuitively, what we need from clusters in G is that an objective function to be approximated (for example the size of a connected dominating set) has a small value on the boundary of each cluster. At this moment let us indicate the running time of ClusteringUDG and we will use previous lemmas to extract useful properties when they are needed.

Lemma 9. *Let $G = (V, E)$ be a unit-disk graphs with identifiers bounded by $m = poly(|G|)$. ClusteringUDG runs in $O(1/\epsilon^6 \cdot \log^3 |G| \log 1/\epsilon)$ rounds.*

Proof. Kuhn et. al. algorithm runs in $O(\log |G| \log^* |G|)$ rounds. The complexity of Clustering is $O(1/\epsilon^6 \cdot \log^3 |G| \log 1/\epsilon)$ in $Aux(G)$. Since every vertex in $Aux(G)$ corresponds to a subgraph of diameter which is less than or equal to two, the running time in G will be asymptotically the same.

3 Applications

In this section, we will describe applications to the MM Problem and the MCDS Problem. Approximations for both problems (as well as to the maximum independent set and the minimum dominating set) follow a similar pattern: First find a clustering using ClusteringUDG and then find an optimal solution locally in each cluster. Finally, modify local solutions and return the union of them. Since our main clustering algorithm is executed in the auxiliary graph which arises from yet another clustering of G obtained from a maximal independent set, we will have three types of clusters.

- *Small clusters*: Clusters in graph G which arise from the maximal independent set obtained in phase one and correspond to vertices in $Aux(G)$.
- *Auxiliary clusters*: Clusters in $Aux(G)$ obtained by Clustering.
- *Big clusters*: Clusters in G obtained by ClusteringUDG. These clusters correspond in an obvious way to clusters in $Aux(G)$.

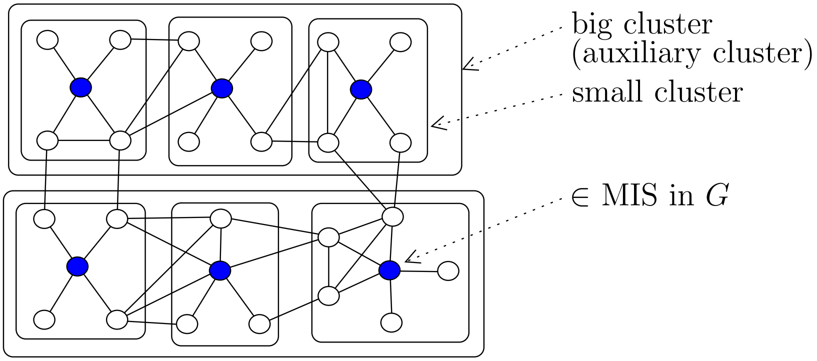


Fig. 1. Clusters in UDG

Analysis of algorithms relies on properties of $Aux(G)$ and it will be useful to develop more terminology related to clusters of $Aux(G)$. Let v be a vertex in $Aux(G)$. If v is in cluster C of $Aux(G)$ but has a neighbor in a different cluster of $Aux(G)$ then v will be called a *border vertex* and the small cluster in G which corresponds to it will be called a *border cluster*. We note, that since $Aux(G)$ has a constant maximum degree, Lemma 6 implies that the number of border vertices in $Aux(G)$ is much smaller than $|Aux(G)|$.

3.1 Maximum Matching

In our first application we will approximate a maximum matching. Recall that if M is a matching in graph G then a vertex is called M-saturated if it is an endpoint of an edge from M. Otherwise it is called M-free.

```
ApproxMaxMatching
Input: Unit-disk graph $G = (V, E)$, $0 < \epsilon < 1$, $0 \leq \alpha < 1$.
Output: A matching in $G$.

(1) Call ClusteringUDG to find a clustering of $G$ with constants as in Corollary 1.
(2) In each cluster $C$, find a matching $M_C$ in the subgraph of $G$ induced by $C$
    which is a $(1 - \alpha)$-approximation of a maximum matching in $G[C]$.
(3) Return $M := \bigcup_C M_C$.
```

Note that the second step of the procedure takes *polylog*$(|G|)$ rounds as the diameter of each big cluster is *polylog*$(|G|)$.

Lemma 10. *Let G be a unit-disk graph. The matching M returned by* ApproxMaxMatching *satisfies*

$$|M| \geq (1 - \alpha - O(\epsilon))\beta,$$

where β is the size of a maximum matching in G.

Proof. With a matching N we can associate the function $I_N : V(G) \rightarrow \{0, 1\}$ defined as $I_N(v) = 1$ if v is N-saturated and $I_N(v) = 0$ otherwise. Clearly, we have $2|N| = \sum_{v \in V(G)} I_N(v)$.

Let M^* be a maximum matching in G. We will first obtain a matching $\bar{M}$ from M^* as follows. For every border cluster D in a big cluster C, delete all edges from M^* which have one endpoint in D and another in $V - C$. Extend the obtained matching to a maximal matching $\bar{M}$ with the property that all edges from $\bar{M}$ are contained in $G[C]$ for some big cluster C. We claim that for any big cluster C

$$\sum_{v \in C} I_{\bar{M}}(v) \geq \sum_{v \in C} I_{M^*}(v) - 5 s_C, \tag{4}$$

where s_C is the number of border clusters in C. Indeed, a subgraph of G induced by a border cluster D has a maximum independent set with at most 5 vertices (this is true for a closed neighborhood of any vertex in a unit-disk graph) and so the number of $\bar{M}$-free vertices in D which are possibly M^*-saturated is at most 5.

Since matching M_C from step (2) is a $(1 - \alpha)$-approximation of a maximum matching in $G[C]$, we have

$$\sum_{v \in C} I_M(v) \geq (1 - \alpha) \sum_{v \in C} I_{\bar{M}}(v)$$

and so

$$|M| \geq (1 - \alpha)|\bar{M}|.$$

Summing (4) over all C's yields

$$|\bar{M}| \geq |M^*| - 5s/2$$

where s is the total number of border clusters. We have $s = O(\epsilon|Aux(G)|)$ and $|M^*| = \Omega(|Aux(G)|)$ as $Aux(G)$ has a constant maximum degree. Therefore,

$$|M| \geq (1 - \alpha - O(\epsilon))|M^*|.$$

Theorem 2. *Let k be a positive integer, let $0 \leq \alpha < 1$, and let $T_\alpha(m)$ be the running time of a sequential $(1-\alpha)$-approximation algorithm for the maximum matching problem in a unit-disk graph of order m. There is a distributed algorithm which finds in a unit-disk graph G a matching M such that*

$$|M| \geq \left(1 - \alpha - O(1/\log^k |G|)\right)\beta$$

where β is the size of a maximum matching in G. The number of rounds of the algorithm is poly-logarithmic in $|G|$ and the sequential running time at each vertex of G is at most $T_\alpha(|G|)$.

Proof. Set $\epsilon = 1/\log^k |G|$ and apply Lemma 10 and Lemma 9.
Since it is possible to find a maximum matching in a polynomial time, we can obtain $\alpha = 0$ in Theorem 2 and have sequential running time which is polynomial in $|G|$.

3.2 Minimum Connected Dominating Set

Algorithm for the connected dominating set is surprisingly simple. It is the analysis that requires some additional work.

```
ApproxMinCDS
Input: A connected unit-disk graph $G = (V, E)$, $0 < \epsilon < 1$ and $0 \leq \alpha$.
Output: A dominating set in $G$.

(1) Call ClusteringUDG to find a clustering of $G$ with constants as in Corollary 1.
(2) In every big cluster $C$ find a connected dominating set $D_C$ which is a $(1+\alpha)$-
    approximation of a minimum connected dominating set in $G[C]$.
(3) For every two big clusters $C$, $C'$ if there is an edge in $Aux(G)$ connecting a
    border cluster from $C$ with a border cluster from $C'$ then connect a vertex
    from $D_C$ and a vertex from $D_{C'}$ by a path of length less than or equal to
    three. In other words, let $D_{C,C'}$ be the set of at most four vertices on the
    path connecting $C$ with $C'$.
(4) Return $D := \bigcup_C D_C \cup \bigcup_{\{C,C'\} \in E(Aux(G))} D_{C,C'}$.
```

It is easy to see that step three can be completed and that D is a dominating set which induces a connected subgraph of G. It is the fact that $|D|$ is close to the optimal that requires a proof. We will first observe that for any connected dominating set D there is a connected dominating set D' such that $D' \cap C$ induces a connected dominating set for any big cluster C and the sizes of D and D' do not differ much.

Lemma 11. *Let G be a connected unit-disk graph and let D be a connected dominating set in G. Then there is a dominating set D' with the following properties.*

1. *$D \subseteq D'$.*
2. *For any big cluster C, $C \cap D'$ induces a connected dominating set in $G[C]$.*
3. *$|D'| - |D| = O(s)$ where s is the number of border clusters.*

Proof. Let D be a connected dominating set. For every border cluster B there is a vertex v_B which dominates all of the vertices from B. Consider the set $D^* = D \cup \bigcup_B \{v_B\}$. For a big cluster C, let s_C be the number of border clusters contained in C. We first observe that the number of connected components in graph $G[D^* \cap C]$ is $O(s_C)$. Indeed, since $G[D^*]$ is connected, every connected component in $G[D^* \cap C]$ contains a vertex from a border cluster in C. Recall that for any border cluster B, $G[B]$ cannot have six independent vertices and so $G[D^* \cap B]$ has at most five connected components. Consequently, the number of connected components in $G[D^* \cap C]$ is $O(s_C)$. Next, consider graph $Conn(D^*, C)$ defined as follows. Vertices of $Conn(D^*, C)$ are connected components in $G[D^* \cap C]$ and we put an edge between w_1 and w_2 if there is a path of length at most three in G connecting a vertex from w_1 with a vertex from w_2. We claim that $Conn(D^*, C)$ is a connected graph. Indeed, assume that $Conn(D^*, C)$ is disconnected and let X_1, X_2 be two connected components in $Conn(D^*, C)$. Let Y_1 and Y_2 be the sets of vertices in $G[C]$ such that Y_i is the union of vertices in clusters from X_i. Let P be a shortest path in $G[C]$ connecting a vertex from Y_1 with a vertex form Y_2. The length of P is at least four. Let $p = u_1, u_2, u_3, u_4, \ldots, u_l$ with $u_1 \in Y_1$, $u_l \in Y_2$ and $l \geq 5$. Observe that u_3 cannot be contained in a border cluster as if $u_3 \in B$ then either $u_3 = v_B$ (the "center" of B) or u_3 is connected with v_B. In either case $v_B \in Y_1$ and there is a shorter path connecting Y_1 with Y_2. If $u_3 \notin B$ then u_3 must be dominated by a vertex v from $D \cap C$ and so there is a path of length at most 3 between u_1 and v. Thus $v \in Y_1$ and there is a shorter path connecting Y_1 with Y_2. Contradiction shows that $Conn(D^*, C)$ is a connected graph. As a result $Conn(D^*, C)$ contains a spanning tree in which an edge corresponds to path of length at most three. Fix one such spanning tree and let D' be the set of vertices in D^* in addition with vertices on each path corresponding to an edge in the spanning tree of $Conn(D^*, C)$. Clearly $D \subseteq D'$. In addition $G[D \cap C]$ is a connected graph. Finally, the number of vertices in $Conn(D^*, C)$ is $O(s_C)$ and so we add $O(s_C)$ vertices to D^* to obtain D'. Summing over all C's gives $|D'| - |D| = O(s)$.

Now the next lemma is very easy.

Lemma 12. *Let G be a connected unit-disk graph. The set D returned by* ApproxMinCDS *is a dominating set which induces a connected subgraph of G and such that*

$$|D| \leq (1 + \alpha + O(\epsilon))\gamma_c,$$

where γ_c is the size of a minimum dominating set in G.

Proof. Let D^* be a connected dominating set of size γ_c. By Lemma 11, there is a dominating set D' such that $D^* \subseteq D'$, $D' \cap C$ induces a connected dominating set in $G[C]$ for each C, and $|\bar{D}|-|D^*| = O(s)$. If D is the dominating set returned by algorithm ApproxMinCDS then $D_C = D \cap C$ is a $(1+\alpha)$-approximation of a minimum connected dominating set in $G[C]$ and so if D^*_C is a minimum connected dominating set in $G[C]$ then $|D_C| \leq (1+\alpha)|D^*_C|$. Since $D' \cap C$ is an optimal set in $G[C]$, we have $|D^*_C| \leq |D' \cap C|$ and so

$$|D| = \sum_C |D_C| + O(s) \leq \sum_C (1+\alpha)|D^*_C| + O(s) \leq \sum_C (1+\alpha)|D' \cap C| + O(s)$$

$$= (1+\alpha)|D'| + O(s) = (1+\alpha)|D^*| + O(s) = (1+\alpha)\gamma_c + O(\epsilon|Aux(G)|).$$

As before $\gamma_c = \Omega(|Aux(G)|)$ and so

$$|D| = (1+\alpha+O(\epsilon))\gamma_c.$$

Theorem 3. *Let k be a positive integer, let $0 \leq \alpha$, and let $T_\alpha(m)$ be the running time of a sequential $(1+\alpha)$-approximation for the minimum connected dominating set problem in a unit-disk graph of order m. There is a distributed algorithm which finds in a connected unit-disk graph G a dominating set D such that $G[D]$ is connected and*

$$|D| \leq \left(1+\alpha+O(1/\log^k |G|)\right)\gamma_c$$

where γ_c is the size of the minimum connected dominating set in G. The number of rounds of the algorithm is poly-logarithmic in $|G|$ and the sequential running time at each vertex of G is at most $T_\alpha(|G|)$.

4 Additional Remarks

As pointed out by one of the referees, in addition to Theorem 3, the methods described in the paper yield a sequential PTAS for the MCDS Problem in unit-disk graphs when the representation of the graph is unknown. Indeed, for a fixed $\epsilon > 0$ from (1), the radius of each cluster obtained by Cluster Set (and so the Clustering) is $O(1/\epsilon^2)$ by Lemma 3. Since in a unit-disk graph of diameter r any indepdenet set has size of at most $O(r^2)$ and each independent set is also a dominating set in the graph, the size of the minimum dominating set is $O(1/\epsilon^4)$. As a result, the size of the minimum connected dominating set in each cluster is also $O(1/\epsilon^4)$. Consequently, to find an optimal solution in each cluster it is enough to check $O(n^{1/\epsilon^4})$ subsets in a cluster and select an optimal. This extends ther result of Nieberg and Hurink from [NH05] where a PTAS for the MDS Problem is given.

References

[AGLP89] B. Awerbuch, A. V. Goldberg, M. Luby, S. A. Plotkin, Network Decomposition and Locality in Distributed Computation, Proc. 30th IEEE Symp. on Foundations of Computer Science , (1989), 364–369.

[CHLWD03] X. Cheng, X. Huang, D. Li, W. Wu, and D.-Z. Du, Polynomial-time approximation scheme for minimum connected dominating set in ad hoc wireless networks, Volume 42, Issue 4, Networks, (2003), 202–208.

[CH06] A. Czygrinow, M. Hańćkowiak, Distributed algorithms for weighted problems in sparse graphs, Journal of Discrete Algorithms, in press, (2006).

[CHS06] A. Czygrinow, M. Hańćkowiak, E. Szymańska, Distributed approximation algorithms for planar graphs, 6th Conference on Algorithms and Complexity, CIAC 2006, accepted, (2006).

[D97] R. Diestel, *Graph Theory*, Springer, New York, (1997).

[DW04] F. Dai, J. Wu, An Extended Localized Algorithm for Connected Dominating Set Formation in Ad Hoc Wireless Networks, IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 10, (2004), 908–920.

[DPRS03] D. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan, Fast Distributed Algorithms for (Weakly) Connected Dominating Sets and Linear-Size Skeletons, In Proc. of the ACM-SIAM Symposium on Discrete Algorithms (SODA), (2003), 717–724.

[GGHZZ01] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu, Discrete mobile centers, In Proc of the 17th annual Symposium on Computational Geometry (SCG), (2001), 188–196.

[KMW05] F. Kuhn, T. Moscibroda, and R. Wattenhofer, On the Locality of Bounded Growth, 24th ACM Symposium on the Principles of Distributed Computing (PODC), Las Vegas, Nevada, USA, (2005), 60–68.

[KMNW05a] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer, Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs, 19th International Symposium on Distributed Computing (DISC), Cracow, Poland, September (2005), 273–287.

[KMNW05b] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer, Local Approximation Schemes for Ad Hoc and Sensor Networks, 3rd ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC), Cologne, Germany, (2005), 97–103.

[L82] D. Lichtenstein, Planar Formulae and Their Uses, SIAM Journal of Computing, 11(2), (1998), 329–343.

[L92] N. Linial, Locality in distributed graph algorithms, SIAM Journal on Computing, 21(1), (1992), 193–201.

[NH05] T. Nieberg, J. L. Hurink, A PTAS for the Minimum Dominating Set Problem in Unit Disk Graphs, 3rd Workshop on Approximation and Online Algorithms, WAOA 2005, Mallorca, Spain, October 2005, Springer LNCS 3879, (2005), 296–306.

[PR01] A. Panconesi, R. Rizzi, Some Simple Distributed Algorithms for Sparse Networks, Distributed Computing 14, (2001), 97–100.

[P00] D. Peleg, *Distributed Computing: A Locality-Sensitive Approach*, SIAM, 2000.

The Weakest Failure Detectors to Boost Obstruction-Freedom

Rachid Guerraoui[1,2], Michał Kapałka[2], and Petr Kouznetsov[3]

[1] Computer Science and Artificial Intelligence Laboratory, MIT
[2] School of Computer and Communication Sciences, EPFL
[3] Max Planck Institute for Software Systems

Abstract. This paper determines necessary and sufficient conditions to implement *wait-free* and *non-blocking* contention managers in a shared memory system. The necessary conditions hold even when universal objects (like compare-and-swap) or random oracles are available, whereas the sufficient ones assume only registers.

We show that failure detector $\Diamond\mathcal{P}$ is the weakest to convert any obstruction-free algorithm into a wait-free one, and Ω^*, a new failure detector which we introduce in this paper, and which is strictly weaker than $\Diamond\mathcal{P}$ but strictly stronger than Ω, is the weakest to convert any obstruction-free algorithm into a non-blocking one.

1 Introduction

Multiprocessor systems are becoming more and more common nowadays. Multithreading thus becomes the norm and studying scalable and efficient synchronization methods is essential, for traditional locking-based techniques do not scale and may induce priority inversion, deadlock and fault-tolerance issues when a large number of threads is involved.

Wait-free synchronization algorithms [1] circumvent the issues of locking and guarantee individual progress even in presence of high contention. Wait-freedom is a liveness property which stipulates that every process completes every operation in a finite number of its own steps, regardless of the status of other processes, i.e., contending or even crashed. Ideal synchronization algorithms would ensure *linearizability* [2,3], a safety property which provides the illusion of instantaneous operation executions, together with wait-freedom.

Alternatively, a liveness property called *non-blockingness*[1] may be considered instead of wait-freedom. Non-blockingness guarantees global progress, i.e., that some process will complete an operation in a finite number of steps, regardless of

[1] The term *non-blocking* is defined here in the traditional way [1]: "some process will complete its operation in a finite number of steps, regardless of the relative execution speeds of the processes." This term is sometimes confused with the term *lock-free*. Note that non-blocking implementations provide a weaker liveness guarantee than wait-free implementations.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 399–412, 2006.

the behavior of other processes. Non-blockingness is weaker than wait-freedom as it does not prevent some processes from starvation.

Wait-free and non-blocking algorithms are, however, notoriously difficult to design [4,5], especially with the practical goal to be fast in low contention scenarios, which are usually considered the most common in practice. An appealing principle to reduce this difficulty consists in separating two concerns of a synchronization algorithm: (1) ensuring linearizability with a minimal conditional progress guarantee, and (2) boosting progress. More specifically, the idea is to focus on algorithms that ensure linearizability together with a weak liveness property called *obstruction-freedom* [6], and then combine these algorithms with separate generic oracles that boost progress, called *contention managers* [7,8,9,10]. This separation lies at the heart of modern (obstruction-free) software transactional memory (STM) frameworks [7].

With obstruction-free (or OF, for short) algorithms, progress is ensured only for every process that executes in isolation for sufficiently long time. In presence of high contention, however, OF algorithms can livelock, preventing any process from terminating. Contention managers are used precisely to cope with such scenarios. When queried by a process executing an OF algorithm, a contention manager can delay the process for some time in order to boost the progress of other processes. The contention manager can neither share objects with the OF algorithm, nor return results on its behalf. If it did, the contention manager could peril the safety of the OF algorithm, hampering the overall separation of concerns principle.

In short, the goal of a contention manager is to provide processes with enough time without contention so that they can complete their operations. In its simplest form, a contention manager can be a randomized back-off protocol. More sophisticated contention management strategies have been experimented in practice [8,9,11]. Precisely because they are entirely devoted to progress, they can be combined or changed on the fly [10]. Most previous strategies were *pragmatic*, with no aim to provide *worst case guarantees*. In this paper we focus on contention managers that provide such guarantees. More specifically, we study contention managers that convert any OF algorithm into a non-blocking or wait-free one, and which we call, respectively, *non-blocking* or *wait-free* contention managers.

Two wait-free contention managers have recently been proposed [12,13]. Both rely on timing assumptions to detect processes that fail in the middle of their operations. This suggests that *some* information about failures might inherently be needed by any wait-free contention manager. But this is not entirely clear because, in principle, a contention manager could also use randomization to schedule processes, or even powerful synchronization primitives like compare-and-swap, which is known to be *universal*, i.e., able to wait-free implement any other object [1]. In the parlance of [14], we would like to determine whether a *failure detector* is actually needed to implement a contention manager with worst case guarantees, and if it is, what is the *weakest* one [15]. Besides the theoretical interest, determining the minimal conditions under which a contention manager can ensure certain guarantees is, we believe, of practical relevance, for this might help portability and optimization.

We show that the eventually perfect failure detector $\Diamond\mathcal{P}$ [14] is the weakest to implement a wait-free contention manager.[2] We also introduce a failure detector Ω^*, which we show is the weakest to implement a non-blocking contention manager. Failure detector Ω^* is strictly weaker than $\Diamond\mathcal{P}$, and strictly stronger than failure detector Ω [15], known to be the weakest to wait-free implement the (universal) consensus object [1].[3]

It might be surprising that Ω is not sufficient to implement a wait-free or even a non-blocking contention manager. For example, the seminal Paxos algorithm [16] uses Ω to transform an OF implementation of consensus into a wait-free one. Each process that is eventually elected a leader by Ω is given enough time to run alone, reach a decision and communicate it to the others. This approach does not help, however, if we want to make sure that processes make progress regardless of the actual (possibly long-lived) object and its OF implementation. Intuitively, the leader elected by Ω may have no operation to perform while other processes may livelock forever. Because a contention manager cannot make processes help each other, the output of Ω is not sufficient: this is so even if randomized oracles or universal objects are available. Intuitively, wait-free contention managers need a failure detector that would take care of *every* non-crashed process with a pending operation so that the process can run alone for sufficiently long time. As for non-blocking contention managers, at least *one* process that never crashes, among the ones with pending operations, should be given enough time to run alone.

The paper is organized as follows. Section 2 presents our system model and formally defines wait-free and non-blocking contention managers. These definitions are, we believe, contributions in their own rights, for they capture precisely the interaction between a contention manager and an obstruction-free algorithm. In Sect. 3 and 4, we prove our weakest failure detector results. In each case, we first present (necessary part) a *reduction* algorithm [15] that *extracts* the output of failure detector Ω^* (respectively $\Diamond\mathcal{P}$) using a non-blocking (respectively wait-free) contention manager implementation. When devising our reduction algorithms, we do not restrict what objects (or random oracles) can be used by the contention manager or the OF algorithm. Then (sufficient part), we present algorithms that implement the contention managers using the failure detectors and registers. These algorithms are devised with the sole purpose of proving our sufficiency claims. We do not seek to minimize the overhead of the interaction between the OF algorithm and the contention manager, nor do we discuss how the failure detector can itself be implemented with little synchrony assumptions and minimal overhead, unlike the transformations presented in [12]. However, as we show in [17], our algorithms can be easily extended to meet these challenges.

[2] $\Diamond\mathcal{P}$ ensures that eventually: (1) every failure is detected by every correct (i.e., non-faulty) process and (2) there is no false detection.

[3] Ω ensures that eventually all correct (i.e., non-faulty) processes elect the same correct process as their leader.

The proofs of a few minor results are omitted due to the space limitations and can be found in the full version of the paper [18].

2 Preliminaries

Processes and Failure Detectors. We consider a set of n processes $\Pi = \{p_1, \ldots, p_n\}$ in a shared memory system [1,19]. A process executes the (possibly randomized) algorithm assigned to it, until the process *crashes* (*fails*) and stops executing any action. We assume the existence of a global discrete clock that is, however, inaccessible to the processes. We say that a process is *correct* if it never crashes. We say that process p_i is *alive* at time t if p_i has not crashed by time t.

A *failure detector* [14,15] is a distributed oracle that provides every process with some information about failures. The output of a failure detector depends only on which and when processes fail, and not on computations being performed by the processes. A process p_i queries a failure detector $\mathcal{D}$ by accessing local variable $\mathcal{D}\textit{-output}_i$—the output of the module of $\mathcal{D}$ at process p_i. Failure detectors can be partially ordered according to the amount of information about failures they provide. A failure detector $\mathcal{D}$ is *weaker than a failure detector* $\mathcal{D}'$, and we write $\mathcal{D} \preceq \mathcal{D}'$, if there exists an algorithm (called a *reduction* algorithm) that transforms $\mathcal{D}'$ into $\mathcal{D}$. If $\mathcal{D} \preceq \mathcal{D}'$ but $\mathcal{D}' \not\preceq \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$, and we write $\mathcal{D} \prec \mathcal{D}'$.

Base and High-Level Objects. Processes communicate by invoking primitive operations (which we will call *instructions*) on *base* shared objects and seek to implement the *operations* of a *high-level* shared object O. Object O is in turn used by an application, as a high-level inter-process communication mechanism. We call invocation and response events of a high-level operation *op* on the implemented object O *application events* and denote them by, respectively, *inv*(*op*) and *ret*(*op*) (or $\textit{inv}_i(\textit{op})$ and $\textit{ret}_i(\textit{op})$ at a process p_i).

An *implementation* of O is a distributed algorithm that specifies, for every process p_i and every operation *op* of O, the sequences of *steps* that p_i should take in order to complete *op*. Process p_i *completes* operation *op* when p_i returns from *op*. Every process p_i may complete any number of operations but, at any point in time, at most one operation *op* can be *pending* (started and not yet completed) at p_i.

We consider implementations of O that combine a sub-protocol that ensures a minimal liveness property, called *obstruction-freedom*, with a sub-protocol that boosts this liveness guarantee. The former is called an *obstruction-free (OF)* algorithm A and the latter a *contention manager* CM. We focus on *linearizable* [2,3] implementations of O: every operation appears to the application as if it took effect instantaneously between its invocation and its return. An implementation of O involves two categories of steps executed by any process p_i: those (executed on behalf) of *CM* and those (executed on behalf) of A. In each step, a process p_i either executes an instruction on a base shared object or (in case p_i executes a step on behalf of *CM*) queries a failure detector.

Obstruction-freedom [6,7] stipulates that if a process that invokes an operation *op* on object O and from some point in time executes steps of A alone[4], then it eventually completes *op*. *Non-blockingness* stipulates that if some correct process never completes an invoked operation, then some other process completes infinitely many operations. *Wait-freedom* [1] ensures that every correct process that invokes an operation eventually returns from the operation.

Interaction Between Modules. OF algorithm A, executed by any process p_i, communicates with contention manager *CM* via *calls* try_i and resign_i implemented by *CM* (see Fig. 1). Process p_i invokes try_i just after p_i starts an operation, and also later (even several times before p_i completes the operation) to signal possible contention. Process p_i invokes resign_i just before returning from an operation, and always eventually returns from this call (or crashes). Both calls, try_i and resign_i, return *ok*.[5]

We denote by $B(A)$ and $B(\mathit{CM})$ the sets of base shared objects, always *disjoint*, that can be possibly accessed by steps of, respectively, A and *CM*, in every execution, by every process. Calls *try* and *resign* are thus the only means by which A and *CM* interact. The events corresponding to invocations of, and responses from, *try* and *resign* are called *cm-events*. We denote by $\mathit{try}_i^{\mathrm{inv}}$ and $\mathit{resign}_i^{\mathrm{inv}}$ an invocation of call try_i and resign_i, respectively (at process p_i), and by $\mathit{try}_i^{\mathrm{ret}}$ and $\mathit{resign}_i^{\mathrm{ret}}$—the corresponding responses.

Executions and Histories. An *execution* of an OF algorithm A combined with a contention manager *CM* is a sequence of *events* that include steps of A, steps of *CM*, cm-events and application events. Every event in an execution is associated with a unique time at which the event took place. Every execution e induces a *history* $H(e)$ that includes only application events (invocations and responses of high-level operations). The corresponding *CM-history* $H_{\mathrm{CM}}(e)$ is the subsequence of e containing only application events and cm-events of the execution, and the corresponding *OF-history* $H_{\mathrm{OF}}(e)$ is the subsequence of e containing only application events, cm-events, and steps of A. For a sequence s of events, $s|i$ denotes the subsequence of s containing only events at process p_i.

We say that a process p_i is *blocked* at time t in an execution e if (1) p_i is alive at time t, and (2) the latest event in $H_{\mathrm{CM}}(e)|i$ that occurred before t is $\mathit{try}_i^{\mathrm{inv}}$ or $\mathit{resign}_i^{\mathrm{inv}}$. A process p_i is *busy* at time t in e if (1) p_i is alive at time t, and (2) the latest event in $H_{\mathrm{CM}}(e)|i$ that occurred before t is $\mathit{try}_i^{\mathrm{ret}}$. We say that a process p_i is *active* at t in e if p_i is either busy or blocked at time t in e. We say that a process p_i is *idle* at time t in e if p_i is not active at t in e.[6] A process *resigns* when it invokes *resign* on a contention manager.

[4] i.e., without encountering *step contention* [20].

[5] An example OF algorithm that uses this model of interaction with a contention manager is presented in [18]. A discussion about overhead of wait-free/non-blocking contention managers that explains when calls to *try*/*resign* can be omitted for efficiency reasons can be found in [17].

[6] Note that every process that has crashed is permanently idle.

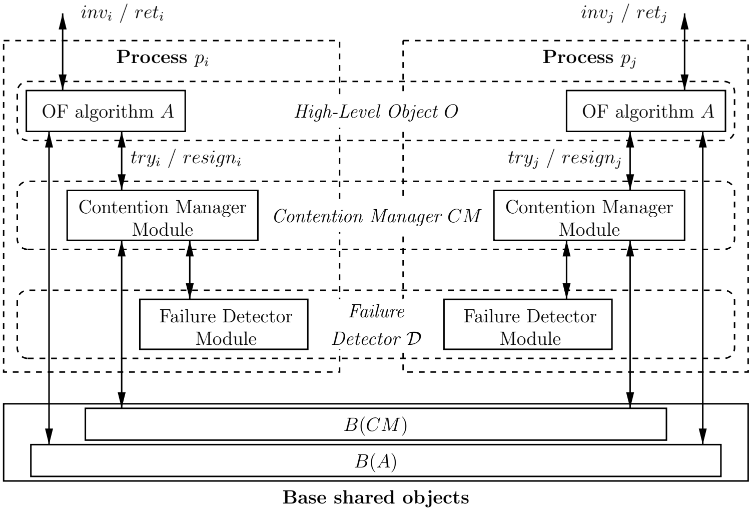


Fig. 1. The OF algorithm/contention manager interface

We say that p_i is *obstruction-free* in an interval $[t, t']$ in an execution e, if p_i is the only process that takes steps of A in $[t, t']$ in e and p_i is not blocked infinitely long in $[t, t']$ (if $t' = \infty$). We say that process p_i is *eventually obstruction-free* at time t in e if p_i is active at t or later and p_i either resigns after t or is obstruction-free in the interval $[t', \infty)$ for some $t' > t$. Note that, since algorithm A is obstruction-free, if an active process p_i is eventually obstruction-free, then p_i eventually resigns and completes its operation.

Well-Formed Executions. We impose certain restrictions on the way an OF algorithm A and a contention manager *CM* interact. In particular, we assume that no process takes steps of A while being blocked by *CM* or idle, and no process takes infinitely many steps of A without calling *CM* infinitely many times. Further, a process must inform *CM* that an operation is completed by calling *resign* before returning the response to the application.

Formally, we assume that every execution e is *well-formed*, i.e., $H(e)$ is linearizable [2,3], and, for every process p_i, (1) $H_{\mathrm{CM}}(e)|i$ is a prefix of a sequence $[op_1][op_2], \ldots$, where each $[op_k]$ has the form $inv_i(op_k), try_i^{\mathrm{inv}}$, try_i^{ret}, $\ldots$, try_i^{inv}, try_i^{ret}, $resign_i^{\mathrm{inv}}$, $resign_i^{\mathrm{ret}}, ret_i(op_k)$; (2) in $H_{\mathrm{OF}}(e)|i$, no step of A is executed when p_i is blocked or idle, (3) in $H_{\mathrm{OF}}(e)|i$, inv_i can only be followed by try_i^{inv}, and ret_i can only be preceded by $resign_i^{\mathrm{ret}}$; (4) if p_i is busy at time t in e, then at some $t' > t$, process p_i is idle or blocked. The last condition implies that every busy process p_i eventually invokes try_i (and becomes blocked), resigns or crashes. Clearly, in a well-formed execution, every process goes through the following cyclical order of modes: *idle*, *active*, *idle*, ..., where each *active* period consists itself of a sequence *blocked*, *busy*, *blocked*, ...

Non-blocking Contention Manager. We say that a contention manager *CM* *guarantees non-blockingness for an OF algorithm A* if in each execution e of A combined with *CM* the following property is satisfied: if some correct process is active at a time t, then at some time $t' > t$ some process resigns.

A *non-blocking contention manager* guarantees non-blockingness for every OF algorithm. Intuitively, this will happen if the contention manager allows at least one active process to be obstruction-free (and busy) for sufficiently long time, so that the process can complete its operation. More precisely, we say that a contention manager *CM* is *non-blocking* if, for every OF algorithm A, in every execution of A combined with *CM* the following property is ensured at every time t:

Global Progress. If some correct process is active at t, then some correct process is eventually obstruction-free at t.

We show in [18] that a contention manager *CM* guarantees non-blockingness for every OF algorithm if and only if *CM* is non-blocking.

Wait-Free Contention Manager. We say that a contention manager *CM* *guarantees wait-freedom for an OF algorithm A* if in every execution e of A combined with *CM*, the following property is satisfied: if a process p_i is active at a time t, then at some time $t' > t$, p_i becomes idle. In other words, every operation executed by a correct process eventually returns.

A *wait-free contention* manager guarantees wait-freedom for every OF algorithm. Intuitively, this will happen if the contention manager makes sure that every correct active process is given "enough" time to complete its operation, regardless of how other processes behave. More precisely, a contention manager *CM* is wait-free if, for every OF algorithm A, in every execution of A combined with *CM*, the following property is ensured at every time t:[7]

Fairness. If a correct process p_i is active at t, then p_i is eventually obstruction-free at t.

We show in [18] that a contention manager *CM* guarantees wait-freedom for every OF algorithm if and only if *CM* is wait-free.

In the following, we seek to determine the *weakest* [15] failure detector $\mathcal{D}$ to implement a non-blocking (resp. wait-free) contention manager *CM*. This means that (1) $\mathcal{D}$ implements such a contention manager, i.e., there is an algorithm that implements *CM* using $\mathcal{D}$, and (2) $\mathcal{D}$ is *necessary* to implement such a contention manager, i.e., if a failure detector $\mathcal{D}'$ implements *CM*, then $\mathcal{D} \preceq \mathcal{D}'$. In our context, a reduction algorithm that transforms $\mathcal{D}'$ into $\mathcal{D}$ uses the $\mathcal{D}'$-based implementation of the corresponding contention manager as a "black box" and read-write registers.

[7] This property is ensured by wait-free contention managers from the literature [12,13].

3 Non-blocking Contention Managers

Let $S \subseteq \Pi$ be a non-empty set of processes. Failure detector Ω_S outputs, at every process, an identifier of a process (called a *leader*), such that all correct processes *in* S eventually agree on the identifier of the same *correct* process *in* S.[8]

Failure detector Ω^* is the composition $\{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$: at every process p_i, Ω^*-*output*$_i$ is a tuple consisting of the outputs of failure detectors Ω_S. We position Ω^* in the hierarchy of failure detectors of [14] by showing in [18] that $\Omega \prec \Omega^* \prec \Diamond\mathcal{P}$.

To show that Ω^* is necessary to implement a non-blocking contention manager, it suffices to prove that, for every non-empty $S \subseteq \Pi$, Ω_S is necessary to implement a non-blocking contention manager. Let *CM* be a non-blocking contention manager using failure detector $\mathcal{D}$. We show that $\Omega^* \preceq \mathcal{D}$ by presenting an algorithm $T_{\mathcal{D} \to \Omega}$ (Algorithm 1) that, using *CM* and $\mathcal{D}$, emulates the output of Ω_S.

Algorithm 1: Extracting Ω_S from a non-blocking contention manager (code for processes from set S; others are permanently idle)

```
     uses: $L$—register
     initially: $\Omega_S$-output$_i \leftarrow p_i$, $L \leftarrow$ some process in $S$
       Launch two parallel tasks: $T_i$ and $F_i$
1.1  parallel task $F_i$
1.2      $\Omega_S$-output$_i \leftarrow L$
1.3  parallel task $T_i$
1.4      while true do
1.5          issue try$_i$ and wait until busy (i.e., until call try$_i$ returns)
1.6          $L \leftarrow p_i$ // announce yourself a leader
```

The algorithm works as follows. Every process $p_i \in S$ runs two parallel tasks T_i and F_i. In task T_i, process p_i periodically (1) gets blocked by *CM* after invoking *try*$_i$ (line 1.5), and (2) once p_i gets busy again, announces itself a leader for set S by writing its id in L (line 1.6). In task F_i, process p_i periodically determines its leader by reading register L (line 1.2).[9]

Thus, no process ever resigns and every correct process in S is permanently active from some point in time. Intuitively, this signals a possible livelock to *CM* which has to eventually block all active processes except for one that should run obstruction-free for sufficiently long time. By Global Progress, *CM* cannot block *all* active processes forever and so if the elected process crashes (and so

[8] Ω_S can be seen as a restriction of the eventual leader election failure detector Ω [15] to processes in S. The definition of Ω_S resembles the notion of Γ-accurate failure detectors introduced in [21]. Clearly, Ω_Π is Ω.

[9] If a process is blocked in one task, it continues executing steps in parallel tasks.

becomes idle), *CM* lets another active process run obstruction-free. Eventually, all correct processes in S agree on the same process in S. Processes outside S are permanently idle and permanently output their own ids: they do not access *CM*.

This approach contains a subtlety. To make sure that there is a time after which the same correct leader in S is permanently elected by the correct processes in S, we do not allow the elected leader to resign (the output of Ω_S has to be eventually stable). This violates the assumption that processes using *CM* run an obstruction-free algorithm, and, thus, a priori, *CM* is not obliged to preserve Global Progress. However, as we show below, since *CM* does not "know" how much time a process executing an OF algorithm requires to complete its operation, *CM* has to provide some correct process with *unbounded* time to run in isolation.

Theorem 1. *Every non-blocking contention manager can be used to implement failure detector* Ω^*.

Proof. Let $S \subseteq \Pi$, $S \neq \emptyset$ and consider any execution of Algorithm 1. If S contains no correct process, then Ω_S-$output_i$ (for every process $p_i \in S$) trivially satisfies the property of Ω_S. Now assume that there is a correct process in S. We claim that *CM* eventually lets exactly one correct process in S run obstruction-free while blocking forever all the other processes in S.

Suppose not. We obtain an execution in which every correct process in S is allowed to be obstruction-free only for bounded periods of time. But the CM-history of this execution corresponds to an execution of some OF algorithm A combined with *CM* in which no active process ever completes its operation because no active process ever obtains enough time to run in isolation. Thus, no active process is eventually obstruction-free in that execution. This contradicts the assumption that *CM* is non-blocking.

Therefore, there is a time after which exactly one correct process $p_j \in S$ is periodically busy (others are blocked or idle forever) and, respectively, register L permanently stores the identifier of p_j. Thus, eventually, every correct process in S outputs p_j: the output of Ω_S is extracted. □

We describe an implementation of a non-blocking contention manager using Ω^* and registers in Algorithm 2 (we prove its correctness in [18]). The algorithm works as follows. All active processes, upon calling *try*, participate in the leader election mechanism using Ω^* in lines 2.3–2.5. The active process p_i that is elected a leader returns from *try* and is (eventually) allowed to run obstruction-free until p_i resigns. Once p_i resigns, the processes elect another leader. Failure detector Ω^* guarantees that if an active process is elected and crashes before resigning, another active process is eventually elected.

Theorem 2. *Algorithm 2 implements a non-blocking contention manager.*

Algorithm 2: A non-blocking contention manager using $\Omega^* = \{\Omega_S\}_{S \subseteq \Pi, S \neq \emptyset}$

```
uses: $T[1,\ldots,n]$—array of single-bit registers
initially: $T[1,\ldots,n] \leftarrow false$

upon $try_i$ do
    $T[i] \leftarrow true$
    repeat
        $S \leftarrow \{p_j \in \Pi \mid T[j] = true\}$
    until $\Omega_S$-$output_i = p_i$

upon $resign_i$ do
    $T[i] \leftarrow false$
```

4 Wait-Free Contention Managers

We prove here that the weakest failure detector to implement a wait-free contention manager is $\Diamond\mathcal{P}$. Failure detector $\Diamond\mathcal{P}$ [14] outputs, at each time and every process, a set of *suspected* processes. There is a time after which (1) every crashed process is permanently suspected by every correct process and (2) no correct process is ever suspected by any correct process.

We first consider a wait-free contention manager *CM* using a failure detector $\mathcal{D}$, and we exhibit a reduction algorithm $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$ (Algorithm 3) that, using *CM* and $\mathcal{D}$, emulates the output of $\Diamond\mathcal{P}$.

Algorithm 3: Extracting $\Diamond\mathcal{P}$ from a wait-free contention manager

```
uses: $R[1,\ldots,n]$—array of registers
initially: $\Diamond\mathcal{P}$-$output_i \leftarrow \Pi - \{p_i\}$, $k \leftarrow 0$, $R[i] \leftarrow 0$
    Launch $n(n-1)$ parallel instances of $CM$: $C_{jk}$, $j,k \in \{1,\ldots,n\}$, $j \neq k$
    Launch $2n-1$ parallel tasks: $T_{ij}$, $T_{ji}$, $j \in \{1,\ldots,n\}$, $i \neq j$, and $F_i$

parallel task $F_i$
    while $true$ do  $R[i] \leftarrow R[i]+1$   // ``heartbeat'' signal

parallel task $T_{ij}$, $j = 1,\ldots,i-1,i+1,\ldots,n$
    while $true$ do
        $x_j \leftarrow R[j]$
        $\Diamond\mathcal{P}$-$output_i \leftarrow \Diamond\mathcal{P}$-$output_i - \{p_j\}$  // stop suspecting $p_j$
        issue $try_i^{ij}$ (in $C_{ij}$) and wait until busy
        issue $resign_i^{ij}$ (in $C_{ij}$) and wait until idle
        $\Diamond\mathcal{P}$-$output_i \leftarrow \Diamond\mathcal{P}$-$output_i \cup \{p_j\}$  // start suspecting $p_j$
        wait until $R[j] > x_j$  // wait until $p_j$ takes a new step

parallel task $T_{ji}$, $j = 1,\ldots,i-1,i+1,\ldots,n$
    while $true$ do  issue $try_i^{ji}$ (in $C_{ji}$) and wait until busy
```

We run several instances of *CM*. These instances use disjoint sets of base shared objects and do not directly interact. Basically, in each instance, only two processes are active and all other processes are idle. One of the two processes, say p_j, gets active and never resigns thereafter, while the other, say p_i, permanently alternates between being active and idle. To *CM* it looks like p_j is always obstructed by p_i. Thus, to guarantee wait-freedom, the instance of *CM* has to eventually block p_i and let p_j run obstruction-free until p_j resigns *or crashes.* Therefore, when p_i is blocked, p_i can assume that p_j is alive and when p_i is busy, p_i can suspect p_j of having crashed, until p_i eventually observes p_j's "heartbeat" signal, which p_j periodically broadcasts using a register. This ensures the properties of $\Diamond\mathcal{P}$ at process p_i, provided that p_j never resigns.

As in Sect. 3, we face the following issue. If p_j is correct, p_i will be eventually blocked forever and p_j will thus be eventually obstruction-free. Hence, in the corresponding execution, obstruction-freedom is violated, i.e., the execution cannot be produced by any OF algorithm combined with *CM*. One might argue then that *CM* is not obliged to preserve Fairness with respect to p_j. However, we show that, since *CM* does not "know" how much time a process executing an OF algorithm requires to complete its operation, *CM* has to provide p_j with *unbounded* time to run in isolation.

More precisely, the processes in Algorithm 3 run $n(n-1)$ parallel instances of *CM*, denoted each CM_{jk}, where $j, k \in \{1, \ldots, n\}$, $j \neq k$. We denote the events that process p_i issues in instance CM_{jk} by try_i^{jk} and $resign_i^{jk}$. Besides, every process p_i runs $2n-1$ parallel tasks: T_{ij}, T_{ji}, where $j \in \{1, \ldots, n\}$, $i \neq j$, and F_i. Every task T_{ij} executed by p_i is responsible for detecting failures of process p_j. Every task T_{ji} executed by p_i is responsible for preventing p_j from falsely suspecting p_i. In task F_i, p_i periodically writes ever-increasing "heartbeat" values in a shared register $R[i]$.

In every instance CM_{ij}, there can be only two active processes: p_i and p_j. Process p_i cyclically gets active (line 3.7) and resigns (line 3.8), and process p_j gets active once and keeps getting blocked (line 3.12). Each time before p_i gets active, p_i removes p_j from the list of suspected processes (line 3.6). Each time p_i stops being blocked, p_i starts suspecting p_j (line 3.9) and waits until p_i observes a "new" step of p_j (line 3.10). Once such a step of p_j is observed, p_i stops suspecting p_j and gets active again.

Theorem 3. *Every wait-free contention manager can be used to implement failure detector $\Diamond\mathcal{P}$.*

Proof. Consider any execution e of $T_{\mathcal{D}\rightarrow\Diamond\mathcal{P}}$, and let p_i be any correct process. We show that, in e, $\Diamond\mathcal{P}$-*output*$_i$ satisfies the properties of $\Diamond\mathcal{P}$, i.e., p_i eventually permanently suspects every non-correct process and stops suspecting every correct process. (Note that if a process p_i is not correct, then $\Diamond\mathcal{P}$-*output*$_i$ trivially satisfies the properties of $\Diamond\mathcal{P}$.)

Algorithm 4: A wait-free contention manager using $\Diamond\mathcal{P}$

```
    uses: $T[1,\ldots,N]$—array of registers (other variables are local)
    initially: $T[1,\ldots,N] \leftarrow \bot$
4.1 upon $try_i$ do
4.2     if $T[i] = \bot$ then $T[i] \leftarrow$ GetTimestamp()
4.3     repeat
4.4         $sact_i \leftarrow \{\, j \mid T[j] \neq \bot \wedge p_j \notin \Diamond\mathcal{P}\text{-}output_i \,\}$
4.5         $leader_i \leftarrow \mathrm{argmin}_{j \in sact}\, T[j]$
4.6     until $leader_i = i$
4.7 upon $resign_i$ do
4.8     $T[i] \leftarrow \bot$
```

Let p_j be any process distinct from p_i. Assume p_j is not correct. Thus p_i is the only correct active process in instance CM_{ij}. By the Fairness property of CM, p_i is eventually obstruction-free every time p_i becomes active, and so p_i cannot be blocked infinitely long in line 3.7. Since there is a time after which p_j stops taking steps, eventually p_i starts suspecting p_j (line 3.9) and suspends in line 3.10, waiting until p_j takes a new step. Thus, p_i eventually suspects p_j forever.

Assume now that p_j is correct. We claim that p_i must eventually get permanently blocked so that p_j would run obstruction-free from some point in time forever. Suppose not. But then we obtain an execution in which p_i alternates between active and idle modes infinitely many times, and p_j stays active and runs obstruction-free only for bounded periods of time. But the CM-history of this execution could be produced by an execution e' of some OF algorithm combined with CM in which p_j never completes its operation because p_j never runs long enough in isolation. Thus, Fairness is violated in execution e' and this contradicts the assumption that CM is wait-free. Hence, eventually p_i gets permanently blocked in line 3.7. Since each time p_i is about to get blocked, p_i stops suspecting p_j in line 3.6, there is a time after which p_i never suspects p_j.

Thus, there is a time after which, if p_j is correct, then p_j stops being suspected by every correct process, and if p_j is non-correct, then every correct process permanently suspects p_j. □

We describe an implementation of a wait-free contention manager using $\Diamond\mathcal{P}$ and registers in Algorithm 4 (we prove its correctness in [18]). The algorithm relies on a (wait-free) primitive *GetTimestamp()* that generates unique, locally increasing timestamps and makes sure that if a process gets a timestamp ts, then no process can get timestamps lower than ts infinitely many times (this primitive can be implemented in an asynchronous system using read-write registers). The idea of the algorithm is the following. Every process p_i that gets active receives a timestamp in line 4.2 and announces the timestamp in register $T[i]$. Every active process that invokes *try* repeatedly runs a leader election mechanism (lines 4.3–4.6): the non-suspected (by $\Diamond\mathcal{P}$) process that announced the lowest (non-$\bot$)

timestamp is elected a leader. If a process p_i is elected, p_i returns from try_i and becomes busy. $\Diamond\mathcal{P}$ guarantees that eventually the same correct active process is elected by all active processes. All other active processes stay blocked until the process resigns and resets its timestamp in line 4.8. The leader executes steps obstruction-free then. Since the leader runs an OF algorithm, the leader eventually resigns and resets its timestamp in line 4.8 so that another active process, which now has the lowest timestamp in T, can become a leader.

Theorem 4. *Algorithm 4 implements a wait-free contention manager.*

Acknowledgements. We are very grateful to Hagit Attiya, Maurice Herlihy, Bastian Pochon, Faith Fich, Victor Luchangco, Mark Moir and Nir Shavit for interesting discussions on the topic of this paper. We would also like to thank the anonymous reviewers for helpful comments.

References

1. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (1991) 124–149
2. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (1990) 463–492
3. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition). Wiley (2004)
4. LaMarca, A.: A performance evaluation of lock-free synchronization protocols. In: Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94). (1994) 130–140
5. Bershad, B.N.: Practical considerations for non-blocking concurrent objects. In: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems (ICDCS'93). (1993) 264–273
6. Herlihy, M., Luchango, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'93). (2003) 522–529
7. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03). (2003) 92–101
8. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: PODC Workshop on Concurrency and Synchronization in Java Programs. (2004)
9. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05). (2005)
10. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05), LNCS, Springer (2005) 303–323
11. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05). (2005)

12. Fich, F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05). (2005)
13. Guerraoui, R., Herlihy, M., Kapałka, M., Pochon, B.: Robust contention management in software transactional memory. In: Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL); in conjunction with the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05). (2005)
14. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43(2)** (1996) 225–267
15. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43(4)** (1996) 685–722
16. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2) (1998) 133–169
17. Guerraoui, R., Kapałka, M., Kouznetsov, P.: Boosting obstruction-freedom with low overhead. Technical report, EPFL (2006) Submitted for publication.
18. Guerraoui, R., Kapałka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Technical report, EPFL (2006)
19. Jayanti, P.: Robust wait-free hierarchies. Journal of the ACM **44**(4) (1997) 592–614
20. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: Proceedings of the 19th International Symposium on Distributed Computing (DISC'05). (2005)
21. Guerraoui, R., Schiper, A.: "Γ-accurate" failure detectors. In: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96), Springer-Verlag (1996)

Fully-Adaptive Algorithms for Long-Lived Renaming

Alex Brodsky[1], Faith Ellen[2], and Philipp Woelfel[2]

[1] Dept. of Applied Computer Science, University of Winnipeg, Winnipeg, Canada
a.brodsky@uwinnipeg.ca
[2] Dept. of Computer Science, University of Toronto, Toronto, Canada
{faith, pwoelfel}@cs.utoronto.ca

Abstract. Long-lived renaming allows processes to repeatedly get distinct names from a small name space and release these names. This paper presents two long-lived renaming algorithms in which the name a process gets is bounded above by the number of processes currently occupying a name or performing one of these operations. The first is asynchronous, uses LL/SC objects, and has step complexity that is linear in the number of processes, c, currently getting or releasing a name. The second is synchronous, uses registers and counters, and has step complexity that is polylogarithmic in c. Both tolerate any number of process crashes.

1 Introduction

Renaming is an interesting and widely-studied problem that has many applications in distributed computing. For *one-shot renaming*, each of the n processes in the system can perform GetName to get a distinct name from a small name space, $\{1, \ldots, m\}$. For *long-lived renaming*, a process that has gotten a name, x, can also perform RelName(x), so that it or another process can get this name later. In this more general version of the renaming problem, each process can alternately perform GetName and RelName any number of times (starting with GetName). After a process performs GetName, its name stays the same until after it next performs RelName. If a process performs GetName again, it may get the same name it got previously, or it may get a different name.

One application of long-lived renaming is the repeated acquisition and release of a limited number of identical resources by processes [12], something that commonly happens in most operating systems. In this case, names correspond to resources that are acquired and released by processes. Each process that wants to acquire a resource performs GetName to get a name, which is essentially permission for exclusive use of the resource. When a process no longer needs the resource, it performs RelName.

Another application of renaming is to improve the time or space complexity of an algorithm when only few processes participate [5]. For example, the time complexity of an algorithm may depend on the maximum number of processes that could participate, because it iterates over all processes in the system. Then

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 413–427, 2006.

a faster algorithm can be obtained by having each participating process first get a new name from a small name space and iterating over this smaller space. For some algorithms, such as those that implement shared objects, a process may participate for only short, widely spaced periods of time. Better performance can be achieved if each process performs `GetName` whenever it begins participating and performs `RelName` when it has finished participating for a while [21].

In one-shot renaming, the number of processes, k, that are getting or have gotten a name grows as an execution proceeds. In many applications, the final value of k is not known in advance. To avoid using an unnecessarily large name space, the size of the name space should be small initially and grow as k grows.

In long-lived renaming, we say that a process is *participating* from the time it begins a `GetName` operation until it completes its subsequent `RelName` operation. In particular, a process participates forever if it fails before releasing the last name that it got. The number of participating processes can increase and decrease during an execution of long-lived renaming. When the number is small, a process should get a small name, even though other processes may have received much larger names earlier in the execution (and may still be participating).

An $m(k)$-*renaming* algorithm [5] is a renaming algorithm in which a process always gets a name in the range $\{1, \ldots, m(k)\}$, where k is the number of processes that participate while it performs `GetName`. Note that, by the pigeonhole principle, $m(k)$-renaming is impossible unless $m(k) \geq k$. The special case when $m(k) = k$ is called *strong renaming.*

The cost of performing renaming is also an issue. Renaming algorithms in which the time complexities of `GetName` and `RelName` are bounded above by a function of the number of participating processes, k, are called *adaptive.* A renaming algorithm whose time complexity only depends on the number of processes, c, concurrently performing `GetName` or `RelName`, but not on the number of names that are in use, is called *fully-adaptive.* This is even better, because k can be much larger than c.

For some renaming algorithms, each process is assumed to have an identifier, which is a name from a large original name space. Other renaming algorithms also work when the original name space is infinite or when processes are anonymous (i.e. they have no original names).

Related Work. The renaming problem has been studied extensively in asynchronous systems beginning with Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [5], who studied one-shot renaming in asynchronous message passing systems. They proved that strong one-shot renaming is impossible, even if only one process can fail. They also proved that a process cannot decide on a new name until it has received messages (either directly or indirectly) from at least half of the other processes. Thus, in this model, adaptive one-shot renaming is impossible and one-shot renaming is impossible unless more than half of the processes in the system participate. In addition, they give two algorithms for one-shot renaming which assume that more than $n/2$ processes participate. The size of the name space in their algorithms also depends on n.

For synchronous message passing systems, Chaudhuri, Herlihy, and Tuttle [19,13] gave an algorithm for one-shot strong renaming with $O(\log k)$ rounds in which a process may send a message to any subset of other processes. They also proved a matching lower bound for comparison-based algorithms. Attiya and Djerassi-Shintel [6] studied the complexity of one-shot strong-renaming in semi-synchronous message passing systems that are subject to late timing faults. They obtained an algorithm with $O(\log k)$ rounds of broadcast from a synchronous message passing algorithm and proved an $\Omega(\log k)$ lower bound for comparison based algorithms or when the original name space is sufficiently large compared to k.

Bar-Noy and Dolev [11] showed how to transform the asynchronous message passing algorithms of Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [5], to shared-memory, using only reads and writes. They obtained a one-shot $\frac{k^2+k}{2}$-renaming algorithm that uses $O(n^2)$ steps per operation and a one-shot $(2k-1)$-renaming algorithm that uses $O(n \cdot 4^n)$ steps per operation.

Burns and Peterson [12] proved that long-lived m-renaming is impossible in an asynchronous shared memory system using only reads and writes unless $m \geq 2k-1$. They also gave the first long-lived $(2k-1)$-renaming algorithm in this model, but its time complexity depends on the size of the original name space. Herlihy and Shavit [18] proved the same lower bound on m for one-shot renaming. Herlihy and Rasjbaum [17] extended this result to systems that also provide set-consensus primitives.

In asynchronous shared memory systems using only reads and writes, the fastest adaptive one-shot $(2k-1)$-renaming algorithm has $O(k^2)$ step complexity [3]. There are also an adaptive one-shot $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity [8] and an adaptive one-shot $O(k^2)$-renaming algorithm with $O(k)$ step complexity [22,8]. For adaptive long-lived renaming, $O(k^2)$-time suffices for $O(k^2)$-renaming [1,4,9,20], but the fastest $(2k-1)$-renaming algorithm has $O(k^4)$ step complexity [7]. There are no fully-adaptive one-shot or long-lived renaming algorithms in this model.

With a single object that supports `Fetch&Increment`, fully-adaptive one-shot strong renaming is very easy: The object is initialized to 1. To get a name, a process simply performs `Fetch&Increment` and uses the responses as its new name [13].

For long-lived renaming, the only fully-adaptive algorithms use much stronger primitives. Moir and Anderson [22] presented a fully-adaptive long-lived strong renaming algorithm that uses an n-bit object storing the characteristic vector of the set of occupied names. This object supports two operations, `SetFirstZero` and bitwise-AND. `SetFirstZero` sets the first 0 bit in the object to 1 and returns the index of this bit. This index becomes the new name of the process that performed the operation. To release this name, the process sets this bit back to 0 by performing bitwise-AND with an n-bit string that has a single 0 in the corresponding position. A similar algorithm can be obtained using an n-bit `Fetch&Add` object in a synchronous system. These algorithms have constant time complexity. However, they use very large objects.

There are a number of adaptive, long-lived strong renaming algorithms from smaller, standard objects. For example, consider an array of n Test&Set objects, each initialized to 0. To get a name, a process performs Test&Set on the array elements, in order, until it gets response 0. Its name is the index of the element from which it gets this response. To release the name i, a process performs Reset on the i'th array element. This algorithm, presented in [22], performs GetName in $O(k)$ time and RelName in constant time. A similar algorithm uses a dynamically allocated doubly-linked list implemented using Compare&Swap, in which each node has a counter containing the number of larger names that are currently occupied, being acquired or being released [16].

The same idea can be used to obtain an adaptive long-lived strong renaming algorithm in a synchronous shared memory with $\lceil \log_2(U+1) \rceil$-bit registers, where U is the size of the original name space. This is because a Test&Set object can be implemented from a $\lceil \log_2(U+1) \rceil$-bit register in a synchronous system in constant time: a process competes for an unset Test&Set object by writing its name to a corresponding register.

Adaptive long-lived strong renaming can also be performed in $O(\log k)$ time. Use a sequence of $O(\log n)$ complete binary trees. The first tree has height 1 and the height of each subsequent tree increases by 1. The leaves of these trees correspond to the new names and a process gets a name by acquiring a leaf. Each node in the tree contains a counter that denotes the number of free leaves in the subtree rooted at that node. To acquire a leaf, a process accesses the counters at the root of each tree until it finds one with a free leaf. It then proceeds down the tree to find a free leaf, using the value of the counter at each node to guide its search and decrementing the counters on this path as it descends. To release its name, the process starts at the corresponding leaf and walks up the tree, incrementing the counter in each node it visits, until it reaches the root. In a synchronous system, each counter can be implemented by an $O(\log n)$-bit object that supports Fetch&Decrement and Increment. In an asynchronous system, a bounded version of Fetch&Decrement is needed, which does not change the value of the object when it is 0. (See [22] for details.)

Our Results. In this paper, we present two new fault tolerant and fully-adaptive algorithms for long-lived strong renaming. They are the first such algorithms that do not rely on storing a representation of the set of occupied names in a single (very large) object. The first algorithm is asynchronous and uses $\Theta(\log U)$-bit LL/SC objects, where U is the size of the original name space. Its step complexity is $O(c)$, where c is the number of processes concurrently performing GetName or RelName. The second algorithm is synchronous, uses $O(\log n)$-bit counters and registers, and has $O(\log^3 c/ \log\log c)$ step complexity. Both algorithms tolerate any number of process crashes.

The key to both algorithms is an interesting sequential data structure that supports GetName and RelName in constant time. We then apply a universal adaptive construction by Afek, Dauber and Touitou [2] to obtain our fully-adaptive asynchronous renaming algorithm. This is presented in Section 3. In

Section 4, we develop our fully-adaptive synchronous algorithm. Directions for future work are discussed in Section 5.

2 Models

We consider models of distributed systems in which n deterministic processes run concurrently and communicate by applying operations to shared objects. Our implementations make use of (multi-writer) registers, counters, and LL/SC objects. A register stores an integer value and supports two operations: $\ell \leftarrow \texttt{read}(R)$, which reads register R and assigns the value to the local variable ℓ, and write $R \leftarrow \ell$, which writes the value of ℓ into register R. A counter, C, supports read, write, $\ell \leftarrow \texttt{Fetch\&Increment}(C)$, and $\ell \leftarrow \texttt{Fetch\&Decrement}(C)$. These last two operations assign the current value of C to ℓ and then increment and decrement C, respectively. An LL/SC object, O, supports write and $\ell \leftarrow \texttt{Load-Linked}(O)$, which reads the value in object O and assigns it to ℓ. It also supports Store-Conditional $O \leftarrow \ell$, which stores the value of ℓ to O only if no store to O has occurred since the same process last performed Load-Linked(O). In addition, this operation returns a Boolean value indicating whether the store occurred. We assume that all operations supported by these objects occur atomically.

In asynchronous systems, an adversarial scheduler decides the order in which processes apply operations to shared objects. The adversary also decides when processes begin performing new instances of GetName and RelName. In synchronous systems, the order chosen by the adversary is restricted. Time is divided into rounds. In each round, every process that is performing an instance of GetName or RelName (and has not crashed) applies one operation. The order chosen by the adversary can be different for different rounds. We assume that the adversary only begins new instances of GetName or RelName in a round if no other instances are in progress. We also assume that the adversary does not begin an instance of GetName and an instance of RelName in the same round. These assumptions can be removed by having a flag which processes update frequently to indicate they are still working on the current batch of operations. Other processes wait until a batch is finished before starting a new batch. This is discussed in more detail in the full version of the paper.

In both models, we measure the complexity of an instance of GetName or RelName by the number of operations it applies. A process that crashes simply performs no further operations. Our algorithms tolerate any number of process crashes.

3 Asynchronous Renaming

Afek, Dauber, and Touitou [2] have shown that, using LL/SC objects, fast sequential implementations of an object suffice for obtaining fully-adaptive implementations:

Theorem 1. *If an object has a sequential implementation in which an update takes a constant number of steps, then it also has a fully-adaptive implementation from* `LL/SC` *objects in an asynchronous system such that an update takes $O(c)$ steps, where c is the number of processes that update the object concurrently.*

This is a special case of their universal construction, which maintains a queue of operations to be performed on the object. To perform an operation, a process records the operation it wants to perform and enqueues its identifier. Then it repeatedly helps the processes at the head of the queue to complete their operations, until its own operation is completed. Processes use `Load-Linked` and `Store-Conditional` to agree on the results of each operation and how the value of the object changes.

We consider a renaming object whose value is the subset of free names in $\{1,\ldots,n\}$. Here n is the number of processes in the system and, thus, an upper bound on the number of names that will ever be needed. This object supports two operations, `GetName` and `RelName`. If $\mathcal{F}$ is the set of free names, then an instance of `GetName` removes and returns a name from $\mathcal{F}$ which is less than or equal to the maximum number of participating processes at any point during the execution of the instance. `RelName(`x`)` returns x to $\mathcal{F}$. It can only be applied by the last process that received x from a `GetName` operation.

Next, we describe a data structure for representing $\mathcal{F}$ and constant time sequential algorithms for performing `GetName` and `RelName`. From these, we get our first fully-adaptive, long-lived, strong renaming implementation, by applying Theorem 1.

Theorem 2. *In an asynchronous system in which processes communicate using* `LL/SC` *objects, there is a fully-adaptive implementation of long-lived strong renaming which performs* `GetName` *and* `RelName` *in $O(c)$ steps.*

The `LL/SC` objects used by this implementation must be able to store process identifiers. If these identifiers are from an original name space of size U, then the `LL/SC` objects must have at least $\lceil \log_2 U \rceil$ bits.

Data Representation. The subset $\mathcal{F} \subseteq \{1,\ldots,n\}$ of free names is represented by two arrays of n registers, D and L, and two counters, M and F. (In the sequential implementation, which is related to Hagerup and Raman's quasidictionary [15], the counters M and F can be replaced by registers.) The values in M, F, and the entries of D are in $\{0,1,\ldots,n\}$. The entries in L are in $\{1,\ldots,n\}$. The array D is used as a direct access table. We say that name i is *occupied* if $D[i] = 0$. If i is occupied and p was the last process that wrote 0 to $D[i]$, then we say that p *occupies* name i. The following invariant will be maintained:

> At any point, if a process p has received name i as its response from a call of `GetName` and, since then, has not called `RelName(`i`)`, then i is occupied by p. (1)

A name that is neither occupied nor free is called *reserved.* Reserved names occur while processes are performing `GetName` and `RelName`, and because of process failures. A name $i > M$ is free if and only if it is unoccupied.

The variable F is the number of free names less than or equal to M and $L[1..F]$ is an unsorted list of the free names less than or equal to M. In particular,

$$1 \le L[j] \le M \text{ for all } 1 \le j \le F \tag{2}$$

is an invariant of our data structure. We use $\mathcal{L}$ to denote the set of names in $L[1..F]$. Another important invariant is the following:

$$D\big[L[j]\big] = j \text{ for all } 1 \le j \le F. \tag{3}$$

This ensures that all free names are unoccupied, there are F different names in $\mathcal{L}$, and, if $i \in \mathcal{L}$, then $D[i]$ is a pointer to a location in L, between 1 and F, that contains the name i. The function $\texttt{InL}(i)$ checks whether name i is in $\mathcal{L}$ using a constant number of operations. Specifically, if $i \in \mathcal{L}$, then $\texttt{InL}(i)$ returns $D[i]$, the unique index $1 \le j \le F$ such that $L[j] = i$. Otherwise, it returns 0.

Note that $i \in \mathcal{F}$ if and only if either $(1 \le i \le M \text{ and } i \in \mathcal{L})$ or $(i > M \text{ and } D[i] > 0)$. There are two other invariants that are maintained by the data structure:

$$M \le \text{ the number of participating processes, and} \tag{4}$$

$$\text{the number of occupied names} + \text{ the number of reserved names} \le M. \tag{5}$$

The data structure is illustrated in Figure 1. Blank entries denote arbitrary values in $\{1, \ldots, n\}$.

Initially, M and F are both 0 and all entries of D are positive, so $\mathcal{F} = \{1, \ldots, n\}$. Furthermore, no processes are participating and there are no occupied or reserved names. Hence, all the invariants are satisfied.

To prove correctness of `GetName` and `RelName`, we must prove that they always maintain the invariants. Furthermore, the name a process receives as its response from a call of `GetName` must be at most the maximum number of participating processes at any point during its execution of this instance of `GetName`.

GetName. The sequential procedure for getting names is quite straightforward. First, a process that wants to get a name, takes the tentative name $M + 1$ and increments M. If this name is unoccupied, the process gets it. If it is occupied, there must be a free name less than or equal to M, since, by Invariant 5, there are at most M names that are occupied or reserved. Thus $\mathcal{L}$ is not empty. In this case, the process decrements F and gets the name that was at the end of the unsorted list. Finally, in both cases, the process occupies the name, x, it got by writing 0 into the corresponding entry, $D[x]$, in the direct access table.

Lemma 1. *No step of* `GetName` *causes any invariant to become invalid. The name returned by a call of* `GetName` *is bounded by the maximum number of participating processes at any point during its execution.*

The step complexity of `GetName` is $O(1)$. A process that fails after incrementing M, but before writing 0 to some element of D, may decrease the number of free names, but does not increase the number of occupied names.

```
Function GetName
  local: $x, j, z$
  $x \leftarrow 1 +$ Fetch&Increment($M$)
  $z \leftarrow$ read($D[x]$)
  if $z = 0$ then
      $j \leftarrow$ Fetch&Decrement($F$)
      $x \leftarrow$ read($L[j]$)
  end
  write $D[x] \leftarrow 0$
  return($x$)
```

```
Function RelName($x$) /* sequential */
  Input: name $x$
  local: $m$, $f$, $i$, $j$

  write $D[x] \leftarrow n$
  $m \leftarrow$ read($M$)
  $f \leftarrow$ read($F$)
  if $x \leq m$ then
      /* append $x$ to $L[1..F]$          */
      $f \leftarrow f + 1$
      write $D[x] \leftarrow f$
      write $L[f] \leftarrow x$
      write $F \leftarrow f$
  end
  $j \leftarrow$ InL($m$)
  if $j > 0$ then
      /* remove name $m$ from $\mathcal{L}$        */
      write $F \leftarrow f - 1$
      $i \leftarrow$ read($L[f]$)
      write $D[i] \leftarrow j$
      write $L[j] \leftarrow i$
  end
  write $M \leftarrow m - 1$
```

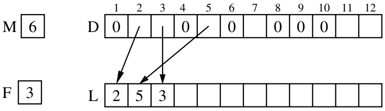


Fig. 1. The functions GetName and RelName and an example of the data structure

Releasing Names Sequentially. The sequential algorithm for releasing a name consists of two phases. In the first phase, a process changes the status of its name, x, from occupied to free. It begins this phase by writing any positive number, for example n, into $D[x]$. This changes x from being occupied to being unoccupied. If $x > M$, then x is now free. Otherwise, the process has to append x to the end of the list $L[1..F]$. This is accomplished by writing x into $L[F+1]$, writing $F+1$ into $D[x]$ and then incrementing F. In the second phase, M is decremented. Before doing so, if it is present, the name M must be removed from $\mathcal{L}$. Suppose that it is in location $L[j]$ and that $L[F] = i$. First F is decremented. Then $D[i]$ is updated to point to location j. Finally, i is copied into $L[j]$, overwriting the value M that was stored there. The order in which these operations are performed is important to ensure that the data structure invariants are maintained.

Lemma 2. *If a process occupying the name x calls the sequential version of* RelName(x)*, then none of its steps causes any invariant to become invalid.*

The step complexity of this sequential version of RelName is $O(1)$. A process that fails after overwriting the 0 in the direct access table entry corresponding to its name, but before decrementing M, decreases the number of occupied names, but might not increase the number of free names.

4 Synchronous Renaming

For synchronous renaming, the data representation is the same as in Section 3. The sequential version of GetName also works when multiple processes begin performing new instances at the same time. Recall that, in our model, no instances of GetName and RelName are performed concurrently. The correctness of GetName follows, as in the sequential case, from Lemma 1.

Unfortunately, the sequential version of RelName does not work if there are multiple processes simultaneously trying to release their names. For example, several processes could write their names to the same location in L. The solution to this problem is to assign distinct ranks to the processes that want to add their names to $\mathcal{L}$. Then the process with rank i can write its name into location $L[F+i]$.

The function Rank can be implemented using a counter, to which processes perform write(*1*) and then Fetch&Increment. Similarly, CountAlive, which counts the number of processes that perform it simultaneously, can be implemented using a counter which is set to 0, incremented, and then read by each of those processes.

If processes crash after they perform Rank, but before they write their names into L, there will be garbage interspersed between the names that have been written. One way to handle this is to write 0's in the portion of L that will be written to before processes write their names there. Afterwards, the names that have been written can be compacted, by removing the 0's that remain. To do this and to solve other similar problems encountered when parallelizing RelName, we use three auxilliary functions, WriteAll, Count, and Compact. They are based on the synchronous DoAll algorithm by Georgiou, Russell, and Shvartsman [14].

WriteAll performs the set of instructions, write $dest(i) \leftarrow val(i)$, for $1 \leq i \leq s$, in a fault tolerant way. Here $dest(i)$ denotes a destination register and $val(i)$ denotes the value to be written there. For example, WriteAll($L[i+f] \leftarrow 0,\ 1 \leq i \leq s$) writes 0's to the s locations following $L[f]$. Georgiou, Russell, and Shvartsman [14, Theorem 8] give an implementation of WriteAll from multi-writer registers which take $O(\log^2 s/\log\log s)$ steps to complete, provided at least $s/2$ of the processes that are simultaneously executing it do not fail. However, if too many processes fail, the remaining processes could take too long to complete all s tasks.

To ensure that RelName remains fully-adaptive, we modify the function WriteAll($dest(i) \leftarrow val(i),\ 1 \leq i \leq s$) so that it returns whenever fewer than $s/2$ processes remain. This is accomplished by having processes perform "if CountAlive $< s/2$ then return" after every constant number of steps. If WriteAll terminates in this way, there are no guarantees about which, if any, of the s tasks have been performed. In this case, the call returns ∞ and we say that it fails. Otherwise, WriteAll returns s.

Note that it does not suffice for processes to wait for a convenient place in the code, such as the end of a phase, to check whether too many processes have crashed. For example, if there is one process that wants to perform GetName just after all but one of the s processes performing RelName crash, then the number of participating processes, c, is 2. If the process has to wait to start performing `GetName` until the one surviving process completes a phase (whose complexity depends on s), the resulting implementation will not be fully-adaptive.

The function $\texttt{Count}(z(i), 1 \leq i \leq s)$ returns the number of values of $i \in \{1, \ldots, s\}$ for which the predicate $z(i)$ is true, provided that at least $s/2$ of the processes that simultaneously begin executing this function, complete it. Otherwise, it returns ∞ and we say that it fails. All processes that complete a particular instance of `Count` return the same value. The implementation of `Count` is very similar to `WriteAll` and it has the same performance [23, Theorem 5.9].

The function $\texttt{Compact}(A, z(i), 1 \leq i \leq s)$ compacts the elements $A[i]$ of the array $A[1..s]$ such that the predicate $z(i)$ is true, so that these elements are stored contiguously starting from the beginning of A. If at least $s/2$ of the processes that simultaneously begin executing this function, complete it, then the processes return the number of $i \in \{1, \ldots, s\}$ for which $z(i)$ is true. If not, the call fails, returning the value ∞, and the contents of $A[1..s]$ can be arbitrary. `Compact` has the same performance as `WriteAll`.

Lemma 3. *If $val(i)$, $dest(i)$, and $z(i)$ can be computed by a process in a constant number of steps, then there are implementations of* $\texttt{Compact}(A, z(i), 1 \leq i \leq s)$, $\texttt{Count}(z(i), 1 \leq i \leq s)$, *and* $\texttt{WriteAll}(dest(i) \leftarrow val(i),\ 1 \leq i \leq s)$ *that have* $O(\log^2 s / \log\log s)$ *step complexity and return within a constant number of steps when fewer than $s/2$ of the processes which simultaneously began executing the same instance remain.*

4.1 Releasing Names

The procedure for releasing names consists of two phases. In the first phase, a process changes the status of its name x. If $x > M$, then it can simply write n (or any other value larger than 0) into $D[x]$ and the name becomes free right away. Otherwise, the process has to insert the name into list $\mathcal{L}$. This is achieved by a call to the procedure $\texttt{InsertInL}(x)$. If several processes call `InsertInL`, but some of them fail, then the number of names that are inserted into $\mathcal{L}$ will be at least the number of these processes that complete their calls. Each name inserted into $\mathcal{L}$ will be a name that was occupied by one of the calling processes. However, it is not necessarily the case that the inserted names include all those that were occupied by the processes completing their calls.

In the second phase, the processes fix the invariants that involve M. For the upper bound on M in Invariant (4), we have to reduce M by at least s, if s processes complete their call to `RelName` (and thus stop participating). To avoid violating the lower bound on M in Invariant (5), we cannot reduce M by more than ℓ, if ℓ processes successfully complete the first phase. Moreover, before we can reduce M from m to m', we have to remove all names in $\{m' + 1, \ldots, m\}$ from the list $\mathcal{L}$ to maintain Invariant (2). This procedure of removing the desired names from $\mathcal{L}$ and reducing M is performed by the procedure `RemoveLargeNames`. The implementation guarantees that if ℓ processes

```
Function RelName(x)
/* synchronous */
  Input: name $x$
  local: $m$, $success$

  $m \leftarrow$ read($M$)
  if $x \leq m$ then
      InsertInL($x$)
  else
      write $D[x] \leftarrow n$
  end
  RemoveLargeNames
```

```
Function InsertInL(x)
  Input: name $x$
  local: $f$, $i$, $s$
  register: $R$

  $f \leftarrow$ read($F$)
  repeat
      $s \leftarrow$ CountAlive
      WriteAll($L[f+i] \leftarrow 0,\ 1 \leq i \leq s$)
      $i \leftarrow$ Rank
      write $L[f+i] \leftarrow x$
      $a \leftarrow$ Compact($L[f+1..f+s],\ L[f+i] > 0,$
                  $1 \leq i \leq s$)
      WriteAll$\big(D\big[L[f+i]\big] \leftarrow f+i,\ 1 \leq i \leq a\big)$
  until CountAlive $\geq a/2$
  write $F \leftarrow f+a$
```

call RemoveLargeNames during their call to RelName and s of them finish, then M is reduced by at least s and at most ℓ.

Lemma 4. *Suppose a set of processes, each occupying a name with value at most M, simultaneously call* InsertInL. *If ℓ of these processes complete that call, then at least ℓ names are added to $\mathcal{L}$. The invariants remain true throughout the execution of the call. At any point during the execution, if ℓ' of these processes have not failed, then the call terminates within $O(\log^3 \ell' / \log\log \ell')$ steps.*

Lemma 5. *Suppose a set of ℓ processes simultaneously call* RemoveLargeNames. *If s of these processes complete their call, then $m - \ell \leq m' \leq m - s$, where m and m' are the values of M immediately before and after the call to* RemoveLargeNames. *The invariants remain true throughout the execution of the call. At any point during the execution, if ℓ' of these processes have not failed, then the call terminates within $O(\log^3 \ell' / \log\log \ell')$ steps.*

The correctness of RelName follows from these lemmas. In the remainder of this section, we show how to implement InsertInL and RemoveLargeNames.

Inserting into $\mathcal{L}$. In Procedure InsertInL, processes first write their names to distinct locations following the end of the list $L[1..F]$ and update the entries in the direct access table D to point to these names. Multiple attempts may be needed, due to process failures. Once they succeed, the processes increment register F, which moves these names into $\mathcal{L}$.

In each attempt, a process first uses CountAlive to find the number of surviving processes, s, that want to free their names. Then, using WriteAll, array elements $L[F+1], \ldots, L[F+s]$ are initialized to 0. Next, each of these processes writes its name into $L[F+i]$, where i is a unique rank between 1 and s assigned to it by Rank. Since some of these processes may have failed prior to writing their names, a, the number of names that were written, may be less than s. Compact is performed to compact $L[F+1..F+s]$, so that these a names are stored in $L[F+1..F+a]$. Finally, another `WriteAll` is performed, setting $D\big[L[F+i]\big] = F+i$, for $i = 1,\ldots,a$, to ensure that Invariant (3) will hold after F is increased to $F+a$. If fewer than $a/2$ processes complete the attempt, another attempt is made with s decreased by at least a factor of 2.

If at least $a/2$ processes complete the attempt, then neither of the calls to `WriteAll` nor the call to `Compact` failed. In this case, F is incremented by a, moving the a names that were appended to L into $\mathcal{L}$. Note that a is the number of processes that write their names into $L[F+1..F+s]$ during the last attempt and, hence, it is an upper bound on the number of processes that complete the call.

By Lemma 3, each attempt that starts with s processes takes $O(\log^2 s/\log\log s)$ steps. Since s decreases by at least a factor of 2 each subsequent attempt, there are at most $\log s$ attempts, for a total of $O(\log^3 s/\log\log s)$ steps, from this point on. If, at any point during an attempt, more than three quarters of the processes that started the attempt have failed, then within a constant number of steps, the attempt will fail.

Removing Large Names from $\mathcal{L}$. Procedure `RemoveLargeNames` is called simultaneously by every process performing `RelName`, after they no longer occupy their names. Thus, processes with names greater than M must wait before they begin `RemoveLargeNames`, until the processes that are performing `InsertInL` are finished.

First, consider an execution in which s processes call `RemoveLargeNames` and none of them crash. In this case, M will be reduced to $M-s$ after all the *large* names, that is, those which are greater than $M-s$, are removed from $\mathcal{L}$. Names $M-s$ or less will be called *small*.

After processes store the value of F in their local variable f, they determine the number, a, of large names in $L[1..f]$ using `Count`. Next, they copy the last a names in $L[1..f]$ to a temporary array H_1 using `WriteAll`. Then, they remove the large names from H_1 using `Compact`. At this point, $H_1[1..b]$ consists of all the small names in the last a locations of $L[1..f]$. When F is decreased to $f-a$, these small names are no longer in $\mathcal{L}$ and become reserved, instead of free.

The number of large names that remain in $\mathcal{L}$ is exactly equal to b, the number of small names in H_1. These large names are all between $M-s+1$ and M, so they and their locations in $L[1..f-a]$ can be found from $D[M-s+1..M]$ using `WriteAll` and `Compact`. They are stored in the temporary arrays $H_2[1..b]$ and $G[1..b]$, respectively.

The final steps are to overwrite each of the b large names in $L[1..f-a]$ with a different name in $H_1[1..b]$ and then decrement M by s. But, to ensure that Invariant (3) is not violated, the direct access table entries, $D[j]$, of the names $j \in H_1[1..b]$ should first point to the b different locations in $L[1..f-a]$ that contain large names. This is done using `WriteAll`. Then each name $j \in H_1[1..b]$ can be written to location $L[D[j]]$, also using `WriteAll`.

Difficulties arise when processes crash. One problem is that this changes s and, hence, some large names become small names. If too many processes fail when the small names in $L[f-a+1..f]$ are copied into $H_1[1..b]$, the phase is

```
Function RemoveLargeNames
  local: $f$, $a$, $b$, $e$, $e'$, $i$, $j$, $m$
  register: $G[1\ldots n]$, $T[1\ldots n]$, $H_1[1\ldots n]$, $H_2[1\ldots n]$

  $m \leftarrow$ read($M$)
  $f \leftarrow$ read($F$)
  repeat
      $s \leftarrow$ CountAlive
      $a \leftarrow$ Count(InL($m - s + 1$) $> 0,\ 1 \le i \le s$)
      WriteAll($H_1[i] \leftarrow L[f - a + i],\ 1 \le i \le a$)
      $b \leftarrow$ Compact($H_1,\ H_1[i] \le m - s,\ 1 \le i \le a$)
  until $b \ne \infty$
  write $F \leftarrow f - a$
  repeat
      repeat
          $s \leftarrow$ CountAlive
          WriteAll($G[i] \leftarrow D[m - s + i],\ 1 \le i \le s$)
          $b \leftarrow$ Compact($G$, InL($m - s + i$) $> 0,\ 1 \le i \le s$)
          WriteAll($H_2[i] \leftarrow L\big[G[i]\big],\ 1 \le i \le b$)
          $e \leftarrow$ WriteAll($D\big[H_1[i]\big] \leftarrow G[i],\ 1 \le i \le b$)
      until $e \ne \infty$
      $e \leftarrow$ WriteAll($L\big[G[i]\big] \leftarrow H_1[i],\ 1 \le i \le b$)
      if $e = \infty$ then
          repeat
              $s \leftarrow$ CountAlive
              WriteAll($T[i] \leftarrow H_1[i],\ 1 \le i \le s$)
              $h \leftarrow$ Compact($T$, InL($G[i]$) $= 0,\ 1 \le i \le s$)
              WriteAll($T[h + i] \leftarrow H_2[i],\ 1 \le i \le s$)
              Compact($T[h + 1..h + s]$, InL($G[i]$) $= 0,\ 1 \le i \le s$)
              $e' \leftarrow$ Compact($T,\ T[i] \le m - s,\ 1 \le i \le s$)
          until $e' \ne \infty$
          repeat
              $s \leftarrow$ CountAlive
              $b \leftarrow$ WriteAll($H_1[i] \leftarrow T[i],\ 1 \le i \le \min\{e', s\}$)
          until $b \ne \infty$
      end
  until $e \ne \infty$
  write $M \leftarrow m - s$
```

simply repeated with s decreased by at least a factor of 2. The same is true for the second phase, in which large names in $L[1..f - a]$ are copied into H_2, their locations are copied into G, and the direct entry table entries for elements in H_1 are made to point to these locations.

However, a failure of the call to WriteAll, in which small names in H_1 overwrite large names in $L[1..f - a]$ means that an unknown subset of these writes have occurred. In this case, H_1 has to be fixed up before starting again from the beginning of the second phase, with s decreased by at least a factor of 2.

Only the small name $H_1[j]$ can overwrite the large name $H_2[j]$ in L. Therefore, exactly one of $H_1[j]$ and $H_2[j]$ is in $L[1..f - a]$. So, to fix up H_1, the first s elements of each of H_1 and H_2 are copied to a temporary array T, using `WriteAll`. Then names in $L[1..f-a]$ and large names are removed from T using `Compact`. Because processes can fail during the construction of T, all this has to be repeated, with s decreased by at least a factor of 2, until none of the calls to `WriteAll` and `Compact` fail.

Finally, a sufficiently long prefix of T is copied back into H_1. This is also repeated, with s decreased by at least a factor of 2, until it does not fail.

Theorem 3. *In a synchronous system in which processes communicate using counters and registers, there is a fully-adaptive implementation of long-lived strong renaming which performs* `GetName` *and* `RelName` *in* $O(\log^3 c / \log\log c)$ *steps.*

5 Conclusions

In this paper, we described the first fault-tolerant fully-adaptive implementations of long-lived strong renaming that do not use base objects with $\Omega(n)$ bits. One algorithm is asynchronous and uses `LL/SC` objects in addition to registers. Because processes help one another to get new names, the original names of processes are used for identification purposes. The other algorithm is synchronous, but uses counters and registers. Moreover, its step complexity is substantially smaller: polylogarithmic instead of linear in c. This algorithm never uses the original names of processes, so it also works for systems of anonymous processes.

Fully adaptive one-shot renaming is very easy to implement using a counter. Hence, it was natural to use a shared memory system with counters when trying to get a fully-adaptive long-lived renaming implementation. Two specific open questions arise from this work: First, can more efficient or simpler fully-adaptive long-lived renaming algorithms be obtained using other strong memory primitives, for example `Compare&Swap`? Second, are there fully-adaptive renaming algorithms or more efficient adaptive renaming algorithms using only registers?

Acknowledgements

We are grateful to Hagit Attiya for helpful discussion. This research was supported by NSERC, the Scalable Synchronization Group at Sun Microsystems, and the German Research Foundation (grant WO1232/1-1).

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–104, May 1999.
2. Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 538–547, May 1995.

3. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112, 1999.
4. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15:67–86, 2002.
5. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
6. H. Attiya and T. Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *Lecture Notes in Computer Science*, 725:204–214, 1993.
7. H. Attiya and A. Fouren. Polynomial and adaptive long-lived $(2k-1)$-renaming. *Lecture Notes in Computer Science*, 1914:149–159, 2000.
8. H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.
9. H. Attiya and A. Fouren. Algorithms adaptive to point contention. *Journal of the ACM*, 50(4):444–468, 2003.
10. H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics (2nd Ed.).* Wiley, 2004.
11. A. Bar-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the 8th Annual ACM Symposium on Principles of distributed computing*, pages 307–318, Aug. 1989.
12. J. Burns and G. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 145–158, Aug. 1989.
13. S. Chaudhuri, M. Herlihy, and M. Tuttle. Wait-free implementations in message-passing systems. *Theoretical Computer Science*, 220(1):211–245, 1999.
14. C. Georgiou, A. Russell, and A. A. Shvartsman. The complexity of synchronous iterative do-all with crashes. *Distributed Computing*, 17:47–63, 2004.
15. T. Hagerup and R. Raman. An efficient quasidictionary. In *Proceedings of SWAT*, volume 2368 of *Lecture Notes in Computer Science*, pages 1–18, 2002.
16. M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. *Electr. Notes Theor. Comput. Sci*, 78, 2003.
17. M. Herlihy and S. Rajsbaum. Algebraic spans. *Math. Struct. in Comp. Science*, 10:549–573, 2000.
18. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, Nov. 1999.
19. M. Herlihy and M. Tuttle. Lower bounds for wait-free computation in message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 347–362, Aug. 1990.
20. M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. *Lecture Notes in Computer Science*, 2180:123–133, 2001.
21. M. Moir. Fast, long-lived renaming improved and simplified. *Science of Computer Programming*, 30(3):287–308, Mar. 1998.
22. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, Oct. 1995.
23. A. A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.

Constructing Shared Objects That Are Both Robust and High-Throughput

Danny Hendler* and Shay Kutten**

Faculty of Industrial Engineering and Management,
Technion, Haifa, Israel

Abstract. Shared counters are among the most basic coordination structures in distributed computing. Known implementations of shared counters are either blocking, non-linearizable, or have a sequential bottleneck. We present the first counter algorithm that is both linearizable, non-blocking, and can provably achieve high throughput in semisynchronous executions. The algorithm is based on a novel variation of the software combining paradigm that we call *bounded-wait combining.* It can thus be used to obtain implementations, possessing the same properties, of any object that supports combinable operations, such as stack or queue. Unlike previous combining algorithms where processes may have to wait for each other indefinitely, in the bounded-wait combining algorithm a process only waits for other processes for a bounded period of time and then 'takes destiny in its own hands'.

In order to reason rigorously about the parallelism attainable by our algorithm, we define a novel metric for measuring the throughput of shared objects which we believe is interesting in its own right. We use this metric to prove that our algorithm can achieve throughput of $\Omega(N/\log N)$ in executions where process speeds vary only by a constant factor, where N is the number of processes that can participate in the algorithm.

We also introduce and use *pseduo-transactions* - a technique for concurrent execution that may prove useful for other algorithms.

1 Introduction

At the heart of many distributed systems are *shared objects* - data structures that may be concurrently accessed by multiple processes. The most widely-used correctness condition for shared objects is *linearizability*, introduced by Herlihy and Wing [14]. Intuitively, linearizability requires that each operation appears to take effect instantaneously at some moment between its invocation and response. *Lock-free* implementations of shared objects require processes to coordinate without relying on mutual exclusion. They are considered more robust, as they avoid the inherent problems of locking, such as deadlock, convoying, and priority inversion.

* Supported in part by the Technion Aly Kaufman Fellowship and by the Israeli ministry of science and technology.

** Supported in part by the Technion foundation for the promotion of research.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 428–442, 2006.

A shared counter is an object that holds an integer and supports the *fetch&increment* operation for atomically incrementing the counter and returning its previous value. Shared counters are among the most basic coordination structures in distributed computing. Consequently, efficient implementations of shared counters received considerable attention in the literature. In spite of these efforts, existing counter implementations are either non-linearizable, blocking, or inherently sequential. This paper presents the first counter algorithm that is both linearizable, nonblocking, and highly parallel.

If the hardware supports the *fetch&increment* primitive, then the simplest way to implement a counter, shared by N processes, is by using the following trivial algorithm: all processes share a single base object on which they perform the *fetch&increment* operation to get a number. Although this central counter is both linearizable and nonblocking (it is, in fact, *wait-free* [11]), it has a sequential bottleneck. Specifically, the worst-case time complexity of this implementation is $\Omega(N)$: if all processes attempt to apply their operations simultaneously to the central counter, the last to succeed incurs a delay linear in N while waiting for all other earlier processes to complete their operations.

Fich et al. [4] proved an $\Omega(N)$ time lower bound on obstruction-free [12] implementations of a wide class of shared objects, that includes counters, stacks and queues. This lower bound establishes that no nonblocking counter algorithm can improve on a central counter in terms of worst-case time complexity. This does not preclude, however, the existence of nonblocking counter algorithms that achieve better worst-case time complexity in semisynchronous executions. Indeed, the worst-case time complexity of our algorithm in such executions is $O(\log N)$, yielding maximal throughput of $\Omega(N/logN)$.

To allow parallelism, researchers proposed using highly-distributed coordination structures such as *counting networks*. Counting networks were introduced by Aspnes et al. [2]. Though they are wait-free and allow parallelism, the counting networks of [2] are non-linearizable. Herlihy et al. demonstrated that counting networks can be adapted to implement wait-free linearizable counters [13]. However, the first counting network they present is blocking while the others do not provide parallelism, as each operation has to access $\Omega(N)$ base objects.

A well-established technique for constructing highly parallel shared objects is that of *combining*. Goodman et al. [5] and Yew et al. [19] used combining for implementing *fetch&add*. In both these algorithms, the current value of the counter is stored at the root of a binary tree. A process applies its operation starting from its leaf and climbing along the path to the root. Whenever two processes meet at an internal node, they combine their operations by generating a single request for adding the sum of both requests. One of these processes proceeds in climbing the tree while the other is blocked and waits at the node. When a process reaches the root, it adds to the central counter the sum of all the requests with which it combined and then starts a process of propagating responses back to the blocked processes. Combining trees can be used to implement linearizable counters and allow high parallelism but are blocking.

Shavit and Zemach introduce *diffracting trees* [18] to replace the "static" tree used in software combining with a collection of randomly created dynamic trees. Diffracting trees can be used to implement shared counters that are linearizable and allow parallelism but are blocking. Hoai Ha et al. introduce another version of (blocking) adaptive combining trees [8].

We introduce a variation on the software combining paradigm, that we call *bounded-wait combining*. Unlike in previous software combining algorithms, where processes may have to wait indefinitely for other processes, in bounded-wait combining, a process only waits for other processes for a bounded period of time and then 'takes destiny in its own hands'. As long as process speeds do not differ by more than some known fixed factor, processes wait for each other, eliminating contention for memory and achieving high parallelism. When the execution is asynchronous, however, processes fall back to an asynchronous modus operandi where high parallelism cannot be guaranteed but progress is. The result is *the first implementation of a linearizable counter that is both nonblocking and provably achieves high parallelism in semisynchronous executions*.

Our algorithm uses Greenwald's *two-handed emulation* mechanism [7] to implement a construct we term *pseudo-transactions*. Pseudo-transactions are weaker than ordinary transactions in that they are not atomic but they permit higher parallelism. Though they cannot replace transactions in general, we believe pseudo-transactions may prove useful also for other algorithms.

Geenwald's two-handed emulation is known to be sequential. However, We use it to implement pseudo-transactions by allowing different processes to operate on different two-handed emulation objects *in parallel*. The two-handed emulation mechanism uses the double compare-and-swap (DCAS) primitive. DCAS can be emulated efficiently by single-word compare-and-swap (CAS) by using, e.g., the algorithm of Harris et al. [9].

Bounded-wait combining can be adapted to work for any *combinable operation* [6,15] and can thus be used to implement nonblocking and highly parallel linearizable stacks and queues. We are not aware of any other stack or queue algorithm to possess these properties.

Hendler et al. presented the *elimination-backoff stack*, a nonblocking linearizable stack algorithm [10]. Their empirical results show that their algorithm achieves high parallelism in practice; nevertheless, it does not provide any deterministic guarantee of parallelism. Moir et al. used ideas similar to these of [10] to obtain a queue algorithm that possesses the same properties [17].

In order to be able to reason rigorously about the parallelism attainable by our algorithm, we define a novel metric for the throughput of shared objects that may be interesting in its own right. By throughput, we mean the ratio between the number of operations that complete in an execution and the execution's duration. The key to this metric is a definition of time that assigns identical times to events that access different base objects and may be executed concurrently. To the best of our knowledge, this is the first formal metric for the throughput of shared objects. We use this metric to prove that our algorithm can achieve maximal throughput of $\Omega(N/\log N)$ in semisynchronous executions.

Model and Definitions. We consider a standard model of an asynchronous shared memory system, in which a finite set of asynchronous processes communicate by applying operations to shared objects [3].

Shared objects are implemented from *base objects*, such as read/write registers, provided by the system. A *configuration* specifies the value of each *base object* and the state of each process. An *initial configuration* is a configuration in which all the base objects have their initial values and all processes are in their initial states. To apply their operations, processes perform a sequence of *steps.* Each step consists of some local computation and one shared memory *event*, which is an application of a synchronization primitive (such as *read*, *write*, or *read-modify-write*) to a base object.

An *execution* is a sequence of events that starts from an initial configuration, in which processes apply events and change states (based on the responses they receive from these events) according to their algorithm.

An *operation instance* is an application of a specific operation with specific arguments to a specific object made by a specific process. If the last event of an operation instance Φ has been applied in an execution, we say that Φ *completes in* E. We define by *completed(E)* the number of operation instances that complete in E. We say that a process p is *active* after execution E if p is in the middle of performing some operation instance Φ, i.e. p has applied at least one event while performing Φ in E, but Φ does not complete in E. If p is active after E, then it has exactly one *enabled* event, which is the next event p will apply.

An execution E is *k-synchronous* if the speeds of any two processes that participate in it vary by a factor of at most k. Formally, we say that E is k-synchronous if, for any two distinct processes p and q and for any execution $E = E_0 E_1 E'$, if p has an enabled event after E_0 and E_1 contains $k + 1$ events by q then E_1 contains at least one event by p.

In addition to *read* and *write*, our algorithm uses the *compare-and-swap* (CAS) and the and *double-compare-and-swap* (DCAS) primitives. (The algorithm can be implemented on systems that support only read, write and CAS by using a DCAS emulation from CAS. See, e.g., [9].)

CAS(w, old, new) writes the value *new* to memory location w only if its value equals *old* and, in this case, returns *true* to indicate success; otherwise it returns *false* and does not change the value of *w. DCAS* operates similarly on two memory locations.

The rest of the paper is organized as follows. Section 2 provides an overview of the algorithm. Section 3 describes the synchronous part of the algorithm in more detail. Section 4 describes the procedures that forward operation requests and dispatch responses. Section 5 introduces our throughput metric. Concluding remarks are brought in Section 6

2 An Overview of the BWC Algorithm

In this section, we provide a high-level description of the *bounded-wait combining* (henceforth, BWC) algorithm. It can be used to provide a linearizable, nonblock-

```
constant LEFT=0, RIGHT=1, FREE=⊥
structure Range {int from, int till}, typedef RRQ queue of Ranges
structure Reqs {int dir, int num}, typdef REQQ queue of Reqs
structure Node {
  Node* parent,                      Node* children[2],                  int reqs initially 0,
  int reqsTaken[2] initially {0,0},  REQQ pending initially EMPTY,       RRQ resp initially EMPTY,
  boolean inPhase initially false,   boolean collected initially false,  int slock initially FREE,
  int phaseTop initially null
}
Node* nodes[2N-1]
```

Fig. 1. The structure of BWC combining-tree nodes

ing and high-throughput implementation of any combinable operation [6,15]. For the sake of presentation simplicity, however, we describe it in the context of implementing a shared counter, supporting the *Fetch&increment* operation.

The algorithm uses a binary tree denoted $\mathcal{T}$. The value of the counter is stored at $\mathcal{T}$'s root. Each leaf of $\mathcal{T}$ is statically assigned to a single process. Every node of $\mathcal{T}$ consists of an instance of the *Node* structure shown in Figure 1, with the *parent* and *children* fields storing pointers to a node's parent and children, respectively. (Other node fields are described in the appropriate context.)

The BWC algorithm is parameterized: it uses an *asynchrony tolerance* parameter k that determines the extent to which processes are willing to wait for other processes. Each node contains a *synchronous lock*, which we simply call an *slock*. Whenever node n's *slock* equals the id of some process q, we say that n is *owned by* q; otherwise, we say that n is *free*. A leaf node is always owned by the process to which it is assigned. *Slocks are respected by processes in k-synchronous executions, but are disregarded in asynchronous executions* (i.e. in executions that are not k-synchronous).

For simplicity of pseudo-code presentation, we assume that variable scoping is dynamic, i.e. called procedures are in the scope of the calling procedure's local variables. The pseudo-code of the main procedure is shown in Figure 2. Initially, the algorithm behaves 'optimistically', in the sense that it operates under the assumption that the execution is k-synchronous. Whenever a process executes this part of the algorithm (implemented by the *SynchPhase* procedure, see Section 3), we say that it operates in a *synchronous mode*. As long as process speeds do not vary 'too much', processes operate in synchronous modes only and the algorithm guarantees low memory contention and high throughput.

While operating synchronously, computation proceeds in *synchronous phases*. In phase i, a subset of the participating processes construct a subtree $\mathcal{T}_i$ of $\mathcal{T}$, that we call a *phase subtree*. For every process q that participates in phase i, $\mathcal{T}_i$ contains all the nodes on the path from q's leaf node to the root.

Participating processes then use $\mathcal{T}_i$ as an ad-hoc combining tree. Each process q, participating in phase i, owns a path of nodes in $\mathcal{T}_i$ starting with q's leaf node and ending with the highest node along the path from q's leaf to the root whose *slock* q succeeded in acquiring. We denote q's path in $\mathcal{T}_i$ by $\mathcal{P}_i^q$.

After $\mathcal{T}_i$ is constructed, it is used as follows. Each participating process q starts by *injecting* a single new *operation request* at its leaf node. Processes then cooperatively perform the task of forwarding (and combining) these requests up $\mathcal{T}_i$. Process q is responsible for the task of forwarding requests from the sub-trees rooted at the nodes along $\mathcal{P}_i^q$. It may have to wait for other processes in order to complete this task. If the execution remains k-synchronous then, eventually, all collected requests arrive at the highest node of $\mathcal{P}_i^q$. If that node is not the root, then q now has to wait, as the task of forwarding these requests farther up $\mathcal{T}_i$ is now the responsibility of another process whose path ends higher in $\mathcal{T}_i$.

Finally, the operation requests of all participating processes arrive at the root. Once this occurs, the single process r whose path $\mathcal{P}_i^r$ contains the root increases the counter value stored at the root by the total number of requests collected. Process r then initiates the task of dispatching operation responses (which are natural numbers in the case of the *Fetch&Inc* operation) down $\mathcal{T}_i$. We call r the *phase initiator*. If the execution remains k-synchronous then eventually a response arrives at the leaf of each participating process.

A challenge in the design of the algorithm was that of achieving high throughput in k-synchronous executions while guaranteeing progress in *all* executions. To that end, the BWC algorithm is designed so that processes can identify situations where processes with which they interact are 'too slow' or 'too fast'. A detailed description of the *SynchPhase* procedure appears in Section 3.

After waiting for a while in vain for some event to occur, a process q may conclude that the execution is not k-synchronous. If and when that occurs, q falls back on an *asynchronous mode*. Once one or more processes start operating in asynchronous modes, high contention may result and high throughput can no longer be guaranteed but progress *is*.

The asynchronous part of the BWC algorithm is simple and its pseudo-code is omitted from this extended abstract for lack of space and can be found in the full paper. We now provide a short description. When process q shifts to an asynchronous mode, it injects an operation request at its leaf node, if it hadn't done so while operating synchronously. Process q then climbs up the tree from its leaf to the root. For every node n along this path, q forwards the requests of n's children to n. After it reaches the root, q descends down the same path to its leaf node, dispatching responses along the way. Process q *does not respect node* slock*s while traversing this path in both directions.* However, it releases the

```
Fetch&Inc()
boolean injected=false
int rc=SynchPhase()
if (rc $\neq$ ASYNCH)
    return rc
else
    return AsynchFAI()
```

Fig. 2. The main procedure of the BWC algorithm

slock of every node it descends from. (This guarantees that if the system reaches quiescence and then becomes semisynchronous, processes will once more operate in synchronous modes.)

Process q keeps going up and down this path until it finds a response in its leaf node. We prove that the BWC algorithm guarantees global progress even in asynchronous executions, hence it is nonblocking.

Actual operation-requests combining and response-propagation is performed by the *FwdReqs* and *SendResp* combining procedures (see Section 4). The BWC algorithm allows different processes to apply these procedures to different nodes of $\mathcal{T}$ concurrently. In asynchronous executions, it is also possible that multiple processes concurrently attempt to apply these procedures to the same node. E.g., multiple processes may concurrently attempt to apply the *FwdReqs* procedure to node n for combining the requests of n's children into n.

The correctness of the algorithm relies on verifying that each such procedure application either has no effect, or procedure statements are applied to n in order without intervening writes resulting from procedures applications to other nodes. Moreover, the data-sets accessed by procedures that are applied concurrently to different nodes are, in general, not disjoint: a procedure applied to node n may have to read fields stored at n's parent or children. To permit high throughput, these reads must be allowed *even if other processes apply their procedures to these nodes concurrently.* It follows that procedures applied to nodes cannot be implemented as transactions. The BWC algorithm satisfies these requirements through an infrastructure mechanism that we call *pseudo-transactions.* We now describe the pseudo-transactions mechanism in more detail.

2.1 Pseudo-transactions

Transactions either have no effect or take effect atomically. In contrast, concurrent reads *are* allowed while a pseudo-transaction executes but intervening writes are prohibited. Intuitively, pseudo-transactions suffice for the BWC algorithm because the information stored to node fields 'accumulates'. Thus reads that are concurrent with writes may provide partial data but they never provide inconsistent data. A formal definition of pseudo-transactions follows.

Definition 1. *We say that a procedure* P *is applied to an object n as a pseudo-transaction if each application of* P *either has no effect or the statements of* P *are applied to n in order and no field written by a statement of* P *is written by a concurrently executing procedure.*

The following requirements are met to ensure the correctness, liveness, and high-parallelism of the BWC algorithm:

1. **Combining correctness:** The combining procedures *FwdReqs* and *SendResp* are applied to nodes as pseudo-transactions.
2. **Node progress:** Progress is guaranteed at every node. In other words, after some finite number of statements in procedures applied to n are performed, some procedure applied to n terminates.

The BWC algorithm meets the above requirements by using the following two mechanisms. First, We treat each node (in conjunction with the *FwdReqs* and *SendResp* combining procedures) as a separate object. *We apply Greenwald's two-handed emulation to each of these objects separately* [7]. A detailed description of two-handed emulation is beyond the scope of this paper, and the reader is referred to [7]. We provide a short description of the emulation in the following.

Greenwald's two-handed emulation uses the *DCAS* operation to ensure that the statements of an applied procedure execute sequentially. To apply a procedure to object n, a process first tries to register a procedure-code and procedure operands at a designated field of n by using *DCAS*. Then, the process tries to perform the read and write operations of the procedure one after the other. Each write to a field of n uses *DCAS* to achieve the following goals: (1) verify that the write has not yet been performed by another process, (2) increment a virtual "program counter" in case the write can be performed, and (3) perform the write operation itself.

In addition to using two-handed emulation, we have carefully designed the node structure so that applications of the *FwdReqs* or *SendResp* combining procedures to different nodes never write to the same field (see Section 4 for more details).

Two-handed emulation guarantees that a combining procedure applied to node n either has no effect (if its registration failed) or its statements are performed with no intervention from other procedures applied to n. As procedures applied to different nodes never write to the same field, combining procedures are applied as pseudo-transactions and requirement 1. above is satisfied.

Applying two-handed emulation to an object n results in a nonblocking implementation, *on condition that procedures applied to other nodes cannot fail DCAS operations performed by a procedure applied to n.* Since procedures applied to different nodes never write to the same field, none of them can fail the other. Thus, Requirement 2. is also satisfied.

3 The Synchronous Modus Operandi

This part of the algorithm is implemented by the *SynchPhase* procedure (see pseudo-code in Figure 3) which iteratively switches over the *mode* local variable. Variable *mode* stores a code representing the current synchronous mode.

In the following description, q is the process that performs *SynchPhase*. The local variable n stores a pointer to the *current node*, i.e. the node currently served by q. Initially, n points to q's leaf. In some of the modes (*UP*, *FORWARD_REQUESTS* and *AWAIT_RESP*), q may have to wait for other processes. Before shifting to any of these modes, the local variable *timer* is initialized to the number of iterations q should wait before it falls back to an asynchronous mode. In the *ROOT_WAIT* mode, q waits so that other processes can join the phase it is about to initiate. In all of these cases, *timer* is initialized to some appropriately selected function of k and $\log N$. In the specification of these waiting times, M denotes the maximum number of events applied by a process in a single iteration of the *SynchPhase* procedure which is clearly a constant number.

We prove in the full paper that this selection of waiting periods guarantees that no process shifts to an asynchronous mode in k-synchronous executions. We now describe the synchronous modes.

UP: This is q's initial mode. Process q starts from its leaf node and attempts to climb up the the path to the root in order to join a phase. To climb from a non-root node to its parent m, q first verifies that m is free (statement 15), in which case it performs a *CAS* operation to try and acquire m's *slock* (statement 16). If it succeeds, q sets its current node n to m and reiterates. If m is not free, q checks the *m.inPhase* flag (statement 18) which indicates whether or not m is part of the current phase's subtree. If it is set then q checks whether n was added to the phase subtree by m's owner (statement 19). It both conditions hold, then q managed to join the current phase and all the nodes along the path from its leaf to n will join that phase's subtree. In this case, q shifts to the *PHASE_FREEZE* mode and stores a pointer to its highest node along this path (statements 20, 21). Otherwise, if q acquired the root node *slock*, then q is about to be the initiator of the next phase. It stores a pointer to the root node, sets the number of iterations to wait at the root to $\Theta(\log N)$, and shifts to the *ROOT_WAIT* mode (statements 12 - 14). If none of the above conditions hold, q decrements *timer* and, if it expires, concludes that the execution is asynchronous and returns the *ASYNCH* code (statements 23, 24).

ROOT_WAIT: q waits in this mode for the iterations timer to expire in order to allow other processes to join the phase subtree. While waiting, q performs a predetermined number of steps, in each of which it applies a single shared memory event. Finally, q shifts to the *PHASE_FREEZE* mode (statements 25-28).

PHASE_FREEZE: in this mode, q freezes the nodes along its path from *phaseTop* to its leaf and the children of these nodes. Freezing a node adds it to the phase's subtree. To determined which of n's children is owned by it, q uses the *LEAF_DIR* macro that, given an internal node n and a process id, returns *LEFT* or *RIGHT* (statement 29). For every node n, if the child of n not owned by q is not free, q sets that child's *inPhase* flag, sets n's *inPhase* flag, and descends to n's child on the path back to its leaf (statements 31-33, 40). When q gets to the parent of its leaf, it injects a single request to its leaf, sets the iterations counter, sets a flag indicating that a request was injected, and shifts to the *FORWARD_REQUESTS* mode (statements 35-38).

FORWARD_REQUESTS: in this mode, q forwards and combines requests along the path starting with the parent of its leaf and ending with *q.topPhase*. For each node n along this path, q checks for each child *ch* of n whether *ch* is in the current phase's subtree and whether requests need be forwarded from it (statement 42). If so and if *ch*'s *collected* flag is set, requests from the subtree rooted at *ch* were already forwarded and combined at *ch*. In this case q calls the *FwdReqs* procedure to forward these requests from *ch* to n and sets the local

```
SynchPhase()
1  Node* n=LEAF(myID), int phaseTop, int mode=UP, int timer=$Mk(k+13)\log N+1$
2  boolean chCollected[2]={false,false}
3  do forever
4    switch(mode)
5      case UP: UpMode()                     6  case ROOT_WAIT: RootWait()
7      case PHASE_FREEZE: PhaseFreeze()      8  case FORWARD_REQUESTS: ForwardRequests()
9      case AWAIT_RESP: AwaitResp()          10 case PROP_RESP: PropResp() od

UpMode()
11 if (ROOT(n))
12   phaseTop=n
13   timer=$2M(k+1)\log N$
14   mode=ROOT_WAIT
15 else if (n.parent.slock=FREE)
16   if (CAS(n.parent.slock, FREE, myID))
17     n=n.parent
18 else if (n.parent.inPhase)
19   if (n.inPhase)
20     phaseTop=n
21     mode=PHASE_FREEZE
22 else
23   timer=timer-1
24   if (timer=0) return ASYNCH

RootWait()
25 if (timer > 0)
26   read n.slock
27   timer=timer-1
28 else mode=PHASE_FREEZE

PhaseFreeze()
29 int whichChild=LEAF_DIR(n, myID)
30 Node *ch=n.children[whichChild]
31 if (n.children[1-whichChild].slock ≠ FREE)
32   n.children[1-whichChild].inPhase=true
33 n.inPhase=true
34 if (ch=LEAF(myID))
35   ch.reqs=ch.reqs+1
36   injected=true
37   timer=$M(k+1)\log N$
38   mode=FORWARD_REQUESTS
39 else
40   n=ch

ForwardRequests()
41 (for i=LEFT; i≤RIGHT; i++)
42   if (n.children[i].inPhase ∧¬ chCollected[i])
43     if (n.children[i].collected)
44       FwdReqs(n, i)
45       chCollected[i]=true
46     else
47       timer = timer -1
48       if (timer = 0) return ASYNCH
49       else continue do-forever (statement 3)
50 n.collected=true
51 if (n ≠ phaseTop)
52   n=n.parent
53 else if (ROOT(n))
54   mode=PROP_RESP
55 else
56   timer=$3Mk\log N$
57   mode=AWAIT_RESP

AwaitResp()
58 if (¬ EMPTY(n.resp))
59   mode=PROP_RESP
60 else
61   timer=timer-1
62   if (timer=0) return ASYNCH

PropResp()
63 if (n = LEAF(myID))
64   Range r=DEQ_R(resp)
65   if (RLEN(r) > 0) return r.first
66   else return ASYNCH
67 SendResp(n)
68 n.inPhase=false, n.collected=false, n.slock=FREE
69 n=n.children[LEAF_DIR(n,myID)]
```

Fig. 3. Pseudo-code for the synchronous part of the algorithm with asynchrony tolerance *k*

flag *chCollected* corresponding to *ch* in order to not repeat this work (statements 43-45). If *ch*'s *collected* flag is not set, *q* decrements *timer* and continues to wait for that event to occur; if the timer expires, *q* returns the *ASYNCH* code (statements 47 - 49). If and when *q* succeeds in forwarding requests from each of *n*'s children that is in the phase, it sets *n*'s *collected* flag and climbs up. Eventually, it shifts to either the *PROP_RESP* mode or the *AWAIT_RESP* mode, depending on whether or not it is the current phase's initiator (statements 50-57).

AWAIT_RESP: In this mode, *q*, when not the initiator of the current phase, awaits for the owner of node *n*'s parent to deliver responses to *n*. If and when this event occurs, *q* shifts to the *PROP_RESP* mode (statements 58, 59). If *timer* expires, *q* returns the *ASYNCH* code (statement 62).

PROP_RESP: In this mode, q propagates responses along the path from *q.phaseTop* down to its leaf node. For each node n along this path, q propagates n's responses to its children and then frees n (statements 67-69). Eventually, q descends to its leaf. If a response awaits there, it is returned as the response of *SynchPhase*. Otherwise, q returns $ASYNCH$ (statements 63-66).

4 The Combining Process

The combining process is implemented by the *FwdReqs* and *SendResp* procedures. The pseudo-code of these procedures appears in Figure 4. As discussed in Section 2.1, the code actually performed is a transformation of the code shown in Figure 4 according to Greenwald's two-handed emulation technique, as indicated by the *2he* attribute. The *FwdReqs* procedure forwards requests from a child node to its parent. The *SendResp* procedure dispatches responses from a parent node to its children. The two procedures use the following node fields.

reqs: For a leaf node n, this is the number of requests injected to n. If n is an internal node other than the root, this is the number of requests forwarded to n. If n is the root, this is the current value of the counter.

reqsTaken: This is an array of size 2. For each child m of n, it stores the number of requests forwarded from m to n.

pending: This is a queue of *Reqs* structures. Each such structure consists of a pair: a number of requests and the direction from which they came. This queue allows a process serving node n to send responses in the order in which the corresponding requests were received. This allows maintaining the linearizability of the algorithm. In k-synchronous executions, the *pending* queue contains at most 2 entries. In asynchronous executions it contains at most n entries, as there are never more than n simultaneous operations applied to the counter.

resp: This is a producer/consumer queue storing response ranges that were received at n and not yet sent to its children. The producing process (the one that serves the parent node) enqueues new response ranges and the consumer process (the one that serves the child node) dequeues response ranges. The producer and consumer processes never write to the same field simultaneously. The *resp* queue contains at most a single range in k-synchronous executions. In asynchronous executions it contains at most n ranges. We now describe the pseudo-code of these two procedures that are performed as pseudo-transactions. Here, q is the process executing the code.

The *FwdReqs* procedure receives two parameters. A pointer n to the node to which requests need to be forwarded, and an integer, *dir*, storing either *LEFT* or *RIGHT*, indicating from which of n's children requests need to be forwarded to n. Let m denote the child node of n that is designated by the *dir* parameter. Process q first initializes a pointer to m (statement 1). Then q computes the number of requests in m that were not yet forwarded to n and stores the result in *delta* (statement 2). If there are such requests, q proceeds to forward them.

```
2he FwdReqs(NODE* n, int dir)
1  Node* child = n.children[dir]
2  int delta=child.reqs - n.reqsTaken[dir]
3  if (delta > 0)
4     n.reqs=n.reqs+delta
5     n.reqsTaken[dir]=n.reqsTaken[dir]+delta
6     ENQ_REQS(n.pending, <dir,delta>
7     if (ROOT(n))
8        ENQ_R(n.resp, n.reqs-delta+1, delta)
```

```
2he SendResp(NODE* n)
9  do twice
10    if (¬ EMPTY(n.resp))
11       Range fResp=FIRST_R(n.resp)
12       int respsLen=RLEN(fResp)
13       RRQ fReqs=FIRST_REQS(n.pending)
14       int reqsLen=REQS_LEN(fReqs)
15       int send=min(respLen, reqsLen)
16       int dir=REQS_DIR(fReqs)
17       ENQ_R(n.children[dir].resp, fResp.start, send)
18       DEQ_R(n.resp, send)
19       DEQ_REQS(n.pending, send)
20 od
```

Fig. 4. Pseudo-code for the combining procedures

Forwarding the requests is implemented as follows. Process q first increases n's *reqs* field by *delta* (statement 4) to indicate that *delta* additional requests were forwarded to n. It then increases n's *reqsTaken* entry corresponding to m by *delta* (statement 5). This indicates that *delta* additional requests were forwarded from m to n. Finally, q adds an entry to n's *pending* queue specifying that *delta* more responses need to be sent from n to m. If n is the root node, then the operations represented by the forwarded requests are applied to the central counter at the root when *n.reqs* is increased by statement 4. In that case, q immediately adds the corresponding range of counter values to n's queue of response ranges (statements 7-8).

The *SendResp* procedure receives a single parameter - n - a pointer to the node from which responses need be sent. Process q executes the loop of statements 9-20 twice. If the responses queue is not empty, the length of its first range is computed and stored to the *respLen* local variable (statements 11, 12). Then the length of the next requests entry is computed and stored to *reqsLen* (statements 13, 14). The minimum of these two values is stored to *send* (statement 15). This number of responses is now sent in the direction specified by the *dir* field of the first requests entry (statements 16, 17). Finally, *send* responses and requests are removed from the *n.resp* and *n.pending* queues, respectively. The loop of statements 9-20 is executed twice. This is enough to propagate the responses for all the requests forwarded to a node in a synchronous phase.

For presentation simplicity, the *reqs* and *reqsTaken* fields used in the pseudo-code of Figure 4 are unbounded counters. To ensure the correctness of the combining process, however, it is only required that the difference between the values of these fields be maintained. This difference cannot exceed N, since this is the maximum number of concurrent operations on the counter. It follows that the code can be modified to use bounded fields that count modulo $2N + 1$.

5 A Metric for the Throughput of Concurrent Objects

We now present a metric for quantifying the throughput of concurrent objects. This metric allows us to reason about the throughput of concurrent objects rigorously. The key to the metric is a definition of time that assigns identical times to events that may be executed concurrently.

Let E be an execution. Consider a subsequence of events of E applied by some process p. As processes are sequential threads of execution, the times assigned to the events applied by p must be strictly increasing. Consider a subsequence of events in E, all of which access the same base object o. Here it is less obvious how to assign times to these events. If the value of o is never cached, then the times assigned to these events must be strictly increasing as accesses of o are necessarily serialized. However, if o may be cached, then it *is* possible for o to be read concurrently by multiple processes accessing their local caches. Thus an alternative assignment of times, where consecutive reads of the same base object may be assigned identical times, is also possible.

For the analysis of the BWC algorithm, we chose to apply the stricter definition where time must strictly increase between accesses of the same object. We define the throughput of an execution as the ratio between the execution's duration and the number of operation instances that complete in it. It follows that the throughput of an algorithm can only increase if the alternative (less strict) assignment of times were to be used. Formal definitions of execution time and throughput follow.

The subsequence of events applied by process p in E is denoted by $E|p$. The subsequence of events that access a base object o in E is denoted by $E|o$. We assign times to execution events. Assigned times constitute a non-decreasing sequence of integers starting from 0. Times assigned to the subsequence of events applied by any specific process p are monotonically increasing. Similarly, times assigned to the subsequence of events that access any specific object o are monotonically increasing. This is formalized by the following definitions.

Definition 2. *Let e be an event applied by process p that accesses base object o in execution E and let $E = E_1eE_2$. The* synchronous time *assigned to e in E, denoted by* time(E,e)*, is defined to be the maximum of the following numbers:*

– *the synchronous time assigned to the last event of E_1 (or 0 if E_1 is empty),*
– *the synchronous time assigned to p's last event in E_1 plus 1 (or 0 if $E_1|p$ is empty),*
– *the synchronous time assigned to the last event in E_1 that accesses o plus 1 (or 0 if $E_1|o$ is empty).*

Definition 3. *The* synchronous duration *of an execution E, denoted by* time(E)*, is 0 if E is the empty execution or $time(E, e_l) + 1$ otherwise, where e_l is the last event of E. The* throughput of a *(non-empty)* execution *E is defined to be* completed(E) / time(E)*.*

Based on the above definitions, we prove the following theorem.

Theorem 1. *The throughput of the BWC algorithm with asynchrony tolerance parameter k in k-synchronous executions in which all processes start their operations concurrently is $\Omega(N/\log N)$.*

To prove theorem 2, we first show that the selection of the waiting times with which the *timer* variable is set guarantees that processes always operate in synchronous modes in k-synchronous executions. We then show that as long as all processes operate in synchronous modes and the execution is k-synchronous, every operation instance completes in $O(\log N)$ time. The proof is omitted from this extended abstract for lack of space and can be found in the full paper.

We also prove the following theorem in the full paper.

Theorem 2. *The BWC algorithm is a nonblocking linearizable counter implementation.*

Intuitively, the linearizability of the BWC algorithm follows from the fact that the counter value is stored at the root node and that responses are dispatched from each node n in the order in which the corresponding requests were received at n. We show that, as long as all processes operate in synchronous modes, the BWC algorithm is wait-free. If some processes fall back on asynchronous modus operandi, then we show that the **Node progress** property (see Section 2.1) guarantees overall progress.

6 Concluding Remarks

In this paper we present Bounded-Wait Combining (BWC), the first nonblocking linearizable counter algorithm that can provably achieve high parallelism in semisynchronous executions. Bounded-Wait Combining can be used to obtained implementations, possessing the same properties, of objects such as stack and queue. We define a novel metric of the throughput of concurrent algorithms and use it to analyze our algorithm. We also introduce and use *pseudo-transactions* - a concurrent execution technique that, though weaker than ordinary transactions, permits higher parallelism. We believe that both our throughput metric and the pseudo-transactions construct may prove useful in the design and analysis of other algorithms.

Our algorithm guarantees the nonblocking property in *all* executions. However, to guarantee high throughput, it is required that processes know in advance an upper bound on the ratio between the fastest and slowest processes. (The requirement for knowledge of timing information is similar to that made in the *known bound* model [1,16].) We believe the BWC algorithm can be extended to adjust adaptively to an unknown system bound on the speed ratio. Also, the tree used by the BWC algorithm is static and of height $\Theta(\log N)$, regardless of the number of processes participating in the computation. It follows that the time it takes to apply an operation is $\Theta(\log N)$ even if a process runs solo. It would be interesting to see whether the algorithm can be made 'dynamic' from this respect while maintaining its properties. We leave these research problems for future work.

Acknowledgements. We thank the anonymous reviewers for many useful comments.

References

1. R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. Comput.*, 26(2):539–556, 1997.
2. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.
3. H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.
4. F. E. Fich, D. Hendler, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *FOCS 2005*, pages 165–173.
5. J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS*, pages 64–75, 1989.
6. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer designing a mimd, shared-memory parallel machine. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 239–254, New York, NY, USA, 1998. ACM Press.
7. M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *PODC 2002*, pages 260–269, 2002.
8. P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-tuning reactive distributed trees for counting and balancing.
9. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC 2002*.
10. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04*, pages 206–215, New York, NY, USA. ACM Press.
11. M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.
12. M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS 2003*, pages 522–529.
13. M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.
14. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, June 1990.
15. C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM TOPLAS*, 10(4):579–601, 1988.
16. N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1992.
17. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA'05*, pages 253–262, New York, NY, USA.
18. N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.
19. P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, 1987.

Byzantine and Multi-writer K-Quorums*

Amitanand S. Aiyer[1], Lorenzo Alvisi[1], and Rida A. Bazzi[2]

[1] Department of Computer Sciences,
The University of Texas at Austin
{anand, lorenzo}@cs.utexas.edu
[2] Computer Science and Engineering Department,
Arizona State University
bazzi@asu.edu

Abstract. Single-writer k-quorum protocols achieve high availability without incurring the risk of read operations returning arbitrarily stale values: in particular, they guarantee that, even in the presence of an adversarial scheduler, any read operation will return the value written by one of the last k writes. In this paper, we expand our understanding of k-quorums in two directions: first, we present a single-writer k-quorum protocol that tolerates Byzantine server failures; second, we extend the single-writer k-quorum protocol to a multi-writer solution that applies to both the benign and Byzantine cases. For a system with m writers, we prove a lower bound of $\big((2m-1)(k-1)+1\big)$ on the staleness of any multi-writer protocol built over a single-writer k-quorum system and propose a multi-writer protocol that provides an almost matching staleness bound of $\big((2m-1)(k-1)+m\big)$.

1 Introduction

Quorum systems have been extensively studied, with applications that include mutual exclusion, coordination, and data replication in distributed systems [1,2,3,4]. systems [1,2,3,4]. A traditional, or strict, quorum system is simply a collection of servers organized in sets called quorums. Quorums are accessed either to write a new value to a *write quorum* or to read the values stored in a *read quorum*: in strict quorums, any read quorum intersects with a write quorum.

Important quality measures of quorum systems are *availability*, *fault tolerance*, *load*, and *quorum size*. lower size have measures are conflicting in strict quorum systems [5]. For instance, the majority quorum system provides the highest availability of all strict quorum systems when the failure probability of individual nodes is lower than 0.5, but it also suffers from high load and large quorum size—and this tension holds true in general [6]. When the failure probability of individual nodes is higher than 0.5, the quorum system with highest availability is the singleton, in which one node handles all requests in the system.

Probabilistic [7] and signed [8] quorum systems have been proposed to achieve high availability while guaranteeing system consistency (non-empty intersection of

* This work was supported in part by NSF Cybertrust award 0430510, NSF award CNS 0509338, and a grant from the Texas Advanced Technology Program.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 443–458, 2006.

quorums) with high probability. These probabilistic constructions offer much better availability than the majority system at the cost of providing only probabilistic guarantees on quorum intersection. If a probabilistic quorum system is used to implement a distributed register with read and write operations, then, with high probability, a read operation will return the value most recently written.

To achieve a high probability of quorum intersection, probabilistic constructions assume, either implicitly (probabilistic quorum systems [7]) or explicitly (signed quorum systems [8]), that the network scheduler is not adversarial. If the scheduler is adversarial, both constructions can return arbitrarily old values, even if servers fail only by crashing. If instead servers can also be subject to Byzantine failures, the situation is a bit more complicated. Signed quorum systems are simply not defined under these circumstances; probabilistic Byzantine quorum systems [7] must instead be configured to prevent read operations from returning values fabricated by Byzantine servers. Note that returning a fabricated value can be much more problematic than returning an arbitrarily old value, especially if readers are required to write back what they read (as it is common to achieve strong consistency guarantees): in this case, the system can become *contaminated* and quickly loose its consistency guarantees[1]. Fortunately, the parameters of probabilistic quorums systems can be chosen to eliminate the possibility of contamination; unfortunately, doing so results in a loss of all the gains made in availability.

k-quorum systems, which we have recently introduced [9], guarantee that a read operation will always return one of the last k written values – even if the scheduler is adversarial. If the scheduler is not adversarial and read quorums are chosen randomly, as is the case with probabilistic systems, k-quorums can guarantee a high probability of intersection with the quorum used by the latest write. In a sense, k-quorums have some of the best features of both strict systems and probabilistic constructions and they can be thought of as a middle ground between them. Like probabilistic constructions, they achieve high availability by performing their writes to small quorums, (called *partial-write-quorums*), and therefore weaken the intersection property of traditional strict quorum systems; unlike probabilistic constructions, however, k-quorums can still provide *deterministic* intersection guarantees: in particular, they require the set of servers contacted during k consecutive writes—the union of the corresponding partial write quorums—to form a traditional strict write quorum. Using this combination, k-quorum systems can bound the staleness of the value returned by a read, even in the presence of an adversarial scheduler: a read operations that contacts a random read quorum of servers is guaranteed to return one of the values written by the last k preceding writes; furthermore, during periods of synchrony the returned value will, with high probability, be the one written by the last preceding write.

In the absence of an adversarial scheduler, probabilistic systems can have higher availability than k-quorum systems. k-quorums make a tradeoff between safety and liveness. By allowing for lower availability than probabilistic systems, they guaran-

[1] This is not a problem if the returned values are simply old values because in that case timestamps can be used to prevent old values from overwriting newer values. Timestamps cannot be used with fabricated values because the timestamps of the fabricated values can themselves be fabricated.

tee a bound on the staleness of returned values even in the presence of an adversarial scheduler. In the absence of an adversarial scheduler, k-quorum systems have higher availability than strict quorum systems when the frequency of write operations is not high (in a sense well defined in [9]). In the same paper, we also propose k-consistency semantics and provide a single-writer implementation of k-atomic registers over servers subject to crash failures.

Our previous paper left several important questions unanswered—in particular, it did not discuss how to handle Byzantine failures, nor how to provide a multi-writer/multi-reader construction with k-atomic semantics. The first question is particularly important in light of the contamination problem that can affect probabilistic Byzantine quorum systems. Answering these questions is harder than in strict quorum systems because the basic guarantees provided by k-quorum systems are relatively weak and hard to leverage. For example, in the presence of multiple writers it is hard for any single writer to guarantee that k consecutive writes (possibly performed by other writers) will constitute a quorum: because of the weaker k consistency semantics, a writer cannot accurately determine the set of servers to which the other writers are writing.

In this paper, we answer both questions. We begin by showing a protocol that implements single-writer k-atomic semantics and tolerates f Byzantine servers and any number of crash-and-recover failures as long as read and write quorums intersect in at least $3f + 1$ servers. Like its crash-only counterpart, the protocol can provide better availability than strict quorum systems when writes are infrequent, and unlike probabilistic solutions, can bound the staleness of the values returned by read operations. Byzantine faults add another dimension to the comparison with probabilistic solutions: the cost, in terms of loss of availability, of preventing reads from returning a value that has never been written by a client, but has instead been generated by Byzantine servers out of thin air. We show that, for equally sized quorums, this cost is considerably higher for probabilistic constructions than for k-quorum systems.

We then investigate the question of k-atomic semantics in a multi-writer/multi-reader setting by asking whether it is possible to obtain a multi-writer solution by using a single writer solution as a building block—that is, by restricting read and write operations in the multi-writer case to use the read and write partial quorums of the single writer solution. This approach appears attractive, because, if successful, would result in a multi-writer system with availability very close to that of a single writer system.

We first show a lower bound on the price that any such system must pay in terms of consistency: we prove that no m-writer protocol based on a solution that achieves k-atomic semantics in the single writer case can provide better than $\big((2m-1)(k-1)+1\big)$-atomic semantics. We then present an m-writer protocol that provides $\big((2m-1)(k-1)+m\big)$-atomic semantics, using a construction that, through a clever use of vector timestamps, allows readers and writers to disregard excessively old values.

2 System Model

We consider a system of n *servers*. Each server (or node) can crash and recover. We assume that servers have access to a stable storage mechanism that is persistent across crashes. We place no bound on the number of non-Byzantine failures and, when considering Byzantine faults, we assume that there are no more than f Byzantine servers—all remaining servers can crash and recover.

Network Model. We consider an asynchronous network model that may indefinitely delay, or drop, messages. We require that the protocols provide staleness guarantees irrespective of network behavior.

For purposes of availability, we assume there will be periods of synchrony, during which, if enough servers are available, operations execute in a timely manner.

Access Model. A read or write operation needs to access a read or a (partial) write quorum in order to terminate successfully. If no quorum is available the operation has two options: it can either abort or remain pending until enough servers become available (not necessarily all at the same time). The operation can abort unless it has already taken actions that can potentially become visible to other clients.

Clients operations may have timeliness constraints. This does not contradict the asynchrony assumption we make about the network but simply reflects the expectation that operations should execute in a timely manner if the system is to be considered available. A client considers any operation that does not complete in time to have *failed*, independent of whether these operations abort or eventually complete. Note that an operation may be aborted and fail before being actually executed if the operation remains locally queued for too long after being issued.

We assume for simplicity that clients do not crash in between operations, although our protocols can be extended to tolerate client crash and recovery by incorporating a logging protocol.

Finally, we assume that writes are blocking. In other words, a writer will not start the next write until the current write has finished. While this assumption is not overly restrictive, we need to make it for a technical reason, as our protocols require a write operation to know exactly where the previously written values have been written to.

Availability. Informally, a system is *available* at time t if operations started at t execute in a timely manner. Consider an execution ρ in a given time interval (possibly infinite) in which a number of operations are started. The system's *availability for execution* ρ is the ratio of the number of operations that complete in a timely manner in ρ to the total number of operations in ρ. If the number of operations is infinite, then the system's availability is the limit of the ratio, if it exists.

The *read and write access patterns* are mappings from the natural numbers to the set of positive real numbers (denoting the duration between the requests). The *failure pattern* of a given node is a mapping from the positive real numbers (denoting global time) to $\{up, down\}$; the system's failure pattern is a set of failure patterns, one for each node.

Given probability distributions on the access patterns (read or write) and failure patterns, the system's *availability* is the expected availability for all pairs of access patterns and failure patterns.

For the purposes of estimating availability, we assume that nodes crash and recover independently, with mean time to recover (MTTR) α and mean time between failures (MTBF) β. We also assume the periods between two consecutive reads or writes to be random variables with means MTBR and MTBW respectively and that MTBW is large compared to MTBF; in other words, writes are infrequent. We define the system's availability in periods in which the network is responsive; i.e. in periods in which the roundtrip delay is negligible compared to MTBF and MTTR. In other words, the availability we are interested in depends on whether nodes are up or down, and not on how slow is the network: indeed, in the presence of an adversarial network scheduler measuring availability becomes meaningless, since the scheduler could always cause it to be equal to zero. We assume that the time allowed for successful completion of an operation is negligible compared to MTBF and MTTR.

Relaxed Consistency Semantics. The semantics of shared objects that are implemented with quorum systems can be classified as *safe*, *regular* or *atomic* [10]. For applications that can tolerate some staleness, these notions of consistency are too strong and one can use define relaxed consistency semantics as follows [9]:

1. k**-safe:** A read that does not overlap with a write returns the result of one of the latest k completed writes. The result of a read overlapping a write is unspecified.
2. k**-regular:** A read that does not overlap with a write returns the result of one of the latest k completed writes. A read that overlaps with a write returns either the result of one of the latest k completed writes or the eventual result of one of the overlapping writes.
3. k**-atomic:** A read operation returns one of the values written by the last k preceding writes in an order consistent with real time (assuming there are k initial writes with the same initial value).

3 K-quorums for Byzantine Faults

We define a k-quorum construction that tolerates f Byzantine servers, while providing k-atomic semantics, as a triple $\big(\mathcal{W}, \mathcal{R}, k\big)$, where $\mathcal{W}$ is the set of write quorums, $\mathcal{R}$ is the set of read quorums, and k is a staleness parameter such that, for any $R \in \mathcal{R}$, and $W \in \mathcal{W}$, $|R \cap W| \geq 3f + 1$ and $|R|, |W| \leq (n - f)$.

Server side protocol Figure 1 shows the server-side protocol. Each server s maintains in the structure *current_data* information about the last write the server knows of, as well as the $k - 1$ writes that preceded it. READ_REQUEST messages are handled using a "listeners" pattern [11]. The sender is added to s's *Reading* set, which contains the identities of the clients with active read operations at s. A read operation r is active at s from when s receives r's READ_REQUEST to when it receives the corresponding STOP_READ. On receipt of a WRITE message, s acknowledges the writer. Then, if the received information is more recent than the one stored in *current_data*, s updates *current_data* and forwards the update to all the clients in *Reading*; otherwise, it does nothing.

Writer's Protocol. Figure 2 shows the client-side write protocol. Each write operation affects only a small set of servers, called a *partial write quorum*, chosen by the writer so that the set of its last k partial write quorums forms a complete write quorum. The information sent to the servers contains not just a new value and timestamp, but also additional data that will help readers distinguish legitimate updates from values fabricated by Byzantine servers. Specifically, the writer sends to each server in the partial write quorum, k tuples—one for each of its last k writes. The tuple for the i-th of these writes includes: i) the value v_i; ii) the corresponding timestamp ts_i; iii) the set E_i of servers that were not written to in the last $k-1$ writes preceding i; and iv) a hash of the tuples of the $k-1$ writes preceding i. The write ends once the set of servers from which the writer has received an acknowledgment during the last k writes forms a complete write quorum[2].

Thus, the value, timestamp, E, and hash information for write i are not only written to i's partial write quorum, but will also be written to the partial write quorums used for the next $k-1$ writes. By the end of these k writes this information will be written to a complete write quorum which is guaranteed to intersect any read quorum in at least $3f+1$ servers.

Reader's Protocol. The reader contacts a read quorum of servers and collects from each of them the k tuples they are storing. The goal of the read operation is twofold: first, to identify a tuple t_i representing one of the last k writes, call it i, and return to the reader the corresponding value v_i; second, to write back to an appropriate partial write quorum (one comprised of servers not in E_i) both t_i and the $k-1$ tuples representing the writes that preceded i—this second step is necessary to achieve k-atomicity.

The read protocol computes three sets based on the received tuples. The *Valid* set contains, of the most recent tuples returned by each server in the read quorum, only those that are also returned by at least f other servers. The tuples in this set are legitimate: they cannot have been fabricated by Byzantine servers.

The *Consistent* set also contains a subset of the most recent tuples returned by each server s in the read quorum. For each tuple t_s in this set, the reader has verified that the hash of the $k-1$ preceding tuples returned by s is equal to the value of h stored in t_s.

The *Fresh* set contains the $2f+1$ most recent tuples that come from distinct servers. Since a complete write quorum intersects a read quorum in at least $2f+1$ correct servers, legitimate tuples in this set can only correspond to recent (i.e. not older than k latest) writes.

The intersection of these three sets includes only legitimate and recent tuples that can be safely written back, together with the $k-1$ tuples that precede them, to any appropriate partial write quorum. The reader can choose any of the tuples in this intersection: to minimize staleness, it is convenient to choose the one with the highest timestamp.

Because of space limitations, we must refer the reader to our extended technical report [12] for the proofs of the following theorems.

Theorem 1. *The single-writer Byzantine k-quorum read protocol in Figure 3 never returns a value that has not been written by the writer.*

Theorem 2. *The single-writer Byzantine k-quorum read protocol in Figure 3 never returns a value that is more than k-writes old.*

If the network is behaving asynchronously, or if the required number of servers is not available, then our protocols will just stall until the systems comes to a good con-

[2] Byzantine servers may never respond. The writer can address this problem by simply contacting f extra nodes for each write while still only waiting for a partial quorum of replies. For simplicity, we abstract from these details in giving the protocol's pseudocode.

```
static Reading = ∅
static current_data[1..k];
while( true ) {
  (msg, sender) = recieveMessage();

  if( msg instanceof READ_REQUEST)
    Reading ∪ = {        };
    send current_data to sender.
  else if( msg instance of STOP_READ )
    Reading = Reading \ {        };
  else if( msg instance of WRITE )
    // say msg is WRITE
            ⟨     [      , . . . ,        −   + 1]⟩
    if(       .     current_data[1].ts )
      current_data[1..k] =
                   [      , . . . ,        −   + 1];
      send ACK(        ) to sender;
      forward current_data to all in Reading.
    else
      send ACK(        ) to sender;
}
```

Fig. 1. K-quorum protocol for non-Byzantine servers

```
static     := 0;
static Tuple[];
void Write(value v)
begin
      :=    + 1;
     = hash( Tuple[   − 1, . . . ,    −   + 1] );
    // E is the set of servers NOT used for the
          previous   − 1 writes
     =    \ ⋃ =  −1
              =  −  +1
    Tuple[  ] = ( ,  ,  ,  );
    delete Tuple[   −   ] to save space

    Find a set      , such that :
     |    ⋃ ∪ =  −1        | =
              =  −  +1
    send WRITE⟨      [  , . . . ,    −   + 1]⟩ to all
          servers in      .

    // wait for acknowledgements
        = ∅
    do
      recv       (  ) from serv
          =      ∪ {     }
    until ( | ∪ =          | ≥      −    )
                =  −  +1
    return
end
```

Fig. 2. K-quorum write protocol tolerating up to f Byzantine servers

figuration. If, during periods of synchrony, all non-Byzantine nodes recover and stay accessible, then our protocols eventually terminate.

Theorem 3. *If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine k-quorum protocol for the writer in Figure 2 eventually terminates.*

Theorem 4. *If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine k-quorum protocol for the reader in Figure 3 eventually terminates.*

Theorem 5. *The construction for Byzantine k-quorum systems shown in Figures 1, 2, 3 provides k-atomic semantics.*

3.1 Comparison to Probabilistic Quorum Systems

In the Byzantine version of probabilistic quorum systems—(f, ϵ)-masking quorum systems [7]—write operations remain virtually unchanged: values are simply written to a write quorum chosen according to a given access strategy. Read operations contact a read quorum, also chosen according to the access strategy, and return the highest timestamped value that is reported by more than p servers, where p is a safety parameter[3]. Choosing any value of p lower than $f + 1$ can be hazardous as, under these circumstances, read operations may return a value that was never written by a client, but instead fabricated by Byzantine nodes. While the probability of an individual read

[3] The original paper [7] uses k to denote this safety parameter. We use p to avoid confusion with the staleness parameter of k-quorum systems. We also use f to denote the threshold on Byzantine faults instead of the original b.

```
// protocol for a reader
received[] // stores the responses from servers
CandidateValues // holds the set of candidate values
Read()
begin
  choose a read quorum R.
  send READ_REQUEST to servers in R.

  received[i] = null, 1 ≤   ≤ |  |
  CandidateValues = ∅
  // receive values from all the servers in R
  while ( |{ :        [ ] ≠      }|   |  | );
  begin
    receive Tuple[   , . . . ,     −   + 1] from server s;
    received[s] = Tuple[   , . . . ,     −   + 1];
    if( isValid( Tuple[   , . . . ,     −   + 1] ) )
      add Tuple[   ] to the set CandidateValues
  end

  // try to choose a value
  // if unsuccessful, wait for more responses.
            = LargestTimestamp( received );
  tryChoosing( );
  while( value_chosen == null )
  begin
    receive Tuple[   , . . . ,     −   + 1] from server s;
    if (    ≤          )
      received[s] = Tuple[   , . . . ,     −   + 1];
      tryChoosing( );
  end

  send STOP_READ to servers in R.

  // write back the chosen value to a partial−write−quorum
  Find a partial−write−quorum, PW, suitable for value_chosen.
    send WRITE⟨          ⟩ to PW
   − wait for acks from PW

  return value_chosen
end

void tryChoosing( )
begin
  (1) Fresh = { Tuple[   , . . . ,     −   ] ∈        |      is one of the 2f+1 largest time−stamped
        entries in        received from different servers }
  (2) Valid = { Tuple[   , . . . ,     −   ] ∈        | Tuple[   ] occurs in the responses of at least
          + 1 servers }
  (3) Consistent = { Tuple[   , . . . ,     −   ] ∈        | the hash, h, in Tuple[   ] matches
        hash( Tuple[     − 1, . . . ,     −   ] ) }
  (4) if (     ∩      ∩         ≠ ∅ )
        value_chosen =   ∈     ∩      ∩        , with the largest timestamp.
end
```

Fig. 3. K-quorum read protocol tolerating up to f Byzantine servers

operation returning a fabricated value can be low, if enough reads occur in the system, the probability that one of them will do so becomes significant, even in the absence of an adversarial scheduler. Byzantine k-quorums are immune from such dangers: read operations may return slightly stale values, but never fabricated values. This property allows for the safe use of write backs to achieve stronger consistency guarantees.

Availability. Although it is possible to tune probabilistic Byzantine quorum systems by choosing $p > f$ so that they never return fabricated values, such a choice of p cannot guarantee that the read availability always increases with n: if $p > \frac{q^2}{n}$, then read availability actually tends to 0 as n increases, because even a reader able to contact a read-quorum is highly unlikely to receive at least p identical responses [7]. To ensure that, with high probability, there are at least $f + 1$ identical responses in a read quorum, probabilistic Byzantine quorum systems would have to choose large quorum sets—requiring the size of the quorum q to be significantly larger than $\sqrt{nf}$. Thus, if the number of Byzantine failures f is large, then the quorum size for probabilistic quorum systems needs to be large in order to avoid fabricated values.

In summary, if probabilistic Byzantine systems are to have high availability when the scheduler is not adversarial, they run the risk of returning fabricated values, and if a value that is dependent on a fabricated value is written to the system, the system becomes contaminated. Also, if they are designed for high availability and the scheduler happens to be adversarial, probabilistic Byzantine systems can always be forced to return fabricated values.

Our system provides high availability for both reads and writes while guaranteeing that we always return one of the latest k values written to the system. There are two main reasons for the higher availability of k-quorums. First, each of their write operations also writes tuples for the preceding $k-1$ writes, causing a write to become visible at more locations than in a probabilistic quorum system with similar quorum sizes and load. Second, k-quorums reads are content to return one of the last k writes, not just the latest one. Read operations will therefore be likely to yield *Valid*, *Consistent*, and *Fresh* sets with a non-empty intersection. In (f, ϵ)-masking quorums a read can return a legitimate value only if the read quorum intersects with a single write-quorum in more than p nodes. This is a much rarer case and the availability of probabilistic quorum systems is consequently lower.

Probability of returning the latest value. The definition of k-atomicity only bounds the worst-case staleness of a read. However, since the choice of read quorums is not dependent on any other quorums chosen earlier, k-quorums can also use a random access strategy to choose read quorums, as in [7]. A random access strategy guarantees that, when the network is not adversarial, a read which does not overlap with a write returns, with high probability, the latest written value.

Let r and w_p denote, respectively, the sizes of the read quorum and of the partial-write-quorums. We can use Chernoff bounds in a manner similar to [7] to establish the following theorem, whose proof is contained in our extended technical report [12].

Theorem 6. *If the read quorum is chosen uniformly at random, then at times when the network is non-adversarial, the probability that a read does not return the latest written value is at most* $e^{-\frac{(\ -\ )}{2}(1-\frac{(\ +1)}{(\ -\ )})^2}$.

This probability can be high if f is small relative to n.

4 Multi Writer k-Quorums

We now study the problem of building a multi-writer k-quorum system using single-writer k-quorum systems. This problem is interesting because the resulting multi-writer system will have almost the same availability as the underlying single-writer systems.

A single-writer multi-reader k-quorum system implements two operations.

1. val sw-kread(wtr): returns one of the k latest written values, by the writer wtr.
2. sw-kwrite(wtr, val): writes the value val to the k-quorum system. It can only be invoked by the writer wtr

We assume that the read and write availability of the single-writer k-quorum system is $a_{sr} = 1 - \epsilon_{sr}$ and $a_{sw} = 1 - \epsilon_{sw}$ respectively.

4.1 A Lower Bound

We show that using k-atomic single-writer systems as primitives for a multi-writer system with m writers, one cannot achieve more than $\big((2m-1)(k-1)+1\big)$-atomic guarantees.

We assume that the the multi-writer solution uses the single writer solution through the *sw-kread* and *sw-kwrite* functions. We use these functions as black boxes, and we assume that an invocation of *sw-kread* on a given register will return any one of the last k writes to that register.

Since we are interested in a multi-writer solution that has the same availability as the underlying single writer system, we should rule out solutions that require a write in the multi-writer system to invoke multiple write operations of the single writer system. In other words, a write operation in the multi-writer system should be able to successfully terminate if a read quorum and a partial write quorum of the single writer system are available. We require that a read quorum be available because otherwise writers would be forced to write independently of each other with no possibility for one writer to *see* other writes. We do not require that a read and a write quorum be available at the same time. So, without loss of generality, we assume that the implementation uses only m single-writer registers, one for each writer. The implementation of a write operation of a the multi-writer register can issue a write operation to the issuing writer's register but not to the other writers' registers; it can also issue read operations to any of the m registers. The read operations on the multi-writer register can only issue read operations on the single-writer registers.

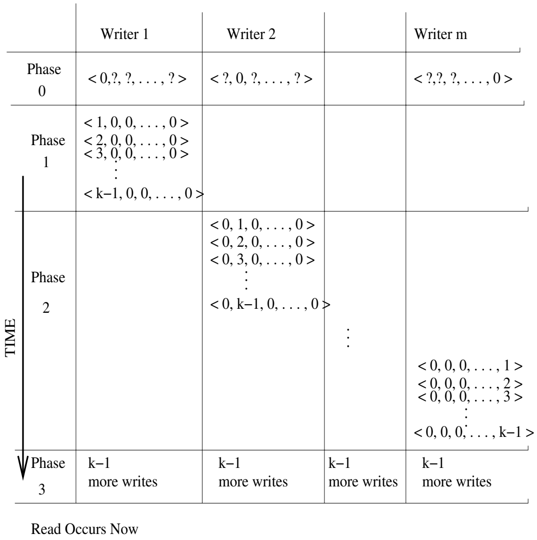
In our lower bound proof, we assume that writers execute a full-information protocol in which every write includes all the history of the writer, including all the values it ever wrote and all the values it read from other writers. If the lower bound applies to a full-information protocol, then it will definitely apply to any other protocol, because a full-information protocol can simulate any other protocol by ignoring portions of the data read. Also, we assume that a reader and a writer read all single-reader registers in every operation, possibly multiple times; a protocol that does not read some registers can simply ignore the results of such read operations.

For a writer wtr, we denote with $v_{wtr,i}$ the i'th value written by wtr. If a client reads $v_{x,i}$, then it will also read $v_{x,j}$, $j \leq i$. We denote with $\boldsymbol{ts}_{wtr}$ a vector timestamp that captures the writer's knowledge of values written to the system. $\boldsymbol{ts}_{wtr}[u]$ is the largest i for which wtr has read a value $v_{u,i}$. In what follows, we will simply denote values with their indices. So, we will say that a writer writes a vector timestamp instead of writing values whose indices are less than or equal to the indices in the vector timestamp.

We now describe a scenario where a reader would return a value that happens to be $\big((2m-1)(k-1)+1\big)$ writes old.

Consider a multi-writer read operation, where the timestamps for all the m values that the reader receives are similar—specifically, the timestamps

$$Rcvd = \left\{ \begin{array}{c} \langle \quad -1, 0, 0, \ldots, 0\rangle, \\ \langle 0, \quad -1, 0, \ldots, 0\rangle, \\ \langle 0, 0, \quad -1, \ldots, 0\rangle, \\ \vdots \\ \langle 0, 0, 0, \ldots, \quad -1\rangle \end{array} \right\}$$



	Writer 1	Writer 2		Writer m
Phase 0	< 0,?, ?, . . . , ? >	< ?, 0, ?, . . . , ? >		< ?,?, ?, . . . , 0 >
Phase 1	< 1, 0, 0, . . . , 0 > < 2, 0, 0, . . . , 0 > < 3, 0, 0, . . . , 0 > ⋮ < k–1, 0, 0, . . . , 0 >			
Phase 2		< 0, 1, 0, . . . , 0 > < 0, 2, 0, . . . , 0 > < 0, 3, 0, . . . , 0 > ⋮ < 0, k–1, 0, . . . , 0 >	⋮	< 0, 0, 0, . . . , 1 > < 0, 0, 0, . . . , 2 > < 0, 0, 0, . . . , 3 > ⋮ < 0, 0, 0, . . . , k–1 >
Phase 3	k–1 more writes	k–1 more writes	k–1 more writes	k–1 more writes

Fig. 4. Write ordering in the multi-writer k quorum system

where the timestamp for the value received from the i-th writer contains information up to the $(k-1)$-th write by that writer, but only contains information about the 0-th write for all remaining writers.

Since all the m timestamp values are similar, the reader would have no reason to choose one value over the other. Let us assume, without loss of generality, that the reader who reads such a set of timestamp returns the value with the timestamp

$$\langle k-1, 0, 0, \ldots, 0\rangle$$

written by the first writer.

We now show a set of writes to the system wherein the value returned would be $\big((2m-1)(k-1)+1\big)$ writes old. The writes to the system occur in 4 phases.

In phase 0, each of the m writers performs a write operation such that the writer's entry in the corresponding timestamp reads 0. For the sake of this discussion, the non-positive values stored in the other entries of the timestamp are irrelevant. We refer to this write as the 0-th write.

In phase 1, writer 1 – whose value is being returned by the read – performs $(k-1)$ writes. During each of these writes, the reads of the k-atomic register of other writers returns their 0-th write. The timestamp vector associated with each of these writes is shown in Figure 4.

In phase 2, each of the remaining $(m-1)$ writers perform $(k-1)$ writes. Since the underlying single-writer system only provides k-atomic semantics, also during this phase all reads to the underlying single-writer system returns the 0-th write for that writer. Hence the timestamp vector associated with these writes would be as shown in Figure 4.

At the end of phase 2, each writer has performed $k - 1$ writes. The total number of writes performed in this phase is $(m - 1)(k - 1)$.

Finally, in phase 3, each writer performs another $k - 1$ writes. There are a total of $m(k - 1)$ writes in this phase. The exact timestamps associated with these writes are not important.

At the end of phase 3, the multi-writer read takes place. Since the underlying single-writer system only provides k-atomic semantics, all the reads to the underlying single-writer system during the read are only guaranteed to return a value which is not any older than the $(k - 1)$-th write. Thus $Rcvd$ could be the set of values received by the reader where the reader chooses

$$\langle k - 1, 0, 0, \ldots, 0\rangle$$

which is $\big(1 + (m - 1)(k - 1) + m(k - 1)\big)$ writes old.

4.2 Multiple Writer Construction

We present a construction for a m-writer, multi-reader register with relaxed atomic semantics using single-writer, multi-reader registers with relaxed atomic semantics. Using k-atomic registers, our construction provides $\big((2m-1)(k-1)+m\big)$-atomic semantics, which is almost optimal.

The single-writer registers can be constructed using the k-quorum protocols from [9], if servers are subject to crash and recover failures, or using the construction from Section 3 if servers are subject to Byzantine failures. In particular, using the single-writer k-atomic register implementation for Byzantine failures described in Section 3, we obtain an m-writer $\big((2m - 1)(k - 1) + m\big)$-atomic register for Byzantine failures.

The Construction. The multi-write construction uses m instances of the single-writer k-atomic registers, one for each writer w_i.

It uses approximate vector timestamps to compare writes from different writers. Each writer w_i, $1 \leq i \leq m$, maintains a local virtual clock lts_i, which is incremented by 1 for each write so that its value equals the number of writes performed by writer w_i.

At a given time, let $\boldsymbol{gts}$ be defined by

$$\forall i : \boldsymbol{gts}[i] = lts_i$$

where the equality holds at the time of interest. The vector $\boldsymbol{gts}$ represents the global vector timestamp and it may not be known to any of the clients or servers in the system. The read and write protocols are shown in Figure 5.

Write Operation. To perform a write operation, the writer first performs a read to obtain the timestamp information about all the writers (lines 4-5). Since the registers used are k-atomic, each of the received timestamp information is guaranteed to be no more than k writes old for any writer.

A writer wtr_i executing a write would calculate (lines 8-9) an approximate vector timestamp $\boldsymbol{ats}$, whose i-th entry is equal to lts_i and whose remaining entries can be at most k older than the local time stamps of the entries at the time the write operation

```
1   static      = 0;
2   void mw-write(      ,      )
3   begin
4     for j = 1 to m
5       ⟨     , $ts_j$⟩ = sw—read(        )
6
7     // Estimate the approx time—stamp
8     ∀  ≠  : $ats$[j] = max_p { $ts_p$[ ] }
9     $ats$[i] = ++
10
11    sw—write(       , ⟨   , $ats$⟩ )
12  end
```

```
14  ⟨   ,   ⟩ mw-read( )
15  begin
16    for j = 1 to m
17      ⟨     , $ts_j$⟩ = sw—read(        )
18
19    Reject = ∅
20    for i = 1 to m
21      for j = 1 to m
22        if( $ts_j$   $ts_i$ || ($ts_j$[ ]   $ts_i$[ ] −  ) )
23          Reject =       ∪ {⟨     , $ts_j$⟩}
24
25    return any ⟨     , $ts_j$⟩ ∉
26  end
```

Fig. 5. Multi-writer K-quorum protocols

was started. Let $\boldsymbol{gts^{beg}}$ and $\boldsymbol{gts^{end}}$ denote the global timestamps at the start and end of the write. Then,

$$\boldsymbol{ats}[i] = \boldsymbol{gts^{end}}[i]$$

$$\boldsymbol{ats}[j] > \boldsymbol{gts^{beg}}[j] - k$$

$$\boldsymbol{gts^{end}} \geq \boldsymbol{gts^{beg}}$$

The writer then writes the value, val, along with the timestamp $\boldsymbol{ats}$ to the single-writer k-atomic system for the writer.

Read operation. To perform a multi-writer read operation, a reader reads from all the m single-writer k-quorum systems. Because of the k-atomicity of the underlying single-writer implementation, each of these m responses is guaranteed to be one of the k latest values written by each writer. However, if some writer has not written for a long time, then the value could be very old when considering *all* the writes in the system. Finding the latest value among these m values is difficult because the approximate timestamps are not totally ordered.

The reader uses elimination rules (lines 19-23) to reject values that can be inferred to be older than other values. This elimination is guaranteed to reject any value that is more than $\big((2m-1)(k-1)+m\big)$ writes old. Finally, after rejecting old values, the reader returns any value that has not been rejected.

Lemma 1. *If a writer w_i performs a write, beginning at the (global) time $\boldsymbol{gts^{beg}}$ and ending at $\boldsymbol{gts^{end}}$, with a (approximate) timestamp $\boldsymbol{t}$, then*

$$\boldsymbol{t} \leq \boldsymbol{gts^{end}}; \quad \boldsymbol{t}[i] = \boldsymbol{gts^{end}}[i]; \text{ and}$$

$$\forall j : \boldsymbol{t}[j] \geq \boldsymbol{gts^{beg}}[j] - k + 1$$

Lemma 2. *Let $\langle val_j, \boldsymbol{ts_j} \rangle$ be one of the m values read in lines 16-17. If a writer, say s, has performed $2k$ writes after $\langle val_j, \boldsymbol{ts_j} \rangle$ has been written (and before the read starts) then $\langle val_j, \boldsymbol{ts_j} \rangle$ will be rejected in lines 19-23.*

Proof: Let $\boldsymbol{gts_j^{beg}}$,$\boldsymbol{gts_j^{end}}$ and $\boldsymbol{gts_s^{beg}}$,$\boldsymbol{gts_s^{end}}$ denote the global timestamp at the beginning and end of the writes for $\langle val_j, \boldsymbol{ts_j} \rangle$ and $\langle val_s, \boldsymbol{ts_s} \rangle$. Also, let $\boldsymbol{gts_{read}^{beg}}$ be the timestamp when the read is started.

Since writer s has performed at least $2k - 1$ writes after writing $\langle val_j, \boldsymbol{ts_j} \rangle$ we have

$$\boldsymbol{gts^{beg}_{read}}[s] \geq \boldsymbol{gts^{end}_{j}}[s] + 2k$$

Also, from the k-atomic properties of the single writer system, we know that

$$\boldsymbol{ts_s}[s] = \boldsymbol{gts^{end}_{s}}[s] > \boldsymbol{gts^{beg}_{read}}[s] - k$$

$$\Rightarrow \boldsymbol{ts_j}[s] \leq \boldsymbol{gts^{end}_{j}}[s] \leq \boldsymbol{gts^{beg}_{read}}[s] - 2k$$

$$< \boldsymbol{gts^{end}_{s}}[s] - k = \boldsymbol{ts_s}[s] - k$$

Hence $\langle val_j, \boldsymbol{ts_j} \rangle$ will be added to Reject in line 23. □

Theorem 7. *The multi-writer read protocol never returns a value that is more than* $\big((2m-1)(k-1)+m\big)$ *writes old.*

Proof: Let $\langle val_j, \boldsymbol{ts_j} \rangle$ be the value returned by the read protocol.

The writer j cannot have written more than $k - 1$ writes after $\langle val_j, \boldsymbol{ts_j} \rangle$ (and before the read begins). From Lemma 2 it follows that each of the remaining $(m - 1)$ writers could have written no more than $2k - 1$ writes after the write for $\langle val_j, \boldsymbol{ts_j} \rangle$ (and before the read begins).

Hence, $\langle val_j, \boldsymbol{ts_j} \rangle$ can be at most $\big(1 + (k-1) + (m-1)(2k-1)\big)$ writes old. □

Lemma 3. *At least one of the m received values is not rejected.*

Theorem 8. *The multi-writer protocol described in Figure 5 provides* $\big((2m-1)(k-1)+m\big)$*-atomic semantics.*

Availability of a Multi-writer System We now estimate the availability of the multi-writer system, assuming that the underlying single-writer k-quorum system has a read and write availability of $a_{sr} = 1 - \epsilon_{sr}$ and $a_{sw} = 1 - \epsilon_{sw}$ respectively.

Each multi-writer write operation involves reading from all the m single-writer k-quorum systems and writing to one single-writer system. Hence the write availability of the multi-writer system, a_{mw}, is at least $(a_{sr})^m a_{sw}$. This is a conservative estimate because we are assuming that, when the network is synchronous, we treat finding a read quorum and finding a partial-write-quorum as independent events. In practice, however, the fact that a particular number of servers (size of read quorum) are up and accessible only increases the probability of being able to find an accessible partial-write-quorum.

Moreover, If the m underlying single-writer k-quorum systems are implemented over the same strict quorum system, then the potential read quorums that can be used for all the m systems will be the same.[4] Thus, we can use the same read quorum to perform all the m read operations. In this case, either all reads are available with probability a_{sr} or all reads fail with probability ϵ_{sr}. Hence the probability of the multi-writer write succeeding is at least $a_{sr}a_{sw}$.

$$a_{mw} \geq a_{sr}a_{sw} \geq 1 - \epsilon_{sr} - \epsilon_{sw}$$

[4] The partial-write-quorums could still be different, if the writers have chosen different partial-write-quorums in the past.

To perform a multi-writer read, our read protocol performs m reads from the m single writer k-quorum implementations. Thus, along similar lines, we can argue that the availability a_{mr} is at least ${a_{sr}}^m$. Using the same underlying strict quorum system for all the m single-writer systems, we can achieve an availability of

$$a_{mr} = a_{sr} = 1 - \epsilon_{sr}$$

Probabilistic freshness guarantees We now estimate the probability that our multi-writer implementation of k-quorums provides the latest value, when all the writes that occur are non-overlapping.

Let δ_{sw} denote the probability that a sw-read does not return the latest value written to the single-writer system. Let δ_{mw} denote the probability that the multi-writer system does not return the latest value written to the system.

Theorem 9. *If the operations are non-overlapping, the probability that the multiple-writer system does not return the latest value is at most $m\delta_{sw}$*

Once again, the proof can be found in our extended technical report [12].

5 Conclusion and Future Work

In this paper we expand our understanding of k-quorum systems in three key directions [9].

First, we present a single-writer k-quorum construction that tolerates Byzantine failures. Second, we prove a lower bound of $\big((2m-1)(k-1)+1\big)$ on the staleness for a m writer solution built over a single-writer k-quorum solution.

Finally, we demonstrate a technique to build multiple-writer multiple-reader k-quorum protocols using a single-writer multiple-reader protocol to achieve $\big((2m-1)(k-1)+m\big)$-atomic semantics.

One limitation of our approach is that it improves availability only when writes are infrequent. Also, we have restricted our study of multi-writer solutions to those that built over a single-writer k-quorum system; it may be possible that a direct implementation can achieve a better staleness guarantee.

References

1. Raynal, M., Beeson, D.: Algorithms for mutual exclusion. MIT Press, Cambridge, MA, USA (1986)
2. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proc. of the Third Symposium on Operating Systems Design and Implementation, USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS (1999)
3. Susan Davidson, H.G.M., Skeen, D.: Consistency in partioned network. Computing Survey **17**(3) (1985)
4. Herlihy, M.: Replication methods for abstract data types. Technical Report TR-319, MIT/LCS (1984)
5. Naor, M., Wool, A.: The load, capacity, and availability of quorum systems. SIAM Journal on Computing **27**(2) (1998) 423–447

6. Peleg, D., Wool, A.: The availability of quorum systems. Inf. Comput. **123**(2) (1995) 210–223
7. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Inf. Comput. **170**(2) (2001) 184–206
8. Yu, H.: Signed quorum systems. In: Proc. 23rd PODC, ACM Press (2004) 246–255
9. Aiyer, A., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: DISC '05, London, UK, Springer-Verlag (2005) 48–62
10. Lamport, L.: On interprocess communication. part i: Basic formalism. Distributed Computing **1**(2) (1986) 77–101
11. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: DISC '02, London, UK, Springer-Verlag (2002) 311–325
12. Aiyer, A., Alvisi, L., Bazzi, R.A.: Byzantine and multi-writer k-quorums. Number TR-06-37 (2006)

On Minimizing the Number of ADMs in a General Topology Optical Network

Michele Flammini[1], Mordechai Shalom[2], and Shmuel Zaks[2,⋆]

[1] Universita degli Studi dell'Aquila Dipartmento di Informatica, L'Aquila-Italy
flammini@di.univaq.it
[2] Department of Computer Science, Technion, Haifa-Israel
{cmshalom, zaks}@cs.technion.ac.il

Abstract. Minimizing the number of electronic switches in optical networks is a main research topic in recent studies. In such networks we assign colors to a given set of lightpaths. Thus the lightpaths are partitioned into cycles and paths, and the switching cost is minimized when the number of paths is minimized. The problem of minimizing the switching cost is NP-hard. A basic approximation algorithm for this problem eliminates cycles of size at most l, and has a performance guarantee of $OPT + \frac{1}{2}(1+\epsilon)N$, where OPT is the cost of an optimal solution, N is the number of lightpaths, and $0 \leq \epsilon \leq \frac{1}{l+2}$, for any given odd l. We improve the analysis of this algorithm and prove that $\epsilon \leq \frac{1}{\frac{3}{2}(l+2)}$. This implies an improvement in the running time of the algorithm: for any ϵ, the exponent of the running time needed for the same approximation ratio is reduced by a factor of 3/2. We also show a lower bound of $\epsilon \geq \frac{1}{2l+3}$. In addition, in our analysis we suggest a novel technique, including a new combinatorial lemma.

Keywords: Wavelength Assignment, Wavelength Division Multiplexing(WDM), Optical Networks,Add-Drop Multiplexer(ADM).

1 Introduction

1.1 Background

Given a WDM network $G = (V, E)$ comprising optical nodes and a set of full-duplex lightpaths $P = \{p_1, p_2, ..., p_N\}$ of G, the wavelength assignment (WLA) task is to assign a wavelength to each lightpath p_i.

In the following discussion we also assume that each lightpath $p \in P$ is contained in a cycle of G. Each lightpath p uses two ADM's, one at each endpoint. Although only the downstream ADM function is needed at one end and only the upstream ADM function is needed at the other end, full ADM's will be installed

[⋆] This research was supported in part by the fund for the promotion of research at the Technion, by B. and I. Green research fund, and by the EU COST 293 research fund.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 459–473, 2006.

on both nodes in order to complete the protection path around some ring. The full configuration would result in a number of SONET rings. It follows that if two adjacent lightpaths are assigned the same wavelength, then they can be used by the same SONET ring and the ADM in the common node can be shared by them. This would save the cost of one ADM. An ADM may be shared by at most two lightpaths. A more detailed technical explanation can be found in [1].

Lightpaths sharing ADM's in a common endpoint can be thought as concatenated, so that they form longer paths or cycles. Each of these longer paths/cycles does not use any edge $e \in E$ twice, for, otherwise they cannot use the same wavelength and this is a necessary condition to share ADM's.

1.2 Previous Work

Minimizing the number of electronic switches in optical networks is a main research topic in recent studies. The problem was introduced in [1] for ring topology. Approximation algorithm for ring topology with approximation ratio of 3/2 was presented in [2], and was improved in [3,4] to $10/7+\epsilon$ and 10/7, respectively.

For general topology [5] describe an algorithm with approximation ratio of 8/5. The same problem was studied in [6] and an algorithm was presented that has a preprocessing phase where cycles of length at most l are included in the solution; this algorithm was shown to have performance guarantee of

$$OPT + \frac{1}{2}(1+\epsilon)N, \quad 0 \leq \epsilon \leq \frac{1}{l+2} \tag{1}$$

where OPT is the cost of an optimal solution, N is the number of lightpaths, for any given odd l. The dominant part in the running time of the algorithm is the preprocessing phase, which is exponential in l.

For $l = 1$ this implies algorithm without preprocessing, having performance guarantee of $OPT + \frac{2}{3}N$. In [7] this algorithm is proven to have a performance guarantee of $OPT + \frac{3}{5}N$ and this bound is shown to be tight.

1.3 Our Contribution

We improve the analysis of the algorithm of [6] and prove a performance of

$$OPT + \frac{1}{2}(1+\epsilon)N, \quad \frac{1}{2l+3} \leq \epsilon \leq \frac{1}{\frac{3}{2}(l+2)}. \tag{2}$$

Specifically, we show that the algorithm guarantees to satisfy an upper bound of $OPT + \frac{1}{2}(1+\epsilon)N$, for $\epsilon \leq \frac{1}{\frac{3}{2}(l+2)}$, and we demonstrate a family of instances for which the performance of the algorithm is $OPT + \frac{1}{2}(1+\epsilon)N$, for $\epsilon \geq \frac{1}{2l+3}$.

Our analysis sheds more light on the structure and properties of the algorithm, by closely examining the structural relation between the solution found by the algorithm vs an optimal solution, for any given instance of the problem.

As the running time of the algorithm is exponential in l, our result imply an improvement in the analysis of the running time of the algorithm. For any given

$\epsilon > 0$, the exponent of the running time needed to guarantee the approximation ratio $(3 + \epsilon)/2$ is reduced by a factor of $3/2$.

In addition, in the development of our bounds we use a purely combinatorial problem, which is of interest by itself.

In Section 2 we describe the problem and some preliminary results. The algorithm and its analysis are presented in Section 3. We conclude with discussion and open problems in Section 4. Some proofs are sketched or omitted in this Extended Abstract.

2 Problem Definition and Preliminary Results

2.1 Problem Definition

An instance α of the problem is a pair $\alpha = (G, P)$ where $G = (V, E)$ is an undirected graph and P is a set of simple paths in G. Given such an instance we define the following:

Definition 2.1 *The paths $p, p' \in P$ are conflicting or overlapping if they have an edge in common. This is denoted as $p \asymp p'$. The graph of the relation $\asymp$ is called the conflict graph of (G, P).*

Definition 2.2 *A proper coloring (or wavelength assignment) of P is a function $w : P \mapsto \mathbb{N}$, such that $w(p) \neq w(p')$ whenever $p \asymp p'$.*

Note that w is a proper coloring if and only if for any color $c \in \mathbb{N}$, $w^{-1}(c)$ is an independent set in the conflict graph.

Definition 2.3 *A valid chain (resp. cycle) is a path (resp.cycle) formed by the concatenation of distinct paths $p_0, p_1, ..., p_{k-1} \in P$ that do not go over the same edge twice. Note that the paths of a valid chain (resp. cycle) constitute an independent set of the conflict graph.*

Definition 2.4 *A solution S of an instance $\alpha = (G, P)$ is a set of chains and cycles of P such that each $p \in P$ appears in exactly one of these sets.*

In the sequel we introduce the shareability graph, which together with the conflict graph constitutes another (dual) representation of the instance α. In the sequel, except one exception, we will use the dual representation of the problem.

Definition 2.5 *The shareability graph of an instance $\alpha = (G, P)$, is the edge-labelled multi-graph $\mathcal{G}_\alpha = (P, E_\alpha)$ such that there is an edge $e = (p, q)$ labelled u in E_α if and only if $p \not\asymp q$, and u is a common endpoint of p and q in G.*

Example: Let $\alpha = (G, P)$ be the instance in Figure 1. Its shareability graph $\mathcal{G}_\alpha$ is the graph at the left side of Figure 2. In this instance $P = \{a, b, c, d\}$, and it constitutes the set of nodes of $\mathcal{G}_\alpha$. The edges together with their labels are $E_\alpha = \{(b, c, u), (a, c, w), (a, b, x), (a, d, x)\}$, because a and b can be joined in their common endpoint u, etc.. Note that, for instance $(b, d, x) \notin E_\alpha$, because although b and d share a common endpoint x, they can not be concatenated, because they have the edge (x, u) in common. The corresponding conflict graph is the graph at the right side of Figure 2. It has the same node set and one edge, namely (b, d). The paths $b, d \in P$ are conflicting because they have a common edge, i.e. (u, v).

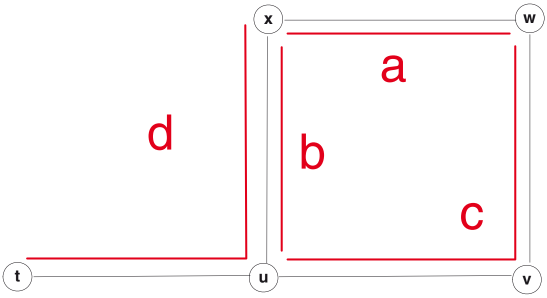


Fig. 1. A sample input

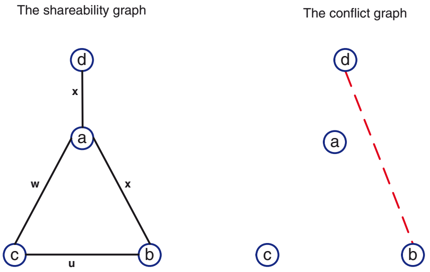


Fig. 2. The shareability and conflict graphs

Note that the edges of the conflict graph are not in E_α. This immediately follows from the definitions.

Note also that, for any node v of $\mathcal{G}_\alpha$, the set of labels of the edges adjacent to v is of size at most two.

Definition 2.6 *A valid chain (resp. cycle) of $\mathcal{G}_\alpha$ is a simple path $p_0, p_1, ..., p_{k-1}$ of $\mathcal{G}_\alpha$, such that any two consecutive edges in the path (resp. cycle) have distinct labels and its node set is properly colorable with one color (in G), or in other words constitutes an independent set of the conflict graph.*

Note that the valid chains and cycles of $\mathcal{G}_\alpha$ correspond to valid chains and cycles of the instance α. In the above example the chain a, d which is the concatenation of the paths a and d in the graph G, corresponds to the simple path a, d in $\mathcal{G}_\alpha$

and the cycle a, b, c which is a cycle formed by the concatenation of three paths in G corresponds to the cycle a, b, c in $\mathcal{G}_\alpha$. Note that no two consecutive labels are equal in this cycle. On the other hand the paths b, a, d can not be concatenated to form a chain, because this would require the connection of a to both b and d at node x. The corresponding path b, a, d in $\mathcal{G}_\alpha$ is not a chain because the edges (b, a) and (a, d) have the same label, namely x.

Definition 2.7 *The sharing graph of a solution S of an instance $\alpha = (G, P)$, is the following subgraph $\mathcal{G}_{\alpha,S} = (P, E_S)$ of $\mathcal{G}_\alpha$. Two lightpaths $p, q \in P$ are connected with an edge labelled u in E_S if and only if they are consecutive in a chain or cycle in the solution S, and their common endpoint is $u \in V$. We will usually omit the index α and simply write $\mathcal{G}_S$. $d(p)$ is the degree of node p in $\mathcal{G}_S$.*

In our example, $S = \{(d, a, c), (b)\}$ is a solution with two chains. The sharing graph of this solution is depicted in Figure 3. Note that for a chain of size at most two, the distinct labelling condition is satisfied vacuously, and the independent set condition is satisfied because no edge of $\mathcal{G}_\alpha$ can be an edge of the conflict graph.

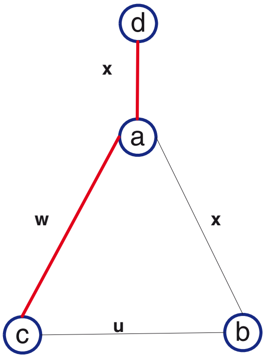


Fig. 3. A possible solution

We define $\forall i \in \{0, 1, 2\}, D_i(S) \overset{def}{=} \{p \in P | d(p) = i\}$ and $d_i(S) \overset{def}{=} |D_i(S)|$. Note that $d_0(S) + d_1(S) + d_2(S) = |P| = N$.

An edge $(p, q) \in E_S$ with label u corresponds to a concatenation of two paths with the same color at their common endpoint u. Therefore these two endpoints can share an ADM operating at node u, thus saving one ADM. We conclude that every edge of E_S corresponds to a saving of one ADM. When no ADMs are shared, each path needs two ADM's, a total of $2N$ ADMs. Therefore the cost of a solution S is $2|P| - |E_S| = 2N - |E_S|$.

The objective is to find a solution S such that $cost(S)$ is minimum, in other words $|E_S|$ is maximum.

2.2 Preliminary Results

Given a solution S, $d(p) \leq 2$ for every node $p \in P$. Therefore, the connected components of $\mathcal{G}_S$ are either paths or cycles. Note that an isolated vertex is a special case of a path. Let $\mathcal{P}_S$ be the set of the connected components of $\mathcal{G}_S$ that are paths. Clearly, $|E_S| = N - |\mathcal{P}_S|$. Therefore $cost(S) = 2N - |E_S| = N + |\mathcal{P}_S|$.

Let S^* be a solution with minimum cost. For any solution S we define

$$\epsilon(S) \overset{def}{=} \frac{d_0(S) - d_2(S) - 2\,|\mathcal{P}_{S^*}|}{N}.$$

Lemma 1. *For any solution S, $cost(S) = cost(S^*) + \frac{1}{2}N(1 + \epsilon(S))$.*

Proof. Clearly $|E_{S^*}| = N - |\mathcal{P}_{S^*}|$. On the other hand $2\,|E_S|$ is the sum of the degrees of the nodes in $\mathcal{G}_S$, namely $2\,|E_S| = d_1(S) + 2d_2(S) = N - d_0(S) + d_2(S)$. We conclude:

$$\begin{aligned} cost(S) - cost(S^*) &= |E_{S^*}| - |E_S| = N - |\mathcal{P}_{S^*}| - \frac{N - d_0(S) + d_2(S)}{2} \\ &= \frac{1}{2}N\left(1 + \frac{d_0(S) - d_2(S) - 2\,|\mathcal{P}_{S^*}|}{N}\right) \end{aligned}$$ □

The following definition extends the concept of a chord from cycles to paths.

Definition 2.8 *Given an instance $\alpha = (G, P)$ and a solution S of α, an edge (p, q) of $\mathcal{G}_\alpha$ is a chord of S if both p and q are in the same connected component of $\mathcal{G}_S$ and $(p, q) \notin E_S$.*

Lemma 2. *For every instance $\alpha = (G, P)$ and there is an optimal solution S^* without chords.*

Sketch of Proof.. Note that any two solutions S_1, S_2 of α such that $cost(S_1) = cost(S_2)$, have the same number of chains, whereas the number of cycles may differ. Let S^* be a solution with maximum number of cycles among the solutions with minimum cost, i.e. optimal. For the complete proof that S^* satisfies the claim, see [8]. □

In the sequel we will always use the dual representation. Henceforth, an element p of P is referred as a node (of $\mathcal{G}_\alpha$), and a path refers to a path of $\mathcal{G}_\alpha$.

3 Main Results

3.1 Algorithm $PMM(l)$

In this section we describe Algorithm $PMM(l)$ presented in [2].

The algorithm with has a preprocessing phase which removes cycles of size at most l, where l is an odd number. Then it proceeds to its processing phase (Function MM) which can be described as follows:

Begin with chains consisting of single nodes (which are always valid). At each iteration, combine a maximum number of pairs of chains to obtain longer chains. This is done by constructing an appropriate graph and computing a maximum matching on it. The algorithm ends when the maximum matching is empty, namely no two chains can be combined into a longer chain.

```
Function MM($\alpha$) {
  Phase 0) $E_S = \emptyset$
          // the chains of $\mathcal{G}_S$ are isolated nodes.
  Phase 1)
    Do {
       Build the graph $\mathcal{G}'_\alpha$ in which each node is a chain of $\mathcal{G}_S$
       and there is an edge labelled $u$ between two chains if
       and only if the chains can be merged into one bigger
       chain by joining them at a common endpoint $u$.
       //In the first iteration $\mathcal{G}'_\alpha = \mathcal{G}_\alpha$
       Find a maximum matching $MM$ of $\mathcal{G}'_\alpha$.
       For each edge $e = (c, c')$ of $MM$ labelled $u$ do {
         Merge the corresponding chains into one chain
         by joining them in the common endpoint $u$
       }
    } Until $MM = \emptyset$.
  return($S$).
}
```

```
Procedure PMM($l$)($\alpha = G, P$) {
Preprocessing:
  Find a maximal set $S_0$ of disjoint valid cycles of length $\leq$
   $l$ in $P$.
  $P_0$ is the set of nodes of the cycles of $S_0$.
  $P_1 \leftarrow P \setminus P_0$. //$S_0$ is maximal, therefore $P_1$
              //does not contain feasible cycles of
              //length $\leq l$.
Processing:
  $\alpha_1 \leftarrow (G, P_1)$.
  $S_1 \leftarrow MM(\alpha_1)$.
  $E_S \leftarrow E_{S_0} \cup E_{S_1}$
  return($S$).
}
```

3.2 Correctness

We first prove the correctness of MM: After Phase 0, the chains of S consist of single nodes. Trivially, these are valid chains. At each iteration of Phase 1, a new chain is constructed only if it is valid, because edges are added to $\mathcal{G}'_\alpha$ only if the corresponding chains can be merged into one chain. Each edge of a matching represents a valid merging operation. Moreover two such valid operations do not affect each other, because each such operation is performed on two chains matched by an edge of some matching. Therefore after each iteration the solution consists of valid chains.

We now conclude with the correctness of $PMM(l)$: S_0 consists of disjoint valid cycles. S_1 consists of disjoint valid chains because of the correctness of MM. Moreover $P_0 \cap P_1 = \emptyset$, therefore S is a set of disjoint valid cycles and chains, i.e. a solution.

3.3 Analysis

We begin our analysis with Lemma 3 which is proven, although in other terminology, in [2]. This will be helpful in understanding the main result of this section, i.e. the improved upper bound. The proof is based on the existence of a matching M having certain size. This matching consists solely of edges of the connected components of $\mathcal{G}_{S^*}$. In our proof we show that using other edges of $\mathcal{G}_\alpha$ we can build a larger matching which leads to a higher upper bound. In Subsection II, we develop a lower bound on the number of edges in $E_S \setminus E_{S^*}$. In Subsection III we prove a combinatorial lemma, which helps us to to build our matching. In Subsection IV we build the improved matching and prove our upper bound. In Subsection V we give a lower bound for the performance of the algorithm.

I-An upper bound

In the sequel S is a solution returned by the algorithm and S^* is an optimal solution without chords, whose existence is guaranteed by Lemma 2.

Lemma 3. *For any solution S of $PMM(l)$, $\epsilon(S) \le \frac{1}{l+2}$.*

Sketch of Proof.. This lemma is actually proven in [2]. For a complete proof in our notations, see [8]. □

We begin by developing some results which will be used in our proof. The first family of results gives a lower bound on the number of edges "touching" cycles of $\mathcal{G}_{S^*}$.

II-Lower bounds for edges of $E_S \setminus E_{S^*}$

Definition 3.1 *For every $X \subseteq P$, $OUT(X) \overset{def}{=} C(X, \overline{X})$ is the cut of X in $\mathcal{G}_S$, namely the set of edges of $\mathcal{G}_S$ having exactly one endpoint in X.*

Lemma 4. *Let C be a cycle of $\mathcal{G}_{S^*}$, then*

$$|OUT(C)| \ge \frac{1}{3}(|C| + |D_0(S) \cap C| - |D_2(S) \cap C|).$$

Proof. Let k be the number of edges of C which are not part of $\mathcal{G}_S$ and $\forall i \in \{0,1,2\}, dc_i \overset{def}{=} |D_i(S) \cap C|$.

The sum of the degrees (in $\mathcal{G}_S$) of the nodes of C is

$$dc_1 + 2dc_2 = |C| - dc_0 + dc_2.$$

On the other hand each edge of $\mathcal{G}_S$ connecting nodes of C contributes 2 to this sum and each edge in $OUT(C)$ contributes 1. As there are no chords of C, the number of edges contributing 2 is $|C| - k$. Therefore

$$\begin{aligned} |C| - dc_0 + dc_2 &= 2(|C| - k) + |OUT(C)| \\ |C| + dc_0 - dc_2 &= 2\ k - |OUT(C)|\,. \end{aligned} \tag{3}$$

For the following discussion consult Figure 4. Consider an edge $e = (p, q)$ of C which is not in $\mathcal{G}_S$. This edge was not added to E_S by the algorithm. This could be only because a node p' in the connected component of p in $\mathcal{G}_S$ is conflicting with a node q' in the connected component of q in $\mathcal{G}_S$. Either $p' \notin C$ or $q' \notin C$, otherwise they would not be conflicting. Assume w.l.o.g. that $p' \notin C$. Let p' be the node closest to p among such nodes. By the choice of p, there is an edge e' connecting p' to a node in C. We call e' the blocking edge of e. Moreover $e' \in OUT(C)$. Therefore, any edge e of C which is not in $\mathcal{G}_S$ has a blocking edge, and any edge in $OUT(C)$ may be a blocking edge of at most two edges. Therefore

$$k \leq 2\,|OUT(C)|\,. \tag{4}$$

Combining (3) and (4) we get

$$|C| + dc_0 - dc_2 = 2k - |OUT(C)| \leq 3\,|OUT(C)|$$

$$|OUT(C)| \geq \frac{1}{3}(|C| + dc_0 - dc_2). \qquad \square$$

Definition 3.2 *The i-neighborhood $N_i(X)$ of X is the set of all the nodes having exactly i neighbors from X in $\mathcal{G}_S$, but are not in X. $N(X) \overset{def}{=} N_1(X)$.*

The following lemma generalizes the previous lemma to a set of cycles.

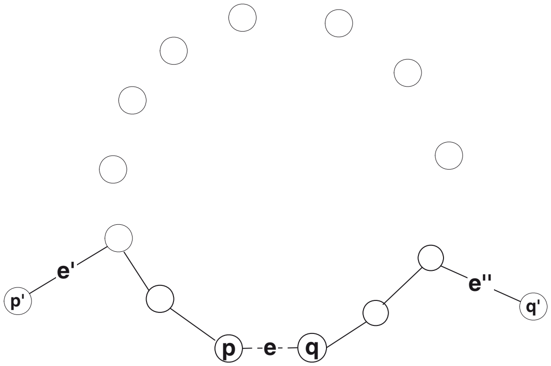


Fig. 4. Blocking and blocked edges

Lemma 5. *Let $\mathcal{C}$ be a set of cycles of $\mathcal{G}_{S^*}$. Let $P_{\mathcal{C}} \overset{def}{=} \cup\mathcal{C}$ be the set of nodes of these cycles. Let $IN(\mathcal{C})$ be the set of edges of $\mathcal{G}_S$ connecting two cycles of $\mathcal{C}$. Then*

$$|N(P_{\mathcal{C}})| \geq \frac{1}{3}|P_{\mathcal{C}}| + \frac{1}{3}|D_0(S) \cap P_{\mathcal{C}}| - \frac{1}{3}|D_2(S) \cap P_{\mathcal{C}}| - 2|IN(\mathcal{C})| - 2|N_2(P_{\mathcal{C}})|$$

Proof. Consider the sum $\sum_{C\in\mathcal{C}}|OUT(C)|$. Each edge in $OUT(P_{\mathcal{C}})$ is counted in this sum. On the other hand each edge in $IN(\mathcal{C})$ is counted twice (once for each cycle it connects) where it should not be counted at all. Similarly each edge having one endpoint in $N_2(P_{\mathcal{C}})$ is counted once where it should not be counted at all. The number of these edges is $2|N_2(P_{\mathcal{C}})|$.

$$\begin{aligned}|N(P_{\mathcal{C}})| &= \sum_{C\in\mathcal{C}}|OUT(C)| - 2|IN(\mathcal{C})| - 2|N_2(P_{\mathcal{C}})| \\ &\geq \frac{1}{3}(|P_{\mathcal{C}}| + |D_0(S)\cap P_{\mathcal{C}}| - |D_2(S)\cap P_{\mathcal{C}}|) - 2|IN(\mathcal{C})| - 2|N_2(P_{\mathcal{C}})|. \quad \square\end{aligned}$$

Definition 3.3 *The odd cycles graph $\mathcal{OG}_S = (\mathcal{OC}_S, \mathcal{OE}_S)$ of a solution S is a graph in which each node corresponds to an odd cycle of $\mathcal{G}_{S^*}$ which does not intersect with P_0 and two nodes are connected with an edge if and only if there is an edge connecting the corresponding cycles in E_s.*

Lemma 6. *Let $\mathcal{X} \subseteq \mathcal{OC}_S$. Then $|N(P_{\mathcal{X}})| \geq \frac{1}{3}|P_{\mathcal{X}}| - 2|IN(\mathcal{X})| - 2(d_2(S) - |P_0|)$.*

Proof. First, we show that $N_2(P_{\mathcal{X}}) \subseteq D_2(S) \setminus P_0 \setminus P_{\mathcal{X}}$. Let $p \in N_2(P_{\mathcal{X}})$. By definition p has degree 2, namely $p \in D_2(S)$. Still by definition $p \notin P_{\mathcal{X}}$. It remains to show that $p \notin P_0$. By definition p has both of its neighbors in $P_{\mathcal{X}}$. Assume $p \in P_0$, then p is in some cycle of S_0. Then both of its neighbors are in this cycle, thus in P_0. But they are also in $P_{\mathcal{X}}$, contradicting the fact that by definition, the cycles of $\mathcal{X}$ do not intersect with P_0. Therefore $|N_2(P_{\mathcal{X}})| \leq d_2(S) - |P_0| - |D_2(S) \cap P_{\mathcal{X}}|$. Substituting this in Lemma 5 we get

$$\begin{aligned}&|N(P_{\mathcal{X}})| \geq \frac{1}{3}|P_{\mathcal{X}}| + \frac{1}{3}|D_0(S)\cap P_{\mathcal{X}}| - \frac{1}{3}|D_2(S)\cap P_{\mathcal{X}}| - 2|IN(\mathcal{X})| \\ &-2(d_2(S) - |P_0| - |D_2(S)\cap P_{\mathcal{X}}|) \\ &= \frac{1}{3}|P_{\mathcal{X}}| + \frac{1}{3}|D_0(S)\cap P_{\mathcal{X}}| + \frac{5}{3}|D_2(S)\cap P_{\mathcal{X}}| - 2|IN(\mathcal{X})| - 2(d_2(S) - |P_0|) \\ &\geq \frac{1}{3}|P_{\mathcal{X}}| - 2|IN(\mathcal{X})| - 2(d_2(S) - |P_0|). \qquad \square\end{aligned}$$

Corollary 3.1 *Let $\mathcal{I}$ be an independent set of $\mathcal{OG}_S$. Then $|N(P_{\mathcal{I}})| \geq \frac{1}{3}|P_{\mathcal{I}}| - 2(d_2(S) - |P_0|)$.*

Proof. By definition $\forall u, v \in \mathcal{I}, (u, v) \notin \mathcal{OE}$. This means that these are not connected by an edge in E_S. In other words $IN(\mathcal{I}) = \emptyset$. □

III-Odd Distanced Nodes with Distinct Colors

In this subsection we develop a result which will be an essential tool in building the matching in Subsection IV, and proving a lower bound on its size. For this purpose we define the "maximum odd distanced nodes with distinct colors" family of problems which are pure combinatorial problems of their own interest.

The cycle version of the problem, $(MODNDC - C)$ is defined as follows:

Input: A cycle C with n nodes numbered from 1 to n clockwise, some of which are colored and the rest are not. If a node is colored, $c(v) \in \mathbb{N}$ denotes its color, otherwise $c(v) = 0$ and it is termed *uncolored.*

Output: A cyclic subsequence $V = (v_0, v_1, ..., v_{k-1})$ of the nodes of C such that:

- Odd distanced: Between every pair of successive nodes $v_i, v_{j=i+1 \mod k} \in V$, the clockwise distance $d(v_i, v_j)$ from v_i to v_j is odd. Note that in particular if $k = 1$ then the $d(v_0, v_0) = n$ is be odd.
- Distinct Colors: Every node v_i in the sequence is colored (i.e. $c(v_i) \neq 0$) and for every pair of distinct nodes v_i and v_j in the sequence, $c(v_i) \neq c(v_j)$.

Measure: Our goal is to find V maximizing the number of nodes of C which are colored with colors from $\{c(v_0), c(v_1), ..., c(v_{k-1})\}$. In the sequel it will be easier to measure a solution V by the number of nodes of C which are colored with colors from $\{c(v_0), c(v_1), ..., c(v_{k-1})\}$, plus the number of nodes which are not colored. In other words, given a solution, we first set $c(v) = 0$ for all v such that $c(v) = \{c(v_0), c(v_1), ..., c(v_{k-1})\}$ and we count the number of nodes v with $c(v) = 0$. We define as $B_c(V)$ the set of nodes colored c after this uncoloring, formally $B_c(V) \overset{def}{=} \{v \in C | c(v) = c\}$. $W(V) \overset{def}{=} B_0$ is the set of uncolored nodes. $B(V) \overset{def}{=} \uplus_{c>0} B_c$ is the set of colored nodes. Our target is to find a solution V such that $|W(V)|$ is maximized. Obviously $C = B(V) \uplus W(V)$.

Definition 3.4 *A cycle C is dedicated if it contains nodes colored with one color and possibly some uncolored nodes. Formally, $|\{c(v)|v \in C\} \setminus \{0\}| = 1$.*

Lemma 7. *Given an instance of the $(MODNDC - C)$ problem, one of the following is true:*

- *(a) C is a dedicated even cycle.*
- *(b) There is a solution V with measure $|W(V)| \geq \lceil \frac{n}{3} \rceil$*

Proof. Let V be an optimal solution. We consider the following cases:

- **Case 1:** $V = \emptyset$**.** It follows from the definition that, if V' and V'' are two solutions such that $V' \subset V''$, then $W(V') \subset W(V'')$, thus $|W(V')| < |W(V')|$. In particular, for any solution $V' \neq \emptyset$, $|W(V')| > |W(\emptyset)|$. As we assumed $V = \emptyset$, it follows that no other solution is feasible.

  If all the nodes are uncolored then $|W(\emptyset)| = n$, thus (b) holds. Otherwise there are some colored nodes. If n is odd, then any singleton of the colored nodes is a non-empty solution, a contradiction. Therefore n is even. If there is only one color, then this is a dedicated even cycle and (a) holds. Otherwise there are at least two colors. Since $V = \emptyset$ no pair of nodes is a solution. Then, for any pair u, v of nodes, either they are an even distance apart, or $c(u) = c(v)$. Fix some node v and let $c(v) = a >$. Then all the nodes u such that $c(u) \neq a$ are at even distance from v. We claim that all the nodes u' such that $c(u') = a$ are also at even distance from v. Assume that there is a node u' such that $c(u') = a$ at odd distance from v, then it is at odd distance from the nodes u such that $c(u) \neq a$. Then u' together with one of the u nodes is a solution, contradiction our assumption. Then all the colored nodes are at even distance from u. We conclude that all the nodes at odd distance from u are uncolored. Then $|W(\emptyset)| \geq \frac{n}{2}$.

- **Case 2:** $V \neq \emptyset$**.** We want to show that $|W(V)| \geq \frac{n}{3} = \frac{|W|}{3} + \frac{|B|}{3}$ which is equivalent to $|B| \leq 2|W|$. For this purpose we will partition the set B into two disjoint sets X, Y, and then prove $|X| \leq |W|$ and $|Y| \leq |W|$.

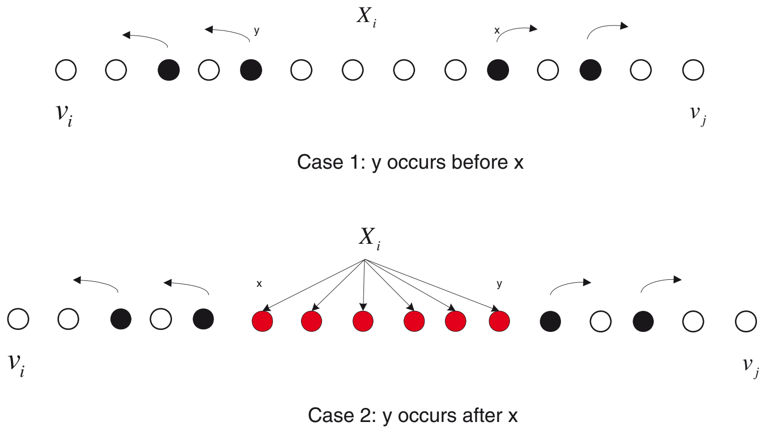
  Let $V = \{v_0, v_1, ..., v_{k-1}\}$. Consider two consecutive nodes $v_i, v_j \in V$. Note that $i = j$ if $k = 1$, thus these nodes need not be distinct. Recall also that the clockwise distance $d(v_i, v_j)$ from v_i to v_j is odd.

  Observe that if there are two colored nodes $x, y \in B(V)$ between these two nodes such that x is closer to v_i and that $d(v_i, x)$ and $d(x, y)$ are odd, then $c(x) = c(y)$. For, otherwise the set $V \uplus \{x, y\}$ is a better solution than V, a contradiction.

  We use this observation to characterize the colored nodes of the solution, i.e. the nodes of $B(V)$. For the following discussion consult Figure 5. Let $x \in B(V)$ be the colored node which is closest to v_i when going clockwise from v_i to v_j and is at odd distance from v_i. Let $y \in B(V)$ be the colored node which is farthest from v_i when going from v_i to v_j and is at even distance from v_i. Note that y is the first node in $B(V)$ at odd distance from v_j when going counterclockwise from v_j to v_i. By these choices, all the colored nodes before x are at even distance from v_i and all the colored nodes after y are at odd distance from v_i. If y occurs before x then there are no colored nodes between x and y, or in other words, all the colored nodes are either before y or after x. Note that this statement holds even if one or both of x, y do not exist. In all these cases we define $X_i = \emptyset$. If y occurs after x then by the observation in the previous paragraph $c(x) = c(y) = c$. Furthermore, by the same observation, for every colored node z between x and y, $c(z) = c$. In this case we define X_i be the set of all the colored nodes from x to y including x and y. Let also Y_i be the set of all other colored nodes between v_i and v_j. Let $X \overset{def}{=} \uplus_{i=0}^{k-1} X_i$ and $Y \overset{def}{=} \uplus_{i=0}^{k-1} Y_i$.

  Obviously $|Y| \leq |W|$, for the nodes of Y are separated by at least one node in W.

  Let $V_i \subseteq W$ be the set of nodes having originally the same color as v_i. Note that X_i has at least one node x which is at even distance from v_i. Therefore $V' = V \setminus \{v_i\} \cup \{x\}$ is a solution. If $|X_i| > |V_i|$ then $|W(V')| > |W(V)|$, a contradiction, hence $|X_i| \leq |V_i|$. Summing up from $i = 0$ to $k - 1$ we have $|X| \leq \uplus_{i=0}^{k-1} |V_i| \leq |W|$.



Case 1: y occurs before x

Case 2: y occurs after x

Fig. 5. The nodes between two nodes of the solution

We conclude that $|B(V)| = |X| + |Y| \leq 2\,|W(V)|$ as required. □

The path version of the problem $(MODNDC - P)$ is defined similarly and similar result is proven in [8].

IV-A better upper bound

Our main result is the following:

Theorem 1. *Given a solution S of $PMM(l)$,*

$$\epsilon(S) \leq \frac{1}{3/2(l+2)}.$$

Sketch of Proof.. The proof is based on a matching M leaving a small number of unmatched nodes. We partition the connected components of $\mathcal{G}_{S^*}$ as: (1) $\mathcal{I}$ which some maximum independent set of $\mathcal{OG}_S$, (2)$\mathcal{D} = \mathcal{OC}_S \setminus \mathcal{I}$, (3) $\mathcal{O}$ which is the set of all odd cycles of $\mathcal{G}_{S^*}$ except those in $\mathcal{OC}_S$, in other words all the odd cycles of $\mathcal{G}_{S^*}$ which intersect with P_0, (4) $\mathcal{E}$ the set of even cycles of $\mathcal{G}_{S^*}$, and (5)P_{S^*}, the set of maximal paths of $\mathcal{G}_{S^*}$.

Note that each cycle in $\mathcal{OC}_S = \mathcal{I} \uplus \mathcal{D}$ has at least $l+2$ nodes, because it is odd and it does not intersect with P_0.

We further partition these sets as: $\mathcal{I} = \mathcal{I}_1 \uplus \mathcal{I}_2 \uplus \mathcal{I}_D$, $\mathcal{D} = \mathcal{D}_1 \uplus \mathcal{D}_2$, $\mathcal{O} = \mathcal{O}_1 \uplus \mathcal{O}_2$,$\mathcal{E} = \mathcal{E}_D \uplus \mathcal{E}_2$.

Initially $\mathcal{I}_D = \mathcal{I}_2 = \mathcal{D}_2 = \mathcal{O}_2 = \mathcal{E}_2 = \emptyset$, thus $\mathcal{I}_1 = \mathcal{I}, \mathcal{D}_1 = \mathcal{D}, \mathcal{O}_1 = \mathcal{O}, \mathcal{E}_D = \mathcal{E}$, and M is the empty matching. The rest of the construction is done in nine phases, following a relatively long discussion, using, in particular, the combinatorial lemma 7. Due to space limitation it is omitted from this Extended Abstract. For a complete proof see [8]. □

V-A lower bound

Lemma 8. *There are infinitely many instances (G, P) and solutions S returned by PMM(l), such that*

$$\epsilon(S) = \frac{1}{2l+3}.$$

Proof. Consider the graph H containing a cycle H_1 of length $l+1$ and a cycle H_2 of length $l+2$. For each k consider an instance α such that $\mathcal{G}_\alpha$ consists of k copies of H and the conflict graph (not shown in the figure) contains all the possible edges except the edges of H and the chords of the cycles H_1 and H_2.

$\mathcal{G}_{S^*}$ consists of the k copies of H_1 and H_2.

Any cycle C of H with l nodes or less has at least four nodes, two from each of H_1 and H_2. At least two pairs of these nodes will be in conflict. Thus, there are no feasible cycles of length up to l. It follows that the algorithm will not make any changes during the preprocessing phase. The matching consisting of the $k(l+1)$ edges between the k copies of the cycles H_1 and H_2 is a maximum matching. If the algorithm finds this maximum matching in the first iteration, it will not be able to extend it in any manner in the next phase and the algorithm will terminate $\mathcal{G}_S$ being this maximum matching. We therefore have $d_0(S) = k, d_2(S) = 0, |\mathcal{P}_{S^*}| = 0, N = k(2l+3)$ and

$$\epsilon(S) = \frac{d_0(S) - d_2(S) - 2\,|\mathcal{P}_{S^*}|}{N} \frac{1}{2l+3}. \qquad \square$$

From Theorem 1 and Lemma 8 we get the following theorem as a corollary.

Theorem 2. *For any solution S returned by algorithm $PMM(l)$, $\epsilon(S) \leq \frac{1}{\frac{3}{2}(l+2)}$ and there are infinitely many instances for which $\epsilon(S) \geq \frac{1}{2l+3}$.*

4 Conclusion and Possible Improvements

We presented an improved analysis for the algorithm in [6] for a network of a general topology and proved $PMM(l) = OPT + \frac{1}{2}(1+\epsilon)N$, where $\frac{1}{2l+3} \leq \epsilon \leq \frac{1}{\frac{3}{2}(l+2)}$. For any given $\epsilon > 0$ this improves the analysis of the time complexity of the algorithm. In addition we use a novel technique in our analysis.

Open problems that are directly related to our work are (1) to further close the gap between the upper and lower bound, and (2) to extend to use of our technique to related problems. As we measure the performance of any algorithm ALG by $ALG \leq OPT + cN$ for some $0 < c < 1$, two other open problems are (3) to find an upper bound smaller than $c = 1/2$, and (4) to determine whether there exists a positive lower bound for c. Another open problem is (5) to improve the result of Lemma 7. For instance if the bound of Lemma 7 can be improved from $n/3$ to $n/2$, then it would imply $\epsilon \leq \frac{1}{\frac{5}{3}(l+2)}$.

References

1. O. Gerstel, P. Lin, and G. Sasaki. Wavelength assignment in a wdm ring to minimize cost of embedded sonet rings. In *INFOCOM'98, Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 69–77, 1998.
2. G. Călinescu and P-J. Wan. Traffic partition in wdm/sonet rings to minimize sonet adms. *Journal of Combinatorial Optimization*, 6(4):425–453, 2002.
3. M. Shalom and S. Zaks. A $10/7 + \epsilon$ approximation scheme for minimizing the number of adms in sonet rings. In *First Annual International Conference on Broadband Networks, San-José, California, USA*, October 2004.
4. L. Epstein and A. Levin. Better bounds for minimizing sonet adms. In *2nd Workshop on Approximation and Online Algorithms, Bergen, Norway*, September 2004.
5. T. Eilam, S. Moran, and S. Zaks. Lightpath arrangement in survivable rings to minimize the switching cost. *IEEE Journal of Selected Area on Communications*, 20(1):172–182, Jan 2002.
6. G. Călinescu, O. Frieder, and P.-J. Wan. Minimizing electronic line terminals for automatic ring protection in general wdm optical networks. *IEEE Journal of Selected Area on Communications*, 20(1):183–189, Jan 2002.
7. M. Flammini, M. Shalom, and S. Zaks. On minimizing the number of adms - tight bounds for an algorithm without preprocessing. In *The Third Workshop on Combinatorial and Algorithmic Aspects of Networking (CAAN), Chester, UK*, July 2006.
8. M. Flammini, M. Shalom, and S. Zaks. On minimizing the number of adms in a general topology optical network. In *Technion, Faculty of Computer Science, Technical Report CS-2006-14*, July 2006.

Robust Network Supercomputing with Malicious Processes[*]

Kishori M. Konwar, Sanguthevar Rajasekaran, and Alexander A. Shvartsman

Department of Computer Science and Engineering, University of Connecticut,
371 Fairfield Road, Unit 2155, Storrs, CT 06269, USA
{kishori, rajasek, aas}@cse.uconn.edu

Abstract. Internet supercomputing is becoming a powerful tool for harnessing massive amounts of computational resources. However in typical master-worker settings the reliability of computation crucially depends on the ability of the master to depend on the computation performed by the workers. Fernandez, Georgiou, Lopez, and Santos [12,13] considered a system consisting of a master process and a collection of worker processes that can execute tasks on behalf of the master and that may act maliciously by deliberately returning fallacious results. The master decides on the correctness of the results by assigning the same task to several workers. The master is charged one work unit for each task performed by a worker. The goal is to design an algorithm that enables the master to determine the correct result with high probability, and at the least possible cost. Fernandez *et al.* assume that the number of faulty processes or the probability of a process acting maliciously is known to the master. In this paper this assumption is removed. In the setting with n processes and n tasks we consider two different failure models, viz., model $\mathcal{F}_a$, where f-fraction, $0 < f < \frac{1}{2}$, of the workers provide faulty results with probability $0 < p < \frac{1}{2}$, given that the master has no *a priori* knowledge of the values of p and f; and model $\mathcal{F}_b$, where at most f-fraction, $0 < f < \frac{1}{2}$, of the workers can reply with arbitrary results and the rest reply with incorrect results with probability p, $0 < p < \frac{1}{2}$, but the master knows the values of f and p. For model $\mathcal{F}_a$ we provide an algorithm—based on the Stopping Rule Algorithm by Dagum, Karp, Luby, and Ross [10]—that can estimate f and p with (ϵ, δ)-approximation, for any $0 < \delta < 1$ and $\epsilon > 0$. This algorithm runs in $O(\log n)$ time, $O(\log^2 n)$ message complexity, and $O(\log^2 n)$ task-oriented work and $O(n \log n)$ total-work complexities. We also provide a randomized algorithm for detecting the faulty processes, i.e., identifying the processes that have non-zero probability of failures in model $\mathcal{F}_a$, with task-oriented work $O(n)$, and time $O(\log n)$. A lower bound on the total-work complexity of performing n tasks correctly with high probability is shown. Finally, two randomized algorithms to perform n tasks with high probability are given for both failure models with closely matching upper bounds on total-work and task-oriented work complexities, and time $O(\log n)$.

Keywords: Distributed algorithms, fault-tolerance, randomized algorithms, reliability, Internet supercomputing.

[*] This work is supported in part by the NSF Grants 0121277 and 0311368.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 474–488, 2006.

1 Introduction

With the advent of high bandwidth Internet connections, Internet supercomputing is increasingly becoming a popular means for harnessing the computing power of an enormous number of processes around the world. Internet supercomputing comes at a cost substantially lower than acquiring a supercomputer or building a cluster of powerful machines. Several Internet supercomputers are in existence today [1,2,3]. For instance, Internet PrimeNet Server, a project comprised of about 30,000 servers, PCs, and laptop computers, supported by Entropia.com, Inc., is a distributed, massively parallel mathematics research Internet supercomputer. PrimeNet Server has sustained throughput of over 1 teraflop. Another popular Internet supercomputer, the SETI@home project, also reported its speed to be in teraflops [18]. Such Internet supercomputers consist of a *master* computer or server and a large number of computers called *workers*, working for the master. The tasks to be carried out are submitted to the master computer; the worker computers subsequently download the tasks (e.g., [1]). After completing a downloaded task the worker returns the result to the master, and then proceeds to download another task. Tasks distributed by the master are typically independent. One of the major concerns involved in such computing environments is the reliability of the results returned by the workers. While most participating computers may be reliable, a large number of the workers have been known to return incorrect results for various reasons. Workers may return incorrect results due to unintended failures caused, for example, by over-clocked processes, or they may claim to have performed assigned work so as to obtain incentives, such as getting higher rank on the SETI@home list of contributed units of work.

Related Work. Several schemes have been proposed to improve the quality of the results obtained from untrusted workers. In this work we consider the problem of a distributed system consisting of a fail-free master process and a collection of worker processes that can execute tasks on the master's behalf as in the model introduced by Fernandez *et al.* [12,13]. They assumed that the worker processes might act maliciously and hence deliberately return incorrect results. The tasks are carried out independently at different processes and the master is charged with one work unit by the worker process for each task they do on behalf of the master. Due to the unreliable nature of the workers, the master cannot count on the correctness of the result returned by only one worker. Instead the master assigns the work to several processes and then tries to infer the correct result from the returned results. Thus the goal is to design algorithms that enable the master to accept correct results with high probability while minimizing the work charged by the workers.

Gao and Malewicz [14] considered the problem of minimizing the expected number of correct results of dependent tasks computed unreliably. There a distributed system is composed of a reliable server that coordinates the computation of unreliable workers, where any worker computes correctly with probability $p < 1$. Any incorrectly computed task corrupts all dependent tasks. They posed the problem of computing a schedule that determines which tasks should be performed by the (reliable) server and which by the (unreliable) workers, and when, so as to maximize the expected number of correct results under a constraint on the computation time.

Paquette and Pelc [20] consider a general model of a fault-prone system in which a decision has to be made on the basis of unreliable information. They assume that a Boolean value is conveyed to the deciding agent by several processes. An *a priori* probability distribution of this value is known to the agent and can be any arbitrary distribution. However, if the agent does not have any information on the distribution, it may assume that it is uniform. Relaying processes are assumed to fail independently (and independent of the choice of the value that is to be transmitted) with a known (but arbitrary and not necessarily equal) probability distribution. Fault-free processes relay the correct value but faulty ones may behave arbitrarily, in a Byzantine way. The deciding agent receives the vector of relayed values and must make a decision concerning the original value. The aim is to design a deterministic decision strategy with the highest possible probability of correctness. Such a deterministic decision strategy is presented, and it is shown that a locally optimal decision strategy need not have the highest probability of correctness globally.

Contributions. We consider a model commonly used in the Internet supercomputing where a master process supervises the execution of many independent tasks on a large set of worker processes. For simplicity we assume that the number of tasks is n and the number of processes is also n. The efficiency of algorithms is measured in terms of *work* complexity. Two definitions of work occur in the literature. The first counts only the work spent on executing tasks, including multiplicities [11]. This is termed *task-oriented work*, and it does not account for idling or waiting processes; we denote it here as $\underline{work}$. The second definition accounts for *total work*, including idling and waiting [9]; we denote it here as $\overline{work}$ (our choice of notation is due to the fact that $\underline{work} \leq \overline{work}$).

We study two failure models in a synchronous environment (Section 2). The first, called $\mathcal{F}_a$, has f-fraction, $0 < f < \frac{1}{2}$ of the workers providing faulty results with probability $0 < p < \frac{1}{2}$, and the master does not have *a priori* knowledge of f and p. The second, called $\mathcal{F}_b$, has at most f-fraction of the workers replying with arbitrary (possibly incorrect) results and the rest replying with incorrect results with probability p, and the master knows the exact values of f and p (this is similar to the model of Fernandez *et al.* [12,13], although our focus is on the asymptotic analysis of work complexity).

For the model $\mathcal{F}_a$, where the master has no knowledge of the model parameters, we provide an algorithm that can estimate the quantities f and p up to a customized degree of accuracy (Section 3). The algorithm has time complexity of $O(\log n)$, $\underline{work}$ complexity of $O(\log^2 n)$, $\overline{work}$ complexity of $O(n \log n)$, and message complexity of $O(\log^2 n)$. Next, we provide an algorithm to detect the possibly faulty processes in failure model $\mathcal{F}_a$ with $\underline{work}$ complexity $O(n)$ and time complexity $O(\log n)$ (Section 4).

Finally, we provide algorithms in failure models $\mathcal{F}_a$ and $\mathcal{F}_b$ to perform n tasks (Section 5). For model $\mathcal{F}_a$ we obtain optimal $\underline{work}$ complexity $O(n)$ and time $O(\log n)$ (Section 5.1). We also demonstrate that such optimal complexity (linear in n) is not possible for $\overline{work}$ complexity, with high probability; and we provide an algorithm for failure model $\mathcal{F}_b$ with $O(n \log n)$ $\overline{work}$ complexity and time $O(\log n)$ (Section 5.2). The above bounds are claimed with high probability.

Other related work. A fundamental problem in the study of complexity of fault-tolerant distributed computation is the Do-All problem. The Do-All problem involves using n processes to cooperatively perform m independent tasks in the presence of failures. Many algorithms have been developed for Do-All in various models of computation, including message-passing [11,21], partitionable networks, and shared-memory models [16,19] under a variety of failure models.

Process groups in distributed systems and services may rely on failure detectors to detect failed processes *completely* and *efficiently*. *Completeness* is the assurance that the faulty process will eventually be detected by every non-faulty process. *Efficiency* means the failed processes are detected *quickly* as well as *accurately*, without making mistakes. Chandra and Toueg [7] showed that it is impossible for a failure detector algorithm to deterministically achieve both completeness and accuracy over an asynchronous network. This result led researchers to look for failure detectors that observes completeness deterministically and efficiency probabilistically [4,5,6,8]. Probabilistic network models have been used to analyze "heartbeat" failure detectors but only with a *single* process detecting failure of a *single* other process.

Document Structure. Section 2 describes the model of computation and the failure models. Section 3 discusses the algorithms for the detection of the parameters p and f based on the Stopping Rule Algorithm. Section 4 presents and analyzes an algorithm that detects faulty processes. In Section 5 we provide algorithms, for failure models $\mathcal{F}_a$ and $\mathcal{F}_b$ for performing n tasks. Section 6 concludes the paper and discusses future work. An online technical report [17] contains the omitted proofs.

2 Models of Computation

We assume a synchronous model where processes communicate by exchanging authenticated messages. The problem is to perform a collection of similarly-sized independent tasks. The tasks are idempotent, meaning that a task can be correctly performed one or more times with the same results. The processes are subject to Byzantine failures in that a process can maliciously return incorrect results for any task. One distinguished infallible process M is identified as the *master* process. The master has n uniquely identified tasks to perform and for each task it needs to collect the result. We assume that it is not feasible for M to perform all tasks by itself (for lack of resources or paucity of time). Consequently, M supervises a set of n processes called *workers*. Workers have unique identifiers from the set $P = \{1, ..., n\}$. The master follows the algorithmic template for executing each task T as shown in Fig. 1 (cf. [12,13]).

```
procedure for master process $M$ and task $T$:
    Choose a set $S \subseteq P$
    Send task $T$ to each process $s \in S$
    Wait for the results from the processes in $S$
    Decide on the result value $v$ from the responses

procedure for each worker $w \in P$:
    Wait to receive a task from the master process $M$
    Upon receiving a task from $M$
        Execute the task
        Send the result to $M$
```

Fig. 1. The algorithmic template for the master and worker processes

Models of failure. The workers are subject to Byzantine failures, with the restriction that a faulty process cannot impersonate another process and cannot tamper with messages. We consider two worker failure models, $\mathcal{F}_a$ and $\mathcal{F}_b$:

Model $\mathcal{F}_a$

(i) *f-fraction, $0 < f < \frac{1}{2}$, of the n workers may fail.*

(ii) *Each possibly faulty worker independently exhibits faulty behavior with probability $0 < p < \frac{1}{2}$.*

(iii) *The master has no a priori knowledge of f and p.*

Model $\mathcal{F}_b$

(i) *There is a fixed bound on the f-fraction, $0 < f < \frac{1}{2}$, of the n workers that can be faulty, and any worker from the remaining $(1-f)$-fraction of the workers fails (i.e., returns incorrect answers) with probability p, $0 < p < \frac{1}{2}$, independently of other workers.*

(ii) *The master knows the values f and p.*

Remark. Our failure models are more restrictive than the common Byzantine model where any process may fail and subsequently behave in a malicious manner. In our models probabilistic constraints limit the ability of processes to behave in a Byzantine way, however when a process is "allowed" to fail, the failure can indeed be Byzantine. *End remark.*

The failure model $\mathcal{F}_b$ is somewhat similar to the behavior considered in the work of Fernandez *et al.* [12,13]. In contrast to the model $\mathcal{F}_b$ they assume that there is a known probability d of the master process receiving the reply from a given worker (that is willing to reply) withing time t after sending the task to the worker. However, they do not specify the distribution and subsume the factor $0 < d < 1$ in the probability of failure, p. Here, we assume $1 < p < \frac{1}{2}$ in order to conform to the failure model considered in [12,13]. Finally, whereas [12,13] study the conditions leading to the successful executions of tasks, in our work we focus on the analysis of work complexity as defined below.

Measures of efficiency. To evaluate the work-efficiency of the algorithms, we use two popular ways of accounting for work: total work and task-oriented work. In both cases we assume that it takes one fixed time step to perform a unit of work, and that it takes one unit of work to perform one task. Chlebus *et al.* [9] use *total work complexity* (a.k.a. *available processor steps*). Here all steps taken by the processes during the execution are counted, including the steps of the idling and waiting non-faulty processes. We refer to this measure as $\overline{work}$ complexity.

Dwork *et al.* [11] define work as the number of performed tasks, counting multiplicities. This approach does not charge for the steps spent by processes while idling or waiting for messages (motivated by the possibility that processes that are not performing tasks can be gainfully employed to work on other problems). We call this task-oriented measure $\underline{work}$ complexity. (Note that it trivially follows that $\underline{work} \leq \overline{work}$, hence our choice of notation.)

We also consider the conventional notions of *time complexity* and *message complexity*, where the latter is the number of point-to-point messages sent during the execution of an algorithm.

Lastly, we use the commonly used definition of *an event $\mathcal{E}$ occurs with high probability* (w.h.p.) to mean that $\mathbf{Pr}[\mathcal{E}] = 1 - O(n^{-\alpha})$ for some constant $\alpha > 0$.

3 Estimation of the Fault Parameters in Model $\mathcal{F}_a$

In the failure model $\mathcal{F}_a$ the parameter values f and p are unknown to the master process. Knowledge of f and p is instrumental in inferring, with high probability, the correct results based on the computations done by the workers. In order to estimate the values of f and p we provide an algorithm based on the Stopping Rule Algorithm of Dagum *et al.* [10]. The motivation for this algorithm comes from the following example. Suppose we have a random variable X, where $X \in \{0, 1\}$, such that $\mathbf{Pr}[X = 0] = p$ and $\mathbf{Pr}[X = 1] = 1 - p = q$. Consider the independent and identically distributed (iid) random variables $X_1, X_2, \cdots, X_m$ whose distribution is that of X. Therefore, $\mathbb{E}X = \mathbb{E}X_1 = \cdots = \mathbb{E}X_m = q$. Suppose we want to use the unbiased estimator $\frac{S}{m}$ of q, where $S_m = X_1 + X_2 + \cdots + X_m$. Now, suppose we choose $m = c \log n$, for some $c > 0$, in an attempt to have a reasonable number of trials for our setting of n processes with n tasks. Now by a simple application of a slight variation of the Chernoff bounds result (see Lemma 1 below) we can show that for $\delta > 0$

$$\mathbf{Pr}\left[\frac{S_m}{m} \geq (1+\delta)q\right] \leq e^{-\frac{\quad 2}{4}} \leq e^{-\frac{2 \quad \log}{4}} \leq n^{-\frac{\quad 2}{4}}$$

A similar relation can be shown for the case $\mathbf{Pr}[\frac{S}{m} \leq (1-\delta)q] \leq n^{-\frac{\quad 2}{4}}$. Observe that unless we have some prior information about the value of q (or p) we many not know what c to choose in order to determine the number of repetitions in order to obtain a desired level of accuracy for the estimation of p. Thus it is desirable to have an algorithm that has an online rule for stopping the computation.

Lemma 1 (Chernoff Bounds). *Let $X_1, X_2, \cdots, X_n$ be n independent Bernoulli random variables with $\mathbf{Pr}[X_i = 1] = p_i$ and $\mathbf{Pr}[X_i = 0] = 1 - p_i$, then it holds for $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X] = \sum_{i=1}^{n} p_i$ that for all $\delta > 0$, (i) $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\frac{{}^{2}}{3}}$, and (ii) $\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{{}^{2}}{2}}$.*

Estimation of f and p. Under model $\mathcal{F}_a$ the n tasks can be done in the following two phases, viz., first estimate the value of p and f followed, second, by the actual computation of the tasks by the workers. At first we provide algorithms to estimate the values of f and p in time $O(\log n)$ and closely matching upper bounds on $\underline{work}$ and $\overline{work}$ complexities. The basic idea is to ask some of the worker processes to compute the result of a task whose result is already known to the master M. Based on the responses we can estimate the value of f and p with high probability (this is shown later in detail in Fig. 3). However, soon we will see that this is not a trivial task and is somewhat counter-intuitive.

Let Z be a random variable distributed in the interval $[0,1]$ with mean μ_Z. Let $Z_1, Z_2, \ldots$ be independently and identically distributed according to Z variables. Our algorithm is based on the *Stopping Rule Algorithm* [10] for estimating the mean μ_Z, and we refer to this algorithm as *SRA* and give it below in Fig. 2 for completeness. We say that $\tilde{\mu}_Z$ is an (ϵ, δ)-approximation of μ_Z if $\mathbf{Pr}[\mu_Z(1-\epsilon) \leq \tilde{\mu}_Z \leq \mu_Z(1+\epsilon)] > 1 - \delta$.

```
Input Parameters: $(\epsilon, \delta)$ with $0 < \epsilon < 1$, $\delta > 0$
Let $\Gamma_1 = 1 + (1+\epsilon)\Gamma$     // $\lambda = (e-2) \approx 0.72$ and $\Gamma = 4\lambda \log(\frac{2}{\delta})/\epsilon^2$
Initialize $N \leftarrow 0, S \leftarrow 0$
While $S < \Gamma_1$ do : $N \leftarrow N + 1$; $S \leftarrow S + Z_N$
Output: $\tilde{\mu}_Z \leftarrow \frac{\Gamma_1}{N}$
```

Fig. 2. The Stopping Rule Algorithm (*SRA*) for estimation of μ_Z

Let us define $\lambda = (e-2) \approx 0.72$ and $\Gamma = 4\lambda \log(\frac{2}{\delta})/\epsilon^2$. Now, Theorem 1 (slightly modified, from [10]) tells us that *SRA* provides us with a (ϵ, δ)-approximation with the number of trials within $\frac{\Gamma_1}{\mu}$ with high probability, where $\Gamma_1 = 1 + (1+\epsilon)\Gamma$.

Theorem 1 (Stopping Rule Theorem). *Let Z be a random variable in $[0,1]$ with $\mu_Z = \mathbb{E}[Z] > 0$. Let $\tilde{\mu}_Z$ be the estimate produced and let N_Z be the number of experiments that SRA runs with respect to Z on input ϵ and δ. Then, (i) $\mathbf{Pr}[\mu_Z(1-\epsilon) \leq \tilde{\mu}_Z \leq \mu_Z(1+\epsilon)] > 1 - \delta$, (ii) $\mathbb{E}[N_Z] \leq \frac{\Gamma_1}{\mu}$, and (iii) $\mathbf{Pr}[N_Z > (1+\epsilon)\frac{\Gamma_1}{\mu}] \leq \frac{\delta}{2}$.*

We present an algorithm for estimating f and p in model $\mathcal{F}_a$ in Fig. 3; we call it algorithm $\mathcal{A}_{f,p}$. Here $(1-f)$–fraction of the workers are non-faulty and they reply with correct answers. On the other hand, every possibly-faulty worker replies with a wrong answer with probability $p > 0$ and independently of other

```
    **Input Parameters:** $(\epsilon, \delta)$ with $0 < \epsilon < 1$, $\delta > 0$
    **procedure** for master process $M$:
1:      Let $\Gamma_1 = 1 + (1 + \frac{\epsilon}{2})\Gamma$
2:      $N_f \leftarrow 0$;  $S_f \leftarrow 0$     // initialize
3:      **While** $S_f < \Gamma_1$ **do**
4:          Choose a subset of workers $P' \subseteq P$, with replacement s.t. $|P'| = \lceil \log n \rceil$;
5:          **Foreach** $s \in P'$ **do** $N_p^s \leftarrow 0$;  $S_p^s \leftarrow 0$ ;    //initialize for all $s \in P'$
6:          **While** $\min_{s \in P'}\{S_p^s\} < \Gamma_1$ **do**
7:              **Foreach** $s \in P'$, s.t., $S_p^s < \Gamma_1$ **do**
8:                  Send a "test" task to $s$ ;
9:                  Receive result from $s$ ;
10:                 **If** result is "incorrect" **then** $S_p^s \leftarrow S_p^s + 1$;
11:                 $N_p^s \leftarrow N_p^s + 1$ ;
12:             **End for**
13:         **End while**
14:         **Foreach** $s \in P'$ **do**
15:             **If** $N_p^s \neq S_p^s$ **then** $S_f \leftarrow S_f + 1$;  $\hat{p}_s \leftarrow \frac{\Gamma_1}{N}$;
16:         **End for**
17:         $N_f \leftarrow N_f + |P'|$ ;
18:     **End while**
19:     $\hat{f}_s \leftarrow \frac{\Gamma_1}{N}$
20:     Return $\hat{f}$ and any of the $\hat{p}_s$s;
```

Fig. 3. Algorithm $\mathcal{A}_{f,p}$ for model $\mathcal{F}_a$ to estimate the values of f and p

tasks assigned to it. Therefore, the master process uses some "test" tasks the answers to which it already knows. If the master gets a wrong reply from a worker to a "test" task, then it can decide that the worker is faulty. However, our goal is to design an (ϵ, δ)-approximation algorithm that can help us estimate f and p with reasonably low work and message complexities.

The basic idea of the algorithm is as follows: choose randomly $O(\log n)$ workers, with repetition, and ask each of the chosen workers $O(\log n)$ "test" questions to decide whether or not it is faulty. Now based on the Stopping Rule Theorem we use the $O(\log n)$ "test" responses to estimate p, i.e., compute $\hat{p}$, and we use the $O(\log n)$ decisions on whether or not the chosen workers are faulty or not to estimate f, i.e., compute $\hat{f}$.

In algorithm $\mathcal{A}_{f,p}$ this is implemented by choosing a subset $P' \subseteq P$ of $\log n$ workers in line 4 and this is repeated until the stopping rule for the estimation of f is reached. In line 5 we initialize two counters N_p^s and S_p^s for every $s \in P'$ used for the estimation of p. The **While** loop in lines 6–13 is run until each of the processes reaches the stopping rule individually. If the condition for stopping the **While** loop in line 6 is not satisfied then $\min_{s \in P'}\{S_p^s\} \geq \Gamma_1$, i.e., for every $s \in P'$, $S_p^s \geq \Gamma_1$, then each process $s \in P'$ reaches its corresponding stopping condition. Observe that the random variable Z in the Stopping Rule Algorithm corresponds to "correct" or "incorrect" results i.e., $Z = 0$ or $Z = 1$, respectively.

We now state the following result.

Theorem 2. *Algorithm $\mathcal{A}_{f,p}$ is an (ϵ, δ)–approximation algorithm, $0 < \epsilon < 1$, $\delta > 0$, for the estimation of f and p with $\underline{\textit{work}}$ complexity $O(\log^2 n)$, $\overline{\textit{work}}$ complexity $O(n \log n)$, message complexity $O(\log^2 n)$, and time complexity $O(\log n)$, with high probability.*

From the above result we conclude that given any $\delta > 0$, $0 < \epsilon < 1$ algorithm $A_{f,p}$ can provide the (ϵ, δ)-approximation to the estimation of f and p in $O(\log n)$ time. Of course, its time increases as the factor $(1 + \frac{1}{\epsilon^2})$ with the decrease of ϵ, i.e., when higher accuracy is demanded (as given by inequalities (**??**) and (**??**) in the proof of Theorem 2 in the appendix).

4 Detection of Faulty Processors

In this section we deal with the detection of faulty processes among the initial set of n processes in the failure model $\mathcal{F}_a$. Although detection of faulty processes is instrumental in avoiding redundant computations, this is also interesting as a method of computation in the failure mode $\mathcal{F}_a$. Furthermore, under certain computing environments, once such faulty processes are detected the master process may choose to stop assigning any tasks to them. For simplicity we suppose that all n processes are faulty and therefore each will provide wrong results with probability p if it is asked a "test" question. We propose an algorithm in Fig. 4 based on the fact that if we send out the "test" messages to the n workers then $p \cdot n$ of them will be expected to be diagnosed as faulty, with high probability, based on their wrong answers. Then in the following round we send the test messages to the remaining processes and expect a fraction p of them to be diagnosed faulty. Proceeding this way we can expect to reduce the number of "test" messages sent out by a factor of p in each round. Thus, to detect all the faulty processes the algorithm would take $O(\log n)$ rounds and and $\underline{\textit{work}}$ complexity $O(n)$ w.h.p., which in fact asymptotically matches with the lower bound $\Omega(n)$.

```
procedure for master process $M$:
    Initially, $F \leftarrow \emptyset$:
  For $t = 0, \cdots, k \log n$, $k > 0$
    Choose a set $S \leftarrow P \setminus F$
    Send each process $s \in S$ the task
    Wait for the replies from the processes in $S$
    $F \leftarrow F \cup \{s : s \text{ is a faulty process}\}$
  End For
```

Fig. 4. Algorithm for detecting faulty processes

First we mention here a result of Karp [15] on probabilistic recurrence relations that is helpful in our analysis of the algorithm. Consider the probabilistic recurrence relation of the type $T(x) = a(x) + T(h(x))$, where $a(\cdot)$ is a non-negative real-valued function of x and $h(x)$ is a random variable ranging over $[0, x]$ and having expectation less than or equal to $m(x)$, where $m(\cdot)$ is a non-negative real-valued function.

Theorem 3 ([15]). *Suppose that $a(x)$ is a nondecreasing, continuous function that is strictly increasing on $\{x|a(x) > 0\}$, and $m(x)$ is a continuous function. Then for every positive real x and every positive integer t,*

$$\mathbf{Pr}[T(x) > u(x) + ta(x)] \leq \left(\frac{m(x)}{x}\right)^t$$

where $u(x) = \sum_{i\geq 0} a(m^i(x))$ is the solution to the equation $u(x) = a(x) + u(m(x))$ with $m^0(x) := 0$ and $m^{i+1} := m(m^i(x))$.

Lemma 2. *The algorithm in Fig. 4 detects all faulty processes among n processes in $O(\log n)$ time and with <u>work</u> $O(n)$ with high probability.*

Proof. We have to show that $S = \emptyset$ with high probability at the end of $O(\log n)$ rounds and <u>*work*</u> during this time is $O(n)$.

At any round t of the algorithm we denote by S_t the number of processes that are not diagnosed as faulty, i.e., the size of the set S. Clearly, $S_0 = n$. Now, let us denote by A_t the event $A_t = \{|S_t - qS_{t-1}| \leq \delta q S_{t-1}\}$ for $t = 1, 2, \ldots$, where $\delta > 0$, and let $q = 1 - p$. We consider and analyse the following two phases.

Phase 1: $S_t > \frac{n}{\log n}$**.** First we want to prove that at every round t, where $t = 1, 2, \ldots, k \log\log n$, with k a constant, the number of "possibly non-faulty" processes decreases by a factor of p. That is, we are interested in computing the probability of the event $A := \bigcap_{t=1}^{k\log\log n} A_t$ and we use T to stand for $k \log\log n$. Therefore, we are expressing them as conditional probabilities

$$\mathbf{Pr}(A) = \prod_{t=1}^{T} \mathbf{Pr}(A_t|A_1 \cdots A_{t-1})$$

Now, observe the fact that $\mathbb{E}[S_t] = qS_{t-1}$ and $A_1 \cap A_2 \cap \cdots \cap A_t$ implies that $((1-\delta)q)^t S_0 \leq S_t \leq ((1+\delta)q)^t S_0$, i.e., $((1-\delta)q)^t n \leq S_t \leq ((1+\delta)q)^t n$. Considering this and by applying Chernoff bounds we can show that

$$\mathbf{Pr}(A_t|A_1 \cdots A_{t-1}) = 1 - \mathbf{Pr}(\bar{A}_t|A_1 \cdots A_{t-1}) \geq 1 - 2e^{-\frac{{}^{2}\qquad{}^{-1}}{4}}$$

and hence

$$\begin{aligned}\mathbf{Pr}(A) &\geq \textstyle\prod_{t=1}^{T}\left(1 - 2e^{-\frac{{}^{2}\qquad{}^{-1}}{4}}\right) \geq \textstyle\prod_{t=1}^{T}\left(1 - 2e^{-\frac{{}^{2}((1-\;\;)\;\;)}{4}}\right)\\ &\geq \left(1 - 2e^{-\frac{{}^{2}((1-\;\;)\;\;)}{4}}\right)^{T} \geq \left(1 - 2e^{-\frac{2}{\log}}\right)^{k\log\log n}\\ &\geq 1 - 4\frac{k\log\log n}{e^{\,2\;\;\overline{\log}}}\end{aligned}$$

In the above algebra we used the fact that $T = k \log\log n$ and hence $\frac{1}{((1-\delta)q)}$ $= O(\log n)$, where $0 < \delta < 1$. Therefore, we estimate <u>*work*</u> W_1 done during the rounds $t = 1, 2, \ldots, T$ to be bounded as $W_1 \leq \sum_{t=1}^{T} ((1-\delta)q)^t n = O(n)$.

Phase 2: $S_t < \frac{n}{\log n}$. Observe that this phase can last at most $O(\log n)$ rounds with high probability because the probability that any faulty process has not been diagnosed as faulty for $k \log n$ is $\frac{1}{n}$ for some $k > 1$. In this phase we want to estimate $\underline{work}$ done by the algorithm for the case when $|S_t| < \frac{n}{\log n}$. Let $W(x)$ denote the amount of work till the end of the algorithm (i.e., until $S = \emptyset$) starting with the round when $S < \frac{n}{\log n}$. Now, we can express $W(x)$ as a probabilistic recurrence relation:

$$W(x) = x + W(h(x))$$

where $\mathbb{E}h(x) = qx$ and hence we identify $m(x)$ as qx. Using Theorem 3 we get

$$\mathbf{Pr}(W(x) \geq u(x) + tx) \leq \left(\frac{m(x)}{x}\right)^t$$

Here we have $u(x) = \sum_{i \geq 0} a(m^i(x)) = \sum_{i \geq 0} q^i x < cx$, where $c = \frac{1}{1-q}$. Now, choosing $t = k' \log n$ and $x = \frac{n}{\log n}$ we have

$$\mathbf{Pr}\left[W\left(\frac{n}{\log n}\right) \geq u\left(\frac{n}{\log n}\right) + k' \log n \frac{n}{\log n}\right] \leq \left(\frac{m(x)}{x}\right)^t \leq q^{k' \log n} \leq \frac{1}{n^{k''}}$$

where k'' is some positive constant and after simplification we get

$$\mathbf{Pr}\left[W\left(\frac{n}{\log n}\right) \geq c'n\right] \leq \frac{1}{n^c}$$

Hence, $\underline{work}$ W_2 done in the second phase is $W_2 = W\left(\frac{n}{\log n}\right) = O(n)$ with high probability.

From the analyses of the two phases, the overall $\underline{work}$ done during the execution of the algorithm is $W = W_1 + W_2 = O(n)$ with high probability. □

5 Performing *n* Tasks with *n* Workers with Fraction *f* of them Faulty with Probability *p*

The ultimate goal of our computational model is to be able to perform all n tasks in either of the failure models $\mathcal{F}_a$ and $\mathcal{F}_b$. Ideally, all n tasks should be done in $O(n)$ $\underline{work}$ and $\overline{work}$ complexities. However, we show that it is not possible to do all n tasks in $O(n)$ $\overline{work}$ with high probability.

Lemma 3. *It is not possible to perform all n tasks correctly, in the failure model $\mathcal{F}_a$, with linear $\overline{work}$ complexity (i.e., $O(n)$) with high probability.*

5.1 Performing Tasks in Failure Model $\mathcal{F}_a$

Discouraged by the result of Lemma 3 for $\overline{work}$ complexity, now we look for the possibility of an algorithm that does n tasks correctly with $O(n)$ $\underline{work}$ and $O(\log n)$

time, with high probability. We provide an algorithm that has $\underline{work}$ complexity $O(n)$, $\overline{work}$ complexity $(n \log n)$, and time $O(\log n)$, with high probability. Recall that in the context of failure mode $\mathcal{F}_a$, among the n workers, f-fraction of workers may provide faulty results with probability p. We assume that the local processing time of the master is negligible. The intuitive idea of the algorithm is to first randomly decide on a set S of $\frac{n}{\log n}$ worker processes and send each of them $O(\log n)$ "test" tasks to compute (a "test" task is a task whose result is already known to the master). Now because of the nature of the failure model $\mathcal{F}_a$ (i.e., the probability of a possibly faulty worker returning erroneous results is independent of other workers) depending on the responses from the workers about the results of these "test" tasks the master process would be able to deduce some (as discussed in the following) of the faulty processes. As a result, with high probability we will detect $g\frac{n}{\log n}$ distinct non-faulty workers, where $g := 1 - f$. Now, we get our n tasks done by these processes by sending to these $g\frac{n}{\log n}$ workers $g\frac{n}{\log n}$ tasks in $O(\log n)$ rounds. Finally, we show that this algorithm runs in $\log n$ time and has $\underline{work}$ $O(n)$. The algorithm is given in Fig. 5 and the following result can be proved for the algorithm whose proof is omitted due to paucity of space.

```
procedure for master process $M$:
    Initially, $C \leftarrow \emptyset$, $J \leftarrow$ set of $n$ tasks
    Choose randomly a subset of workers $S$, possibly with repetition,
      $S \subseteq P$, s.t. $|S| = k\frac{n}{\log n}$ workers $k > 0$ is a constant
  For $i = 1, \cdots, k' \log n$, $k' > 0$
      Send to each worker $s \in S$ a "test" task.
      Collect the responses from all the workers.
  End For
  If all the responses from a worker $s \in S$ are correct then
      $C \leftarrow C \cup \{s\}$
  End if
  For $i = 1, \cdots, \frac{n}{|C|}$
      Send $|C|$ jobs from $J$, not sent in a previous iteration,
        one to each worker in $C$.
      Collect the responses from the $C$ workers.
  End For
```

Fig. 5. Algorithm to perform n tasks with n workers with f-fraction of them faulty with probability p

Theorem 4. *The algorithm in Fig. 5 performs all n tasks in $O(\log n)$ time and has $\underline{work}$ $O(n)$ and $\overline{work}$ $O(n \log n)$.*

5.2 Performing Tasks in Failure Model $\mathcal{F}_b$

Now, we consider the setting of failure model $\mathcal{F}_b$ to complete n tasks using n processes. Below we provide a simple algorithm that completes all n tasks with high probability for $0 < p$, $f < \frac{1}{2}$, and $(1-f)(1-p) > \frac{1}{2}$. The basic idea is to distribute the tasks according to a random permutation from S_n, the set of all permutations of $[n] = \{1, 2, \cdots, n\}$. In other words, if in an iteration, the random permutation $\pi \in S_n$ is chosen, then the task j is sent to process $\pi(j)$.

```
    procedure for master process $M$:
 1:     For $i = 1, \cdots, k \log n$, for some constant $k > 0$
 2:         Choose a random permutation $\pi \in_R S_n$
 3:         Foreach $j \in [n]$
 4:             Send task $j$ to process $\pi(j)$
 5:         End For
 6:         Collect the responses from all the workers.
 7:     End For
 8:     Foreach task $j \in [n]$
 9:         Choose the majority of the results of computation, for task $j$, as the result.
10:     End For
```

Fig. 6. Algorithm to perform n tasks using n workers in failure model $\mathcal{F}_b$

Theorem 5. *The algorithm in Fig. 6 performs all n tasks correctly in $O(\log n)$ time and has $\overline{work}$ and $\underline{work}$ complexities $O(n \log n)$, for $0 < p, f < \frac{1}{2}$, and $(1-f)(1-p) > \frac{1}{2}$, with high probability.*

Proof. Let us denote by $T(i)$ the number of the majority results for task i as decided in line 9 of the algorithm. For instance, if i was tried 8 times and the results are a, a, c, a, a, b, a, b, then the fraction of the majority is $\frac{5}{8}$. We are interested in showing that for every task $j \in [n]$ the event $\{T(i) < (1-\delta)(1-f)(1-p)k \log n\}$, for $\delta > 0$, occurs with very low probability. To show that let us consider any task $i \in [n]$. We know that for one iteration of the loop in line 1

$$\mathbf{Pr}[\text{Task } i \text{ is computed correctly}] \geq (1-f)(1-p)$$

Now, since task i executed $k \log n$ times and these trials are done independently, by applying Chernoff bound we have

$$\mathbf{Pr}[T(i) < (1-\delta)(1-f)(1-p)k \log n] \leq \mathbf{Pr}[T(i) < (1-\delta)\mu] \leq e^{-\frac{2}{4}} \leq n^{-\frac{2}{8}}$$

where $\mu > k \log n(1-f)(1-p) > \frac{k}{2} \log n$. Now, we want to show that $\bigcup_{i=1}^{n} \{T(i) < (1-\delta)(1-f)(1-p)k \log n\}$ occurs with low probability and hence

$$\begin{aligned} &\mathbf{Pr}[\textstyle\bigcup_{i=1}^{n} \{T(i) < (1-\delta)(1-f)(1-p)k \log n\}] \\ &\qquad \leq \textstyle\sum_{i=1}^{n} \mathbf{Pr}[T(i) < (1-\delta)(1-f)(1-p)k \log n] \leq \frac{n}{n^{-\frac{2}{8}}} \leq \frac{1}{n} \end{aligned}$$

for some $\ell > \frac{k\delta^2}{8} - 1$. Now, since the **For** loop in line 1 is run $O(\log n)$ times and since at every iteration a task is sent to every process, then $\underline{work}$ and $\overline{work}$ complexities are both $O(n \log n)$, with high probability. □

6 Conclusion

In this paper we presented and discussed a model of computation abstracting the type of computation commonly used in Internet supercomputing where many of the worker machines may be faulty. The problem is to get n independent tasks done by a master process by harnessing the power of n workers. We assumed two different failure models, viz., one that is motivated by the model proposed by Fernandez *et al.* [12,13], we call it failure model $\mathcal{F}_b$, and a more realistic failure model that we call model $\mathcal{F}_a$. In model $\mathcal{F}_b$ it is assumed that the fraction of faulty processes is known and the probability of failure is known to the master process in advance. However, in model $\mathcal{F}_a$ we removed this assumption and instead provided algorithms that are used to estimate these quantities up to a customized degree of accuracy at a reasonably low cost in terms of time and work. We considered two ways in which the master process is charged by the workers: task-oriented work and total work. We provided algorithms to detect the possibly faulty processes in failure model $\mathcal{F}_a$. Finally, we provided algorithms to perform n tasks in models $\mathcal{F}_a$ and $\mathcal{F}_b$. For model $\mathcal{F}_a$ we obtain optimal $O(n)$ task-oriented work complexity (i.e., $\underline{work}$) and running time $O(\log n)$, w.h.p. We also demonstrated that such optimality is not possible in the total work measure (i.e., $\overline{work}$). For model $\mathcal{F}_b$ we obtain $O(n \log n)$ $\overline{work}$ and $\underline{work}$ complexities, and running time of $O(\log n)$, w.h.p.

Future work includes considering more virulent failure behaviors and task sets with inter-task dependencies.

Acknowledgement. The authors would like to thank Elaine L. Sonderegger and the anonymous reviewers for their valuable comments that helped improve the paper.

References

1. Distributed.net. http://www.distributed.net/.
2. Internet primenet server. http://mersenne.org/ips/stats.html.
3. Seti@home. http://setiathome.ssl.berkeley.edu/.
4. M.K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of 11th International Workshop on Distributed Algorithms (WDA97)*, pages 126–140, Septermber, 1997.
5. C. Almeida and P. Verissimo. Timing failure detection and real-time group communication in real-time systems. In *Proceedings of 8th Euromicro Workshop on Real-Time Systems*, June, 1996.
6. R. Bollo, J.P.L. Narzul, M Raynal, and F. Tronel. Probabilistic analysis of a group failure detection protocol. In *Proceedings of 4th International Workshop on Object-Oriented Real-Time Dependable Systems(WORDS'99)*, January 1999.
7. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):22i5–267, March, 1996.
8. W. Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. In *Proceedings of 30th International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, June, 2000.

9. B.S. Chlebus, R. De Prisco, and A. A Shvartsman. Performing tasks on restartable message-passing processors. In *Proceedings of the WDAG*, pages 96–110, 1997.
10. P. Dagum, R.M. Karp, M. Luby, and S. Ross. An optimal algorithm for monte carlo estimation. In *Proceedings of the Foundations of Computer Science*, pages 142–149, 1995.
11. C. Dwork, J. Y. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 91–102, 1992.
12. A. Fernandez, Ch. Georgiou, L. Lopez, and A Santos. Reliably executing tasks in the presence of malicious processors. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 490–492, 2005.
13. A. Fernandez, Ch. Georgiou, L. Lopez, and A Santos. Reliably executing tasks in the presence of malicious processors. Technical Report Numero 9 (RoSaC-2005-9), Grupo de Sistemas y Comunicaciones, Universidad Rey Juan Carlos, 2005. http://gsyc.escet.urjc.es/publicaciones/tr/RoSaC-2005-9.pdf.
14. L. Gao and G. Malewicz. Toward maximizing the quality of results of dependent tasks computed unreliably. *Theory of Computing Systems*, to appear (preliminary version OPODIS'04).
15. R.M. Karp. Probabilistic recurrence relations. *Journal of the Association for Computing Machinery*, 41(6):1136–1150, 1994.
16. Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis. In *Combining tentative and definite executions for dependable parallel computing*, pages 381–390, 1991.
17. K.M. Konwar, S. Rajasekaran, and A.A. Shvartsman. Robust network supercomputing with malicious processes, 2006. http://www.cse.uconn.edu/~kishori/KRS2006.pdf.
18. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. Seti@home - massively distributed computing for seti. *Computing in Science & Enginering*, 3(1), 2001.
19. C. Martel and R. Subramonian. On the complexity of certified write-all algorithms. *Journal of Algorithms*, 16(3):361–387, 1994.
20. M. Paquette and A Pelc. Optimal decision strategies in byzantine enviornments. In *Proceedings of the 13th Colloquium on Structural Information and Communication Complexity*, pages 245–254, 2004.
21. R. De. Prisco, A. Mayer, and M. Yung. Time-optimal message-efficientwork performance in the presence of faults. In *Proceedings of the 13th ACM Symp. Principles of Distributed Computing*, pages 161–172, 1994.

Distributed Resource Allocation in Stream Processing Systems

Cathy H. Xia[1], James A. Broberg[2], Zhen Liu[1], and Li Zhang[1]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
{cathyx, zhenl, zhangli}@us.ibm.com
[2] School of Computer Science & Information Technology, RMIT-University,
Melbourne, Australia
jbroberg@cs.rmit.edu.au

Abstract. Distributed stream processing architecture has emerged as appealing solution to coping with the analysis of large amount of data from dispersed sources. A fundamental problem in such stream processing systems is how to best utilize the available resources so that the overall system performance is optimized. We consider a distributed stream processing system that consists of a network of cooperating servers, collectively providing processing services for multiple data streams. Each stream is required to complete a series of operations on various servers. We assume all servers have finite computing resources and all communication links have finite available bandwidth. The problem is to find distributed schemes to allocate the limited *computing resources* as well as the *communication bandwidth* in the system so as to achieve a maximum concurrent throughput for all output streams. We present a generalized multicommodity flow model for the above problem. We develop a distributed resource allocation algorithm that guarantees the optimality. We also provide detailed analysis on the complexity of the algorithm and demonstrate the performance using numerical experiments.

1 Introduction

With the continued fast development and pervasive use of IT technologies, digital information collected and generated by machines put increasing stress on the processing capabilities of IT infrastructure. As a consequence, distributed stream processing architecture emerges as appealing solution to coping with the analysis of large amount of data. In such a paradigm, incoming data are processed on the fly and results are forwarded to other servers for further processing. Only a small amount of data, compared to the input rate, would be stored after these analyses.

A fundamental problem in such stream processing networks is how to best utilize available resources and coordinate the multiple servers so that the overall system performance is optimized. Since the applications are often running in a decentralized, distributed environment, at any given time, no server has the global information about all the servers in the system. One server's best

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 489–504, 2006.

decision for itself may inadvertently degrade the performance of the overall system. It is thus difficult to determine the best resource allocation policy at each server in isolation, such that the overall throughput of the system is optimized. In addition, the system must adapt to the dynamic changes in network conditions as well as the input and resource consumption fluctuations. The system needs to coordinate the processing, communication, storage/buffering, and the input/output of neighboring servers to meet these challenging requirements.

Previous work on resource management for stream processing systems has focused on either heuristics for avoiding overload or simple schemes for load-shedding (e.g. [4,9,14,15,19]). The problem of dynamic/distributed resource allocation has not yet been fully studied. This paper addresses the fundamental resource allocation questions in a distributed setting.

Specifically, we consider a distributed stream processing system that consists of a network of cooperating servers, collectively providing processing services for multiple data streams. Each stream is required to complete a series of operations on various servers. The size of the data streams may change after each operation. For example, a filtering operation may shrink the stream size, while a decryption operation may expand the stream size. This makes our corresponding flow network different from the conventional flow network since flow conservation no longer holds. We assume all servers have finite computing resources and all communication links have finite available bandwidth. The goal is to find distributed schemes to allocate the limited *computing resources* and the *communication bandwidth* in the system so as to achieve a maximum concurrent throughput for all output streams. Here the concurrent throughput means the maximum proportional throughput of the system assuming the demand proportions of the multiple streams are fixed and the system gives a fair effort for all applications.

Our problem can be formulated as a generalized multicommodity flow problem. Multicommodity flow problems have been studied extensively in the context of conventional flow networks. Readers are referred to [6,2,3] for the solution techniques and the related literature. Traditional multicommodity flow problem looks for the best way of using the link capacities to deliver the maximum throughput for the flows in the network. In our problem, in addition to the link bandwidth constraints, we also have processing power constraints for each server. Furthermore, the traditional flow conservation is generalized so as to allow flow shrinkage or expansion.

We present a generalized multicommodity flow model for the above problem and develop a distributed resource allocation algorithm. The algorithm is based on setting up buffers and using back-pressure governed by potential functions to move flows across various servers and communication links. Our algorithm can be considered as an extension of the algorithm in [2,3] that allows flow shrinkage or expansion. This also completes in both theory and experiments our previous simulation study [7].

The overall contributions of this paper are as follows:

- We consider the distributed resource allocation problem for a network of servers, constrained in both computing resources and communication band-

width. Our model also captures the shrinkage and expansion characteristics peculiar to stream processing, which generalizes the conventional flow network. We describe an extended graph representation that unifies the two types of resources seamlessly and then map the problem to a generalized multicommodity flow problem. (Section 2-3).
- We present a distributed algorithm to solve the resource allocation problem so as to achieve a maximum concurrent throughput for all output streams. We show that our algorithm guarantees the optimality and provide detailed analysis on the complexity. We also discuss how the algorithms can adapt to dynamic changes (Section 4-5).
- We present experimental studies and demonstrate the performance of the algorithm as the network size scales up (Section 6).

Related work and some concluding remarks are finally presented in Section 7 and Section 8.

2 The Stream Processing Model

2.1 Distributed Environment

We consider a distributed stream processing system that consists of a network of cooperating servers, collectively providing processing services for multiple data streams. Each server only has knowledge about its neighboring servers. Each data stream requires to be processed according to a given sequence of tasks. Different servers may be responsible for different sets of tasks. Multiple tasks on the same server can be executed in parallel and share the computing resource.

We assume the system is constrained in both computing resources and communication bandwidth. Hence, each server is faced with two decisions: first, it has to decide the allocation of its computing resource to multiple processing tasks; second, it has to decide how to share the bandwidth on each output link among the multiple flows going through. The overall system throughput depends critically on the coordination of processing, communication, buffering/storage, and input and output of various interconnected servers. In addition, such coordination/allocation schemes have to be distributed and adaptive to the dynamically changing network environment.

2.2 Graph Representation

Graph representation can help to better describe the problem. First we introduce two types of graphs: the task sequence graph and the physical server graph.

The task sequence graph describes the logical relationship among the processing tasks for each data stream. The processing for each data stream consists of a series of tasks. The various servers are assigned to process one task for each data stream. A task may be assigned to multiple servers, and tasks that belong to different streams may be assigned to the same server. The placement of the various tasks onto the physical network itself is an interesting problem. There have been studies on how to place the various tasks onto the physical network. Readers are referred to [18] for an overview. Here, we assume the task to server assignment is given.

The physical server graph describes the relationship among the servers based on the task sequence graph. Based on the task to server assignment, the tasks of each stream form a directed acyclic sub-graph in the physical server graph. Consider, for example, a system with 8 servers and 2 streams. Stream S1 requires the sequential processing of Tasks A, B, C, and D, and Stream S2 requires in sequence Tasks G, E, F, and H. Suppose the tasks are assigned such that $\mathcal{T}_1 = \{A\}$, $\mathcal{T}_2 = \{B\}$, $\mathcal{T}_3 = \{B, E\}$, $\mathcal{T}_4 = \{C\}$, $\mathcal{T}_5 = \{C, F\}$, $\mathcal{T}_6 = \{D\}$, $\mathcal{T}_7 = \{G\}$, $\mathcal{T}_8 = \{H\}$, where $\mathcal{T}_i$ denotes the set of tasks that are assigned to server i. Then the directed acyclic sub-graph of the physical network is shown in Figure 1, where the sub-graph composed of solid links corresponds to stream S1 and the sub-graph composed of dashed links corresponds to stream S2. It is easily verified that the sub-graphs corresponding to individual streams are directed acyclic graphs (DAG).

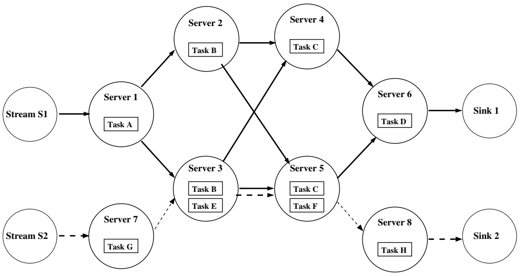


Fig. 1. Physical Server Graph

The problem can now be represented using a generic graph $G = (\mathcal{N}, \mathcal{E})$. Set $\mathcal{N}$ consists of source nodes, sink nodes and processing nodes. Directed edges in $\mathcal{E}$ represent possible information flow between various nodes. The source nodes correspond to the source of the input data streams. The sink nodes correspond to the receivers of the eventually processed information. Processing nodes stand for the various processing servers. An edge $(u, v) \in \mathcal{E}$ for server v indicates that there must be a task residing on server v that can handle data output from node u.

We refer to the different types of eventual processed information as *commodities*. We assume there are K different types of commodities, each associated with a unique source node s^k and a unique sink node t^k, $k = 1, ..., K$. Graph G is assumed to be connected. Note that G itself may not be a DAG, however, the subgraphs corresponding to individual streams are DAGs. Each processing node u has available computing resource R_u. Each pair $\{u, v\}$ of connected nodes has a finite communication bandwidth B_{uv}.

We assume it takes computing resource $r^k_{u,v}$ for node $u \in \mathcal{N}$ to process 1 unit of commodity k flow for downstream node v with $v \in \mathcal{O}(u)$. Each unit of commodity k input will result in $\beta^k_{u,v} (> 0)$ units of output after processing. This β parameter only depends on the task being executed for its corresponding stream. We shall refer to the parameter $\beta^k_{u,v}$ as *shrinkage factor*, which represents the shrinking (if < 1) or expanding (if > 1) effect in stream processing. Thus flow conservation may not hold in the processing stage.

It is possible for a stream to travel along different paths to reach the sink. The resource consumption can be different along different paths. However, the resulting outcome does not depend on the processing path. This leads to the following properties on the β parameters:

Property 1. For any two distinct paths $p = (n_0, n_1, ..., n_l)$ and $p' = (n'_0, n'_1, ..., n'_{l'})$ that have the same starting and ending points, i.e.$n_0 = n'_0$ and $n_l = n'_{l'}$, we must have $\prod_{i=0}^{l-1} \beta^k_{n_i,n_{i+1}} = \prod_{i=0}^{l'-1} \beta^k_{n'_i,n'_{i+1}}$, for any commodity k. For a given node n, denote g^k_n the product of the $\beta^k_{u,v}$'s along any path from s^k to node n.

That is, no matter which path it takes, a successful delivery of f amount of commodity k flow from source s^k to node n will result in the same amount $g^k_n f$ of output at node n. Clearly, $g^k_{s_k} = 1$. For simplicity, denote $g^k = g^k_{t_k}$. The problem is to determine the most efficient paths to process the flows so as to achieve a maximum concurrent throughput for all output streams.

3 Problem Formulation

3.1 Extended Graph Representation

We next present an extended graph representation of the original graph G that will facilitate us to address the dual resource constraints.

We introduce a bandwidth node, denoted as n_{uv}, for each pair of nodes u and v that are connected. If $(u, v) \in \mathcal{E}$, then in the new graph, there will be directed edges (u, n_{uv}) and (n_{uv}, v) (see Figure 2). The role of the bandwidth node n_{uv} is to transfer flows. We assume that each bandwidth node has resource $R_{n_{uv}} = B_{uv}$. It takes one unit of resource (bandwidth) to transfer a unit of flow, and it will become one unit of flow for the downstream node, i.e. $\beta^k_{n_{uv},v} = 1$, $r^k_{n_{uv},v} = 1$. In addition, we set $r^k_{u,n_{uv}} = r^k_{u,v}$, $\beta^k_{u,n_{uv}} = \beta^k_{u,v}$.

Denote the new graph as $G' = (\mathcal{N}', \mathcal{E}')$. In the new system, each node only has a single resource constraint. If it is a bandwidth node, then it is constrained by bandwidth; if it is a processing node, then it is constrained by the computing resource. The new system is then faced with a unified problem: finding efficient ways of shipping all K commodity flows to their respective destinations subject to the (single) resource constraint at each node.

Clearly, after the transformation, an original graph G with N nodes, M edges and K commodities would result in a new graph G' with $N + M$ nodes, $2M$ edges and K commodities. We will work on the new graph G' from here on.

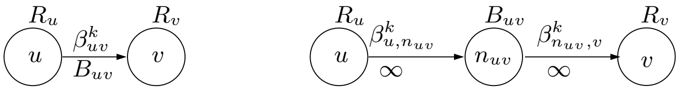


Fig. 2. An expanded graph representation of the problem

3.2 Linear Program Formulation

With the extended graph representation, the original dual-resource constrained problem is transformed into a resource allocation problem with single resource constraint on each node, which can be formulated as a linear program.

Denote $\mathcal{I}'(n)$ the set of all predecessor nodes and $\mathcal{O}'(n)$ the set of all successor nodes of node $n \in G'$. Denote $x^k_{n,n'}$ the amount of commodity k flow to be processed at node $n \in N'$ that will generate output to downstream node $n' \in \mathcal{O}'(n)$. A feasible solution has to satisfy the following conditions:

$$x^k_{n,n'} \ \geq 0, \ \ \forall k; n, n' \in \mathcal{N}' \tag{1}$$

$$\sum_k \sum_{n' \in \mathcal{O}'(n)} x^k_{n,n'} r^k_{n,n'} \leq R_n, \forall n \in \mathcal{N}' \tag{2}$$

$$\sum_{n' \in \mathcal{O}'(n)} x^k_{n,n'} - \sum_{n'' \in \mathcal{I}'(n)} x^k_{n'',n} \beta^k_{n'',n} = \begin{cases} f^k, \text{if } n = s^k \\ -g^k f^k, \text{if } n = t^k \\ 0, \text{otherwise} \end{cases} \quad \forall k; n \in \mathcal{N}'. \tag{3}$$

Condition (1) and (2) ensure respectively the non-negativity requirement on the assignment and the resource constraint on each node. Condition (3) ensures the flow balance such that incoming flows arrive at the same rate as outgoing flows being consumed at each node for each commodity. Note that such balance holds only on the nodes, not when the flow is shipped across edges due to the shrinkage and expansion effects.

The goal is to allocate the resources so as to achieve a maximum concurrent throughput for all output streams. The problem can be formulated as the following linear program:

$$\max \delta \quad \text{subject to } f^k = \delta d^k, \quad \forall k; \quad \text{and (1)-(3)}.$$

where d^k's are given. That is, the problem will find the maximum fraction δ so that the network is feasible with input rates δd^k for all commodities k. Note that the demand proportions d^k, $k = 1, \ldots, K$ are fixed and the system gives a fair effort for all applications.

The linear program represents the centralized solution, hence it does not help for our goal of obtaining distributed solutions. However, it can serve to provide the theoretical optimum so that we know the quality of our distributed solutions.

3.3 Multicommodity Flow

The resource allocation problem on the new graph G' can be interpreted as a generalized multicommodity flow problem. The variables $x^k_{n,n'}$ can be thought of as flow values of a commodity k. The demands d^k represent the rate at which flow is produced at the sources. So at each instant of time, the system is required to digest demands d^k. Different from the traditional flow problem, flow conservation no longer holds in our model when flows are shipped across edges due to the shrinkage/expansion effects. Furthermore, the capacity constraints are on the nodes instead of edges. We are interested in efficient ways of shipping the multicommodity flows to their respective destinations subject to the node resource constraints so as to achieve a maximum concurrent throughput for all output streams.

We will consider the time evolution of the network as proceeding in a sequence of synchronous rounds, each lasting for unit time. In each round, d^k units of flow are generated at each source node. The flows may be shipped between nodes, the resource of each upstream node will be consumed depending on the total amount of flow shipped, and the total amount of flow received by the downstream node will be reduced (or expanded) based on the shrinkage factor.

An algorithm for the continuous flow problem will be considered to be *stable* if the total amount of flow residing in the system at any time remains bounded. Clearly such an algorithm is able to deliver the injected flow in the long run. One can therefore construct the solution for the static problem using the average of the continuous solution. We present such a distributed algorithm for the continuous flow problem in the next section.

4 The Local Control Algorithm

We next present a local control algorithm that guides the flow transmission in between neighboring nodes. For a given set of demands d^k, the algorithm will achieve a stable solution for the continuous flow problem if the demands d^k are feasible. Our algorithm can be considered as an enhancement of the algorithm in [2,3] so that it allows general flow shrinkage or expansion.

4.1 The Algorithm

We will maintain an input and an output queue for each commodity at the tail and the head of each edge $e \in \mathcal{E}$, and denote $q^k(e^h)$ and $q^k(e^t)$ the corresponding queue heights. Let $\bar{q}^k(e^h) = q^k(e^h)/d^k$ and $\bar{q}^k(e^t) = q^k(e^t)/d^k$ be the relative heights (normalized by the demand).

We define a potential function $\Phi(\bar{q})$ associated with each queue, where the potential function is twice-differentiable and convex. We shall consider exclusively the potential function $\Phi(y) = \frac{y^2}{2}$. More efficient algorithms may be possible by choosing other potential functions such as the exponential function [3]. Define $\Phi^k(e) = \Phi(\bar{q}^k(e^t)) + \Phi(\bar{q}^k(e^h))$, for any edge e and commodity k. Define the potential of a given node $n \in \mathcal{N}'$ as $\Phi(n) = \sum_{k=1}^{K} \sum_{n' \in \mathcal{O}'(n)} \Phi^k(e_{n,n'})$, where $e_{n,n'}$ denotes edge (n, n'). The potential of the entire system, Φ, is simply $\Phi = \sum_{n \in \mathcal{N}'} \Phi(n)$. The algorithm will decide locally $x^k_{n,n'}$ the amount of commodity k flow to move across edge (n, n') with the aim to minimize the potential of the entire system.

The algorithm will proceed in a sequence of synchronous rounds. Each round the following four phases are performed.

The Local Control Algorithm (LC):

Phase 1. For each commodity k, inject d^k units of flow at its corresponding source s_k.

Phase 2. Balance all queues for commodity k at node n to equal heights.

Phase 3. For every node $n \in \mathcal{N}'$, push $x^k_{n,n'} \geq 0$ amount of commodity k flow across edge (n, n') for all $n' \in \mathcal{O}'(n)$, (let $e_{n,n'}$ denote the edge (n, n')), so that

$$\min \sum_k \sum_{n' \in \mathcal{O}'(n)} \left[\Phi(\bar{q}^k(e^t_{n,n'}) - \bar{x}^k_{n,n'}) + \Phi(\bar{q}^k(e^h_{n,n'}) + \beta^k_{n,n'} \bar{x}^k_{n,n'}) \right] \qquad (4)$$

$$\text{s.t.} \sum_k \sum_{n' \in \mathcal{O}'(n)} d^k \cdot \bar{x}^k_{n,n'} \cdot r^k_{n,n'} \leq R_n, \qquad (5)$$

where $\bar{x}^k_{n,n'} = x^k_{n,n'}/d^k$. If $n' = t_k$, we set the second term in (4) to be zero as the potential at sink node t_k is always zero.

Phase 4. Absorb flow that reached its destination.

4.2 Optimal Resource Allocation at Each Node

In Phase 3, node n's resource allocation decision is obtained by solving the local optimization problem defined by (4) and (5). In words, node n will allocate its resources so as to maximize the potential drop at node n subject to the resource constraint R_n. Using Lagrangian multipliers, the optimal solution must satisfy $\bar{x}^k_{n,n'} = \max\{\frac{\Delta^k(e_{n,n'}) - sd^k \cdot r^k_{n,n'}}{1+(\beta^k_{n,n'})^2}, 0\},$, where $\Delta^k(e_{n,n'}) = \bar{q}^k(e^t_{n,n'}) - \beta^k_{n,n'} \bar{q}^k(e^h_{n,n'})$, and $s(\geq 0)$ is the Lagrangian multiplier. The optimal value of s is the minimum $s \geq 0$ such that (5) is satisfied. That is,

$$\sum_k \sum_{n' \in \mathcal{O}'(n)} d^k r^k_{n,n'} \max\{\frac{\Delta^k(e_{n,n'}) - sd^k \cdot r^k_{n,n'}}{1 + (\beta^k_{n,n'})^2}, 0\} \leq R_n.$$

The solution can be obtained using the reverse 'water-filling' method as shown in Figure 3. Denote $h^k_{n,n'} = \frac{\Delta^k(e_{n,n'})}{d^k r^k_{n,n'}}$, and $a^k_{n,n'} = \frac{(d^k r^k_{n,n'})^2}{1+(\beta^k_{n,n'})^2}$.

For each k and $n \in \mathcal{O}'(n)$, there is a bucket of height $h^k(n, n')$ and width $a^k(n, n')$. The optimal s (see the dashed line in Figure 3) is the line above which the total area equals to available capacity R_n, which can be obtained by water filling into the bucket made by turning Figure 3 upside down. This value can also be found using a binary search.

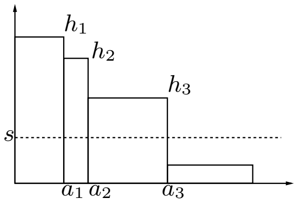


Fig. 3. Reverse water filling

4.3 Analysis

The following theorem shows that the algorithm will achieve a stable solution for demands d^k if demands $(1 + \epsilon)d^k$ are feasible. In particular, all queues in the system will stay bounded as desired. See Appendix for the detailed proof.

Theorem 1. *If for demands $(1+\epsilon)d^k$, $k=1,...,K$, there is a feasible solution that uses paths of length at most L, then for continuous flow problem with demands d^k, the local control algorithm is stable. In particular, all queues are bounded by*

$$\bar{q}_{max} := 2\sqrt{M}(\bar{\beta}^L \vee 1)\cdot C(\bar{\beta}, L)K(1+\epsilon)^2/\epsilon, \tag{6}$$

where

$$C(\bar{\beta}, L) = \frac{1+\bar{\beta}^2}{2}\cdot\frac{1-\bar{\beta}^{2L}}{1-\bar{\beta}^2}, \qquad \textit{and} \qquad \bar{\beta} = \max_{k,n,n'} \beta^k_{n,n'}. \tag{7}$$

Stop Criterion: Based on Theorem 1, since all queues are bounded, after running the LC Algorithm continuously for T rounds, when T is large enough, the flow remaining in the system will satisfy:

$$\sum_n \sum_{n'\in\mathcal{O}'(n)} \frac{[\bar{q}^k(e^t_{n,n'}) + \bar{q}^k(e^h_{n,n'})]d^k}{g^k_n} \le \delta d^k T, \tag{8}$$

for a given $\delta \in (0,1)$. Note that due to the shrinkage factor g^k_n, a unit of commodity k flow at node n corresponds to $1/g^k_n$ units of flow from source s_k being successfully shipped to node n.

After T rounds, we will have input d^kT units of commodity k into the source node s_k. From (8), at most $\delta d^k T$ units of commodity k (in the view of the source node s_k) remain in the system, thus $(1-\delta)d^kT$ units of commodity k should have successfully reached sink t_k. Hence the long run consumption rate is at least $(1-\delta)d^k$ for commodity k. (8) can be used as the stop criterion for the LC Algorithm, and the average (i.e. dividing by T) of the continuous solution gives a feasible solution to the static flow problem with demand $(1-\delta)d^k$.

The following results are then immediate.

Corollary 1. *If the multicommodity problem is feasible for demands $(1+\epsilon)d^k$, $k=1,...,K$, then there is also feasible solution (for the static problem) satisfying demands $(1-\delta)d^k$, obtained by averaging the continuous solution from the LC Algorithm.*

Theorem 2. *The LC Algorithm will satisfy demands $(1-\delta)d^k$, for any $\delta \in (0,1)$, after running for $T_{max} = R/\delta$ rounds, where $R = 8M^{3/2}(\bar{\beta}^L \vee 1)\cdot C(\bar{\beta}, L)K(1+\epsilon)^2/\epsilon$.*

Notice that when all shrinkage factors are 1's, $C(\bar{\beta}, L)$ simply becomes L and the above bound reduces to be the same as that in the original AL-algorithm [2]. For streaming applications, however, $C(\bar{\beta}, L)$ is typically much smaller because the β's are close to zero due to the high selectivity of stream applications.

5 Maximize Concurrent Throughput

We now describe how to use the LC Algorithm to find the maximum concurrent throughput. Note that for the concurrent problem, the proportion of injected flow among multiple commodities $d^1 : d^2 : \ldots : d^k$ is fixed, the problem is to find the maximum concurrent flow value δ such that input at rate $\boldsymbol{d}(\delta) := (\delta d^1, \delta d^2, \ldots, \delta d^K)$ is feasible.

We will set the input rate to be $\boldsymbol{d}(\delta)$, and vary δ till it converges to optimal δ^*. The key observation is that if δ is too large, then input rate $\boldsymbol{d}(\delta)$ is too much for the network and some queues will grow unbounded; on the other hand, if δ is smaller than δ^*, then $\boldsymbol{d}(\delta)$ should also be feasible and all queues will stay bounded. Hence, we start out δ at some value that is smaller than the true maximum concurrent flow value and increase it till we find that some queue exceeds the bound defined in Theorem 1, at which point we can be sure that the optimal δ^* is within a factor of two. We can then perform a bisection search to determine δ^* exactly.

The above procedure can be summarized as follows.

```
Search Algorithm for Max Concurrent Throughput:

Step 0. Initialize $\delta = \delta_0$, and $\eta > 0$ to be the relative error tolerance level.
Step 1. Run the LC Algorithm until $2R$ rounds pass, or one of (normalized)
    queues exceeds $\bar{q}_{max}$. If $2R$ rounds pass and no queue has exceeded $\bar{q}_{max}$,
    double $\delta$ and repeat Step 1. Otherwise, go to Step 2.
Step 2. Bisection Step.
    Step 2.0 Initialize $\delta^h = \delta$, and $\delta_l = 0$. If $\delta > \delta_0$, set $\delta_l = \delta/2$.
    Step 2.1 Set $\delta = (\delta^h + \delta_l)/2$. Run the LC Algorithm until $T_{max}$ rounds pass,
        or one of (normalized) queues exceeds $\bar{q}_{max}$. If $T_{max}$ rounds have passed,
        then set $\delta_l = \delta$ and go to Step 2.2. Otherwise, set $\delta^h = \delta$ and go to Step
        2.2.
    Step 2.2 If $(\delta^h - \delta_l)/\delta_l < \eta$, Stop. Otherwise, go back and repeat Step 2.1.
```

It can be easily shown that the total number of rounds of the above search algorithm is in the order of $T_{\max} \log \frac{(\delta^* - \delta_0)}{\delta^* \eta}$, and it can achieve a concurrent throughput of $(1 - \eta)\delta^*$.

6 Implementation and Numerical Results

In dynamic large scale environments, the centralized solutions are difficult to realize as the solutions may not be able to keep up with the changing environment. Furthermore, the centralized solutions may change by large amounts over a short time frame, which could lead to unstable system behaviors. Our algorithms presented in the previous sections have the advantages that they do not require global information, and can easily adapt to dynamic changes. The input and output queues, or the buffers in our solution make it easy to implement in real systems. They could directly correspond to the communication buffers at the servers. At each iteration, a server only needs to know the buffer levels at its neighboring nodes to determine the appropriate amount of processing and transmission. As shown in the earlier sections, such a local control mechanism will lead to global optimal solutions. Dynamic changes of the system characteristics such as changes to flow shrinkage factors or server capacities will lead to changes to the parameters in the problem formulation. Each server can keep a running estimate of these parameters. As the changes occur, each server naturally adjusts its decisions locally. Our numerical results below demonstrate that this procedure continues to provide high quality solutions.

We now present experimental results to demonstrate the performance of our distributed algorithms. We generate random graphs (based on Erdos-Renyi random graphs) of different sizes by varying the number of nodes N, the number of edges M, and the number of commodities K. The parameters are then generated from independent uniform random samples. We apply our distributed algorithm and collect the corresponding goodput (i.e. the amount of flow reached its corresponding sink successfully) for each commodity and the total run time. The corresponding optimal solution for the static linear program was obtained using LP solver.

Figure 4 shows the performance for a 3-commodity 40-node problem, where the desired goodput for the 3 commodities are respectively set at: $d^0 g^0 = 15.5$, $d^1 g^1 = 4.5$ and $d^2 g^2 = 11.1$. The left plot in Figure 4 shows the performance of the LC Algorithm, where the goodput achieved each round for each commodity is plotted as a function of the number of rounds. Observe that the goodput improves monotonically until it reaches the desired level. which shows that the LC Algorithm continuously improves its allocation thus pushing more and more flow in the right direction. We see that the LC Algorithm was able to converge to desired goodput eventually. The convergence was really quick for commodity 0 and commodity 1, and a bit slower for commodity 2. Although it takes about 6000 rounds for commodity 2 to converge to its desired throughput, in computation (CPU) time it is less than a second, which is quite impressive.

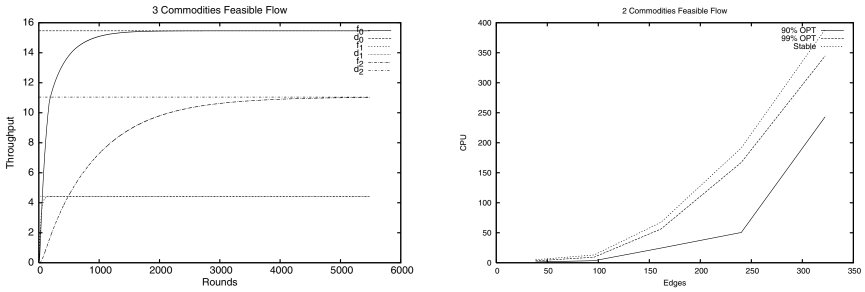


Fig. 4. For 3 commodities, 40 nodes. Left: Total flow reaching sink. Right: CPU usage as a function of network size.

The right plot in Figure 4 shows the CPU time it takes on average for the LC Algorithm to converge for networks for varying sizes. The three curves show respectively the CPU time it takes to reach 90% optimal, 99% optimal, and completely stable. We can see that it cuts at least half of the run time if a 90%-optimal solution is acceptable. Observe that the larger the network, the longer it takes to converge. The curve grows super-linearly in the total number of edges of the network, which is consistent with the complexity results derived earlier that $T_{max} \sim M^{3/2}$. In fact, since our analytical bound on the complexity is based on worst case analysis, we find that in most of our experiments, the convergence time of the distributed algorithms is typically many orders of magnitude faster than the analytically bound. In addition, the maximum queue size in the system is also typically many orders of magnitude lower than the analytical bound given by (6), which indicates that the distributed algorithm should work well in practice as the assumption on large buffer capacity is not really necessary.

The maximum concurrent flow problem is solved by using the bisection search algorithm and running the LC Algorithm multiple times. The numerical results are similar as before except that convergence speed and computation time grow a factor of $\log \frac{(\delta^* - \delta_0)}{\delta^* \eta}$ as discussed earlier. We omit these figures due to space limit.

7 Related Work

There has been a large body of work on stream processing systems. Much work has been done on data models (e.g. [1,11]), and operator scheduling (e.g. [8,5]). Recent resource management work for stream processing systems has focused on either heuristics for avoiding overload or simple schemes for load-shedding (e.g. [9,14,19]).

In this paper, we have transformed the dual-resource allocation problem into an explicit maximum concurrent flow problem. Our multicommodity flow model can be considered as a generalization of the traditional multicommodity flow model that allows flow shrinkage and expansion. Prior literature on multicommodity flow problems is also extensive. A large body of work centers on the much simpler problem of 1-commodity flow (also known as the max-flow problem). A survey of the many 1-commodity algorithms can be found in [10]. For multicommodity flows, much of the past work (e.g. [13,16,17,12]) has focused on fast algorithms to solve the centralized problem. Most of these algorithms are not well suited to distributed implementation or do not adapt well to changing network topology.

Awerbuch and Leighton [2,3] proposed a distributed multicommodity flow algorithm that finds an approximation to a feasible flow if one exists. We have extended the above algorithm to our situation. Our local control algorithm is essentially an enhancement of the above algorithm so that it allows general flow shrinkage or expansion. We theoretically analyze the algorithm and give bounds on the performance that specifically takes the shrinkage factors into account.

8 Conclusion

In summary, we have studied the problem of how to distribute the processing of multiple data streams over a network of cooperative servers. The network is resource constrained in both computing resources at each server and in bandwidth capacity over the various communication links. We present a graph representation of the problem and show how to map the original problem into an equivalent multicommodity flow problem. We have developed distributed algorithms and presented both theoretical analysis and numerical experiments. We show that the proposed distributed algorithms are able to achieve the optimal solution in the long run.

References

1. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.
2. B. Awerbuch and F. Leighton. A simple local-control approximation algorithm for multicommodity flow. *Proc. of the 34th IEEE Symp. on Foundations of Computer Science (FOCS)*, 459-468, 1993.
3. B. Awerbuch and F. Leighton. Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks. *Proc. of the 26th ACM Symp. on Theory of Computing(STOC)*, 487-496, 1994.
4. Y. Ahmad, et.al. Distributed Operation in the Borealis Stream Processing Engine, *SIGMOD'05.*
5. B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. *SIGMOD*, June 2003.
6. M. S. Bazaraa, J. J. Jarvis and H. D. Sherali. *Linear Programming and Network Flows*, John Wiley & Sons, 1977.
7. J.A. Broberg, Z. Liu, C.H. Xia and L. Zhang. A Multicommodity Flow Model for Distributed Streaming Processing, poster in *SIGMETRICS*, 2006.
8. D. Carney, U. C etintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. *29th VLDB*, Sept. 2003.
9. S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. *30th VLDB*, Sept. 2004.
10. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press: Cambridge, Mass. ; McGraw-Hill Book Company: Boston, 1990.
11. C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. *SIGMOD*, June 2003.
12. Approximating Fractional Multicommodity Flows Independent of the Number of Commodities , *SIAM J. Discrete Math.* 13 (2000), no. 4, 505-520.
13. T.C. Hu, Multi-Commodity Network Flows, *Operations Research*, 11, pg. 344-360, 1963.
14. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In CIDR, Jan. 2003.
15. P. Pietzuch, J. Shneidman, J. Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Hourglass: A Stream-Based Overlay Network for Sensor Applications. *HIP'04.*
16. Shahrokhi, F. and Matula, D.W. (1990). The maximum concurrent flow problem. *J. Assoc. Comput. Mach.* 37, 318-334.
17. C. Stein, *Approximation algorithms for multicommodity flow and shop scheduling problems*, PhD thesis, MIT, 1992.
18. U. Srivastava, K. Munagala and J. Widom, Operator placement for in-network stream query processing *Proceedings of the 24-th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 250-258, 2005.

19. N. Tatbul, U.C etintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In 29th VLDB, Sept. 2003.

Appendix

Proof. **Proof of Theorem 1.** Let f be any feasible flow for the flow problem with demands $(1+\epsilon)d^k$. We will compare the performance of the LC Algorithm against this flow. For graph G, consider any decomposition of f (the feasible solution for demand $(1+\epsilon)d^k$) into a system $\mathcal{P}$ of flow paths p of length at most L (and flow values $f(p)$ so that $\sum_{p\in\mathcal{P}} f(p) = |f|$). Denote $\mathcal{P}^k$ the set of all paths belonging to commodity k (thus start from source s_k).

Consider any node $n \in \mathcal{N}'$, the LC Algorithm selects to process flows (among all commodities and all the outgoing edges at node n) so as to minimize the resulting potential function $\Phi(n)$ (or equivalently, maximize the potential drop at node n) subject to the resource constraint R_n. Hence, as a worst case for the potential drop of the LC Algorithm, we consider the case that the LC Algorithm actually processes flow (across node n) as prescribed by $\mathcal{P}$. For any edge $e_{n,n'} = (n, n')$ with $n' \in \mathcal{O}'$, suppose that $\mathcal{P}$ sends a flow of $\psi^k_{n,n'}$ for commodity k across $e_{n,n'}$, the potential drop caused by commodity k at $e_{n,n'}$ is equal to

$$\frac{1}{2}[(\bar{q}^k(e^t_{n,n'}))^2 + (\bar{q}^k(e^h_{n,n'}))^2] - \frac{1}{2}[(\bar{q}^k(e^t_{n,n'}) - \bar{\psi}^k_{n,n'})^2 + (\bar{q}^k(e^h_{n,n'}) + \beta^k_{n,n'}\bar{\psi}^k_{n,n'})^2]$$
$$= \bar{\psi}^k_{n,n'}[\bar{q}^k(e^t_{n,n'}) - \beta^k_{n,n'}\bar{q}^k(e^h_{n,n'}) - \frac{1+(\beta^k_{n,n'})^2}{2}\bar{\psi}^k_{n,n'}]$$
$$= \bar{\psi}^k_{n,n'}[\bar{q}^k_n - \beta^k_{n,n'}\bar{q}^k_{n'} - \frac{1+(\beta^k_{n,n'})^2}{2}\bar{\psi}^k_{n,n'}] \tag{9}$$

where $\bar{\psi}^k_{n,n'} = \psi^k_{n,n'}/d^k$, and $\bar{q}^k_n$ denotes the normalized amount of commodity k in any queue at node n at the beginning of Phase 3 (since all queues at a given node are balanced and have equal heights).

Hence, the potential drop at node n under the LC Algorithm is at least:

$$\sum_{k=1}^{K} \sum_{n'\in\mathcal{O}'(n)} \bar{\psi}^k_{n,n'}[\bar{q}^k_n - \beta^k_{n,n'}\bar{q}^k_{n'} - \frac{1+(\beta^k_{n,n'})^2}{2}\bar{\psi}^k_{n,n'}]. \tag{10}$$

Note that $\bar{\psi}^k_{n,n'}$ can be split among the flow paths p across $e_{n,n'}$ belonging to commodity k. If path $p \in \mathcal{P}^k$ contains edge $e_{n,n'}$, then the $f(p)$ amount of flow at source s^k will become $g^k_n f(p)$ amount of flow at node n, where g^k_n is the product of the $\beta^k_{n,n'}$'s along any path from s^k to n, and is invariant for all paths from source s^k to node n due to Property 1. Let $\bar{f}(p) = f(p)/d^k$. Then,

$$\bar{\psi}^k_{n,n'} = \sum_{p\in\mathcal{P}^k : e_{n,n'}\in p} g^k_n \bar{f}(p) \le g^k_n \sum_{p\in\mathcal{P}^k} \bar{f}(p) = g^k_n(1+\epsilon), \tag{11}$$

since $\sum_{p\in\mathcal{P}^k} \bar{f}(p) = \sum_{p\in\mathcal{P}^k} f(p)/d^k = (1+\epsilon)d^k/d^k = (1+\epsilon)$.

Based on (11), each entry in (10) is bounded below by

$$\sum_{p\in\mathcal{P}^k:e_{n,n'}\in p} g_n^k \bar{f}(p)[\bar{q}_n^k - \beta_{n,n'}^k \bar{q}_{n'}^k - \frac{1+\bar{\beta}^2}{2} g_n^k(1+\epsilon)]$$
$$= \sum_{p\in\mathcal{P}^k:e_{n,n'}\in p} \bar{f}(p)[g_n^k\bar{q}_n^k - g_{n'}^k\bar{q}_{n'}^k - \frac{1+\bar{\beta}^2}{2}(1+\epsilon)(g_n^k)^2].$$

where $\bar{\beta}$ is defined by (7), and the last equality is because $g_{n'}^k = g_n^k\beta_{n,n'}^k$ due to Property 1.

Summing up over all nodes in $\mathcal{N}'$, we then have

$$\sum_{n\in\mathcal{N}'}\sum_{k=1}^{K}\sum_{n'\in\mathcal{O}'(n)}\sum_{p\in\mathcal{P}^k:e_{n,n'}\in p} \bar{f}(p)[g_n^k\bar{q}_n^k - g_{n'}^k\bar{q}_{n'}^k - \frac{1+\bar{\beta}^2}{2}(1+\epsilon)(g_n^k)^2]$$
$$=\sum_{k=1}^{K}\sum_{p\in\mathcal{P}^k}\sum_{e_{n,n'}\in p} \bar{f}(p)[g_n^k\bar{q}_n^k - g_{n'}^k\bar{q}_{n'}^k - \frac{1+\bar{\beta}^2}{2}(1+\epsilon)(g_n^k)^2]$$
$$\geq\sum_{k=1}^{K}\sum_{p\in\mathcal{P}^k} \bar{f}(p)[\bar{q}^k - \frac{1+\bar{\beta}^2}{2}\cdot\frac{1-\bar{\beta}^{2L}}{1-\bar{\beta}^2}\cdot(1+\epsilon)]$$
$$=\sum_{k=1}^{K}(1+\epsilon)[\bar{q}^k - C(\bar{\beta},L)(1+\epsilon)]$$
$$=(1+\epsilon)\sum_{k}\bar{q}^k - C(\bar{\beta},L)K(1+\epsilon)^2,$$

where $\bar{q}^k$ $(=\bar{q}_{s_k}^k)$ is the normalized amount of commodity k in the queues of source node s_k, and $C(\bar{\beta},L)$ is given by (7). For the last inequality, we used the fact that $g_n^k \leq \bar{\beta}^j$ if n is the j-th hop along path p.

Hence, the potential drop due to the movement of flow in Phases 3 and 4 of the LC Algorithm is at least:

$$(1+\epsilon)\sum_{k}\bar{q}^k - C(\bar{\beta},L)K(1+\epsilon)^2.$$

We next look at the potential change caused by injecting new flow at Phase 1 of the LC Algorithm. Recall that $\bar{q}^k$ is the normalized amount of commodity k in the queues of the source s^k of commodity k after injecting new flow at Phase 1. Suppose there are J outgoing edges from the source s^k, and a fraction $\bar{x}_j$ of the newly injected d^k amount of commodity k flow was assigned to edge i, $j=1,\ldots,J$, and $\sum_{j=1}^{J}\bar{x}_j = 1$. Then the total potential increase is:

$$\sum_{j=1}^{J}\frac{1}{2}(\bar{q}^k)^2 - \frac{1}{2}(\bar{q}^k-\bar{x}_j)^2 = \sum_{j=1}^{J}\frac{1}{2}\bar{x}_j(2\bar{q}^k-\bar{x}_j) \leq \sum_{j=1}^{J}\bar{x}_j\bar{q}^k = \bar{q}^k.$$

Thus the potential increase caused by phase 1 of the LC Algorithm is at most $\sum_k \bar{q}^k$.

Phase 2 of the LC Algorithm can only decrease the potential. This is because the total potential function associated with all the queues at a given node is Schur convex, as discussed earlier. Thus balancing all the queues at a given node can only decrease the potential at that node.

Hence, the overall potential increase in one round of the LC Algorithm is at most

$$-\epsilon \sum_k \bar{q}^k + C(\bar{\beta}, L)K(1+\epsilon)^2. \tag{12}$$

This value is guaranteed to be negative (i.e. the potential decreases) if

$$\sum_k \bar{q}^k > C(\bar{\beta}, L)K(1+\epsilon)^2/\epsilon. \tag{13}$$

Note that flow is only pushed through edge (n, n') when $\bar{q}_n^k > \beta_{n,n'}^k \bar{q}_{n'}^k$. In other words, if we normalize queue $\bar{q}_n^k$ by g_n^k for all k and n (so that all queue heights are relative to the source queue), then the LC Algorithm is only pushing flow from high to low and the source queue $\bar{q}_k$ should always be the highest. Hence $\bar{q}^k \geq \bar{q}_n^k / g_n^k$.

At each round, either condition (13) does not hold, then $\bar{q}_n^k \leq g_n^k \bar{q}^k \leq (\bar{\beta}^L \vee 1) \cdot C(\bar{\beta}, L)K(1+\epsilon)^2/\epsilon$ and there are at most $2M$ queues of each commodity in the system, where $2M$ is the number of edges in the graph G'. If condition (13) does hold, then according to (12), the overall potential Φ will decrease in that round. Hence by induction, the potential at any round must be limited to

$$\Phi \leq 2M \cdot \frac{1}{2}[(\bar{\beta}^L \vee 1) \cdot C(\bar{\beta}, L)K(1+\epsilon)^2/\epsilon]^2.$$

In the worst case, all of this potential may be concentrated in a single queue. Hence, the maximum value a $\bar{q}^k(e)$ can attain is bounded by $\bar{q}_{max}$ given by (6).

Proof. **Proof of Theorem 2.** Note that the left hand side of (8) is bounded by $2M\bar{q}_k d^k$, by setting (8) to equality, we can then obtain an upper bound for the stopping time $T = R/\delta$, where $R = 2M\bar{q}_{max}$. The claimed result then follows immediately based on (6).

Low-Latency Atomic Broadcast in the Presence of Contention

Piotr Zieliński

University of Cambridge
Cavendish Laboratory
piotr.zielinski@cl.cam.ac.uk

Abstract. The Atomic Broadcast algorithm described in this paper can deliver messages in two communication steps, even if multiple processes broadcast at the same time. It tags all broadcast messages with the local real time, and delivers all messages in order of these timestamps. The Ω-elected leader simulates processes it suspects to have crashed ($\Diamond S$). For fault-tolerance, it uses a new cheap Generic Broadcast algorithm that requires only a majority of correct processes ($n > 2f$) and, in failure-free runs, delivers all non-conflicting messages in two steps. The main algorithm satisfies several new lower bounds.

1 Introduction

In a distributed system, messages broadcast by different processes at approximately the same time might be *received* by other processes in different orders. Atomic Broadcast is a fault-tolerant primitive, usually implemented on top of ordinary broadcast, which ensures that all processes *deliver* messages to the user in the same order. Applications of Atomic Broadcast include state machine replication, distributed databases, distributed shared memory, and others [4].

As opposed to ordinary broadcast, Atomic Broadcast requires multiple communication steps, even in runs without failures. One of the goals in broadcast protocol design is minimizing the latency in common, failure-free runs, while possibly allowing worse performance in runs with failures. This paper presents an algorithm that is faster, in this respect, than any previously proposed one, and requires only two communication steps, even if multiple processes broadcast at the same time.

The definition of latency in this context can be a source of confusion. In this paper, latency is the time between the atomic broadcast (*abcast*) of a message and its atomic delivery. Note that some papers [5, 8] ignore the step in which the sender physically broadcasts the message to other processes; in that case one step must be added to the reported latency figure.

The algorithm presented here assumes an asynchronous system with a majority of correct processes and the $\Diamond S$ failure detector [2]. Motivated by the increasing availability of services such as GPS or Network Time Protocol [9], I additionally assume that each process is equipped with a (possibly inaccurate) real-time clock. The optimum latency of two steps is achieved if the clocks

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 505–519, 2006.

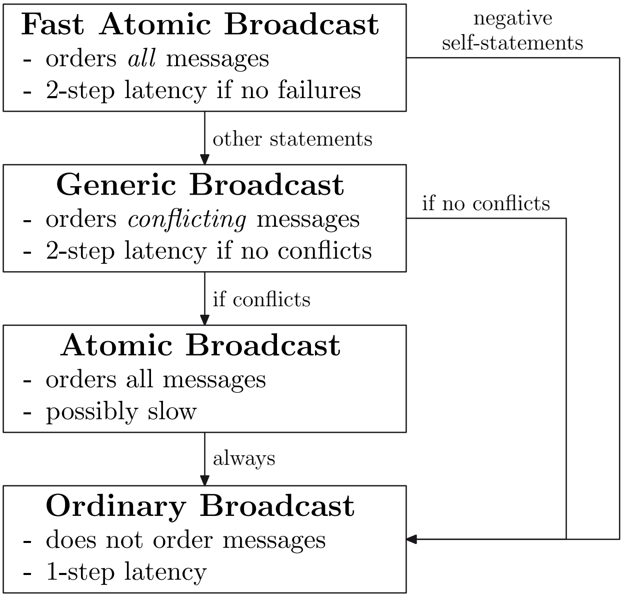


Fig. 1. Layered structure of the Atomic Broadcast algorithm presented in this paper

are synchronized, and degrades gracefully otherwise. In particular, no safety or liveness properties depend on the clock accuracy.

The algorithm employs a well-known method first proposed by Lamport [7]: senders independently timestamp their messages, which are then delivered in the order of these timestamps. The novelty of my approach consists of using real-time clocks in conjunction with unreliable failure detectors [2] and Generic Broadcast [1, 12] to ensure low latency and fault-tolerance at the same time.

The layered structure of the algorithm is shown in Fig. 1. Each process tags all its abcast messages with the local real time, and disseminates this information using Generic Broadcast. Both positive and negative *statements* are used ("m abcast at time 51" vs. "no messages abcast between times 30 and 50"). To achieve fault-tolerance, the Ω-elected leader occasionally broadcasts negative statements on behalf of processes it suspects to have crashed ($\Diamond S$). Any Generic Broadcast algorithm can be used here, including the new algorithm presented in Sect. 5, which is specifically tailored for this situation. It requires only a majority of correct processes ($n > 2f$) and, in failure-free runs, delivers all non-conflicting messages in two communication steps.

This paper is structured as follows. Section 2 formalizes the system model and gives a precise definition of Atomic Broadcast. Section 3 presents the main ideas, which are then transformed into an algorithm in Sect. 4. Section 5 describes a new Generic Broadcast algorithm, that is optimized for the use by the main algorithm. Section 6 shortly discusses some secondary properties of the algorithm. Section 7 presents three new lower bounds that prove that two-step delivery cannot be maintained if the system assumptions are relaxed (for example, by eliminating real-time clocks). Section 8 concludes the paper. Detailed proofs can be found in the extended version of this paper [15].

1.1 Related Work

A run of an algorithm is *good* if the real-time clocks are accurate and there are no failures or suspicions. In such runs, the algorithm presented here delivers all messages in two steps. No such algorithm has been proposed before; out of over fifty Atomic Broadcast protocols surveyed by Défago et al. [4], the only *indulgent* algorithm capable of delivering all messages faster than in three steps was proposed by Vicente and Rodrigues [13]. It achieves a latency of $2d + \delta$, where d is the single message delay and $\delta > 0$ is an arbitrarily small constant. The price for having a very small δ is high network traffic; the number of messages is proportional to $1/\delta$. In comparison, my algorithm achieves the latency of $2d$ with the number of messages dependent only on the number of processes.

Several broadcast protocols achieve a two-step latency in some good runs, such as those with all messages spontaneously received in order (Optimistic Atomic Broadcast [10, 14]) or those without conflicting messages (Generic Broadcast [1, 10, 12, 14]). In comparison, the algorithm presented in this paper delivers messages in two steps in all good runs.

Défago et al. [4] proposed a classification scheme for broadcast algorithms. According to that scheme, the algorithm described in this paper is a time-based communication-history algorithm for closed groups, similarly to the original Lamport's algorithm [7], which motivated it.

2 System Model and Definitions

The system model consists of n processes $p_1, \ldots, p_n$, out of which at most f can fail by crashing. Less than a half of all the processes are faulty ($n > 2f$). Processes communicate through asynchronous reliable channels, that is, there is no time limit on message transmission time, and messages between correct processes never get lost.

Each process is equipped with: (i) a possibly inaccurate, non-decreasing real-time clock, (ii) an unreliable leader oracle Ω, which eventually outputs the same correct leader at all correct processes, and (iii) a failure detector $\Diamond S$. Failure detector $\Diamond S$ outputs a list of processes that it suspect to have crashed. It ensures that (i) all crashed processes will eventually be suspected by all correct processes, and (ii) at least one correct process will eventually never be suspected by any correct process [3]. Detector $\Diamond S$ is the weakest suspicion-list-like failure detector that makes Atomic Broadcast solvable in asynchronous settings. It can implement Ω [3], so the Ω assumption can, technically, be dropped.

In Atomic Broadcast, processes abcast messages, which are then delivered by all processes in the same order. Formally [6],

Validity. If a correct process abcasts a message m, then all correct processes will eventually deliver m.

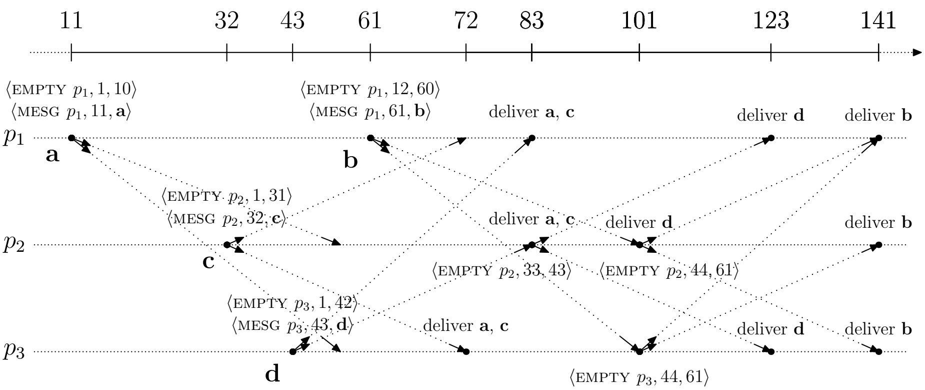


Fig. 2. A run that uses ordinary broadcast for all statements (not fault-tolerant). One step is 40 units of time. The fault-tolerant version is shown in Fig. 4.

Uniform Agreement. If a process delivers a message m, then all correct processes eventually deliver m.

Uniform Integrity. For any message m, every process delivers m at most once, and only if m was previously abcast.

Uniform Total Order. If some process delivers message m' after message m, then every process delivers m' only after it has delivered m.

I measure latency in *communication steps*, where one communication step is the maximum message delay d between *correct* processes (possibly ∞). Processes do not know d. In good runs, the algorithm described in this paper delivers all messages in two communication steps ($2d$), regardless of the number of processes abcasting simultaneously. In other runs, the performance can be worse, however, the above four properties of Atomic Broadcast always hold (Sect. 6).

3 Atomic Broadcast Algorithm

The algorithm employs a well-known method proposed by Lamport [7]: senders independently timestamp their messages, which are then delivered in the order of these timestamps. In the scenario in Fig. 2, messages **a**, **b**, **c**, **d** are tagged by their senders with timestamps 11, 61, 32, and 43. As a result, they should be delivered in the order: **a**, **c**, **d**, and **b**.[1]

3.1 Runs Without Failures

How can we implement this idea is a message-passing environment? To deliver messages in the right order, a process, say p_3, must know the timestamps of

[1] For simplicity, I assume that timestamps are (possibly very large) integers whose last digit is the process number, so that no two messages can carry the same timestamp.

messages **a**, **b**, **c**, **d**, and that no other messages were abcast. Let the *abcast state* of p_i at time t be the message (if any) abcast by p_i at time t. For example, the abcast state of p_2 at time 32 is **c**, and empty at time 34.

Processes share information by broadcasting their abcast states. When a process p_i abcasts a message m at time t, it broadcasts two statements (as separate messages):

1. A *positive* statement $\langle \textsc{mesg}\ p_i, t, m\rangle$ saying that p_i abcast message m at time t.
2. A *negative* statement $\langle \textsc{empty}\ p_i, t'+1, t-1\rangle$ saying that p_i abcast no messages since abcasting its previous message at time t' ($t' = 0$ if m is the first message abcast by p_i).

In the example in Fig. 2, the following statements are broadcast:

$$\langle \textsc{empty}\ p_1, 1, 10\rangle,\quad \langle \textsc{mesg}\ p_1, 11, \mathbf{a}\rangle,\quad \langle \textsc{empty}\ p_1, 12, 60\rangle,\quad \langle \textsc{mesg}\ p_1, 61, \mathbf{b}\rangle,$$
$$\langle \textsc{empty}\ p_2, 1, 31\rangle,\quad \langle \textsc{mesg}\ p_2, 32, \mathbf{c}\rangle,\quad \langle \textsc{empty}\ p_3, 1, 42\rangle,\quad \langle \textsc{mesg}\ p_3, 43, \mathbf{d}\rangle.$$

After receiving these statements, we have complete information about all processes abcast states up to time 32. We can deliver messages **a** and **c**, in this order, because we know that no other messages were abcast with a timestamp ≤ 32. On the other hand, **d** cannot be delivered yet, because we do not have any information about the abcast state of process p_2 after time 32. If we delivered **d**, and later found out that p_2 abcast **e** at time 42, we would violate the rule that messages are delivered in order of their timestamps ($43 \not< 42$).

To deliver **d**, we need p_2's help. When p_2 learns that p_3 abcast a message at time 43, it announces its abcast states by broadcasting $\langle \textsc{empty}\ p_2, 33, 43\rangle$ (Fig. 2). Similarly, when processes p_2 and p_3 learn about **b**, they broadcast $\langle \textsc{empty}\ p_2, 44, 61\rangle$ and $\langle \textsc{empty}\ p_3, 44, 61\rangle$ respectively. Note that p_1 does not need to broadcast anything, because by the time it learnt about **c** and **d**, it had already broadcast the necessary information while abcasting **b**. After receiving all these statements, we now have complete information about all processes abcast states up to time 61. In addition to previously delivered **a** and **c**, we can now deliver **d** and **b** as well. Note that, in the failure-free case, the order in which processes receive statements is irrelevant.

3.2 Dealing with Failures

What would happen if p_2 crashed immediately after sending **c**? Process p_2 would never broadcast the negative statement $\langle \textsc{empty}\ p_2, 33, 43\rangle$, so the algorithm would never deliver **d** and **b**.

To cope with this problem, the current leader (Ω) broadcasts the required negative statements on behalf of all processes it suspects ($\Diamond S$) to have failed. For example, if p_1 is the leader, suspects p_2, and learns that p_3 abcast **d** at time 43, then p_1 broadcasts $\langle \textsc{empty}\ p_2, 1, 43\rangle$. This allows message **d** to be delivered.

Allowing p_1 to make negative statements on behalf of p_2 opens a whole can of worms. To start with, $\langle \textsc{empty}\ p_2, 1, 43\rangle$ blatantly contradicts $\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle$

broadcast earlier by p_2. A similar conflict occurs if p_1 is wrong in suspecting p_2, and p_2 decides to abcast another message **e**, say at time 42. In general, two statements *conflict* if they carry different information about the abcast state of the same process at the same time.

The problem of conflicting statements can be solved by assuming that, if a process receives two conflicting statements, then the first one wins. For example, receiving

$$\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle, \langle \textsc{empty}\ p_2, 1, 43\rangle, \langle \textsc{mesg}\ p_2, 42, \mathbf{e}\rangle$$

is equivalent to

$$\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle, \langle \textsc{empty}\ p_2, 33, 43\rangle.$$

We can ensure that all processes receive all conflicting statements in the same order by using Generic Broadcast [1, 12] to broadcast them. Unlike Atomic Broadcast, Generic Broadcast imposes order only on conflicting messages, which leads to good performance in runs without conflicts.

3.3 Latency Considerations

In order to achieve a two-step latency in good runs, all positive statements must be delivered in two communication steps. Negative statements are even more problematic, because they may be issued one step after the abcast event that triggered them (e.g., $\langle \textsc{empty}\ p_2, 33, 43\rangle$ triggered by p_3 abcasting **d** at time 43 in Fig. 2). Therefore, negative statement must be delivered in at most one step. The following observations show how to satisfy these requirements.

Observation 1: No conflicts in good runs. Good runs have no suspicions, so processes issue statements only about themselves. These *self-statements* never conflict. Since no conflicting statements are issued, Generic Broadcast will deliver all statements in two communication steps [1, 12].

Observation 2: No conflicts involving negative self-statements. Statements made by processes can be divided into three disjoint groups: positive self-statements, negative self-statements, and negative statements made by the leader. Negative self-statements do not conflict with any of these because (i) self-statements do not conflict because they talk about different processes or times, and (ii) negative statements do not conflict because they carry the same information "no messages". Therefore, negative self-statements do not require Generic Broadcast; ordinary broadcast, which takes only one communication step, is sufficient.

4 Implementation

Figure 3 presents the details of the algorithm sketched in Sect. 3. It can be conceptually divided into two parts: broadcasting (lines 1–13) and delivery (lines 14–28).

```
1  $t_i^{\max} \leftarrow 0$                              { the highest timestamp used so far }
2  when $p_i$ executes $abcast(m)$ do
3    broadcast $\langle$ACTIVE $time_i\rangle$ using ordinary broadcast
4    broadcast $\langle$EMPTY $p_i, t_{\max}+1, time_i - 1\rangle$ using ordinary broadcast
5    broadcast $\langle$MESG $p_i, time_i, m\rangle$ using Generic Broadcast
6    $t_i^{\max} \leftarrow time_i$                     { $time_i$ increases after this line }
7  when $p_i$ received $\langle$ACTIVE $t\rangle$ in the past and $t_i^{\max} < t \leq time_i$ do
8    broadcast $\langle$EMPTY $p_i, t_i^{\max}+1, t\rangle$ using ordinary broadcast
9    $t_i^{\max} \leftarrow t$                          { $time_i$ increases after this line }
10 when change in $t_i^{\max}$ or the output of the failure detector or leader oracle do
11   if $p_i$ considers itself a leader
12     for all suspected processes $p_j \neq p_i$ do
13       broadcast $\langle$EMPTY $p_j, 1, time_i\rangle$ using Generic Broadcast
14 task delivery at process $p_i$ is
15   $todeliver_i \leftarrow \emptyset$; $known_i[j] \leftarrow \emptyset$ for all $j = 1, \ldots, n$
16   repeat forever
17     wait for a statement delivered by ordinary or Generic Broadcast
18       if negative statement $\langle$EMPTY $p_j, t_1, t_2\rangle$ delivered then
19         $known_i[j] \leftarrow known_i[j] \cup [t_1, t_2]$
20       if positive statement $\langle$MESG $p_j, t, m\rangle$ delivered then
21         if $t \notin known_i[j]$ then
22           send $\langle$ACTIVE $t\rangle$ to itself
23           add $(m, t)$ to $todeliver_i$
24           $known_i[j] \leftarrow known_i[j] \cup \{t\}$
25         else if $p_i = p_j$ then $abcast(m)$          { the sender tries again }
26     let $t_i^{\mathrm{known}} = \max\{\, t \mid [1, t] \subseteq known_i[j]$ for all $j\,\}$
27     for all $(m, t) \in todeliver_i$ with $t \leq t_i^{\mathrm{known}}$, in the order of increasing $t$ do
28       atomically deliver $m$; remove $(m, t)$ from $todeliver_i$
```

Fig. 3. Atomic Broadcast algorithm with a two-step latency in good runs. It requires the $\Diamond P$ failure detector; see Sect. 4.3 for the modifications required for $\Diamond S$.

4.1 Broadcasting Part (Lines 1–13)

Each process p_i maintains two variables: $time_i$ and $t_i^{\max}$. The read-only variable $time_i$ is an integer representing the current reading of p_i's clock. Its value increases at the end of every "block" (e.g. lines 2–6), but remains constant within it. Variable $t_i^{\max}$ represents the highest time for which p_i broadcast a statement about its abcast state. For example $t_1^{\max} = 61$ after p_1 abcast **a** and **b** (Fig. 4). Initially $t_i^{\max} = 0$.

To abcast a message m, a process p_i first broadcasts $\langle$ACTIVE $time_i\rangle$, which informs other processes that some message was abcast at time $time_i$. Then, p_i broadcasts one negative and one positive statement using ordinary and Generic Broadcast, respectively. They inform other processes that p_i abcast m at time $time_i$, and nothing between $t_i^{\max} + 1$ and $time_i - 1$. Finally, p_i updates $t_i^{\max}$.

When process p_i receives $\langle$ACTIVE $t\rangle$ with $t_i^{\max} < t \leq time_i$, it broadcasts $\langle$EMPTY $p_i, t_i^{\max} + 1, t\rangle$ and updates $t_i^{\max}$. This informs other processes that p_i

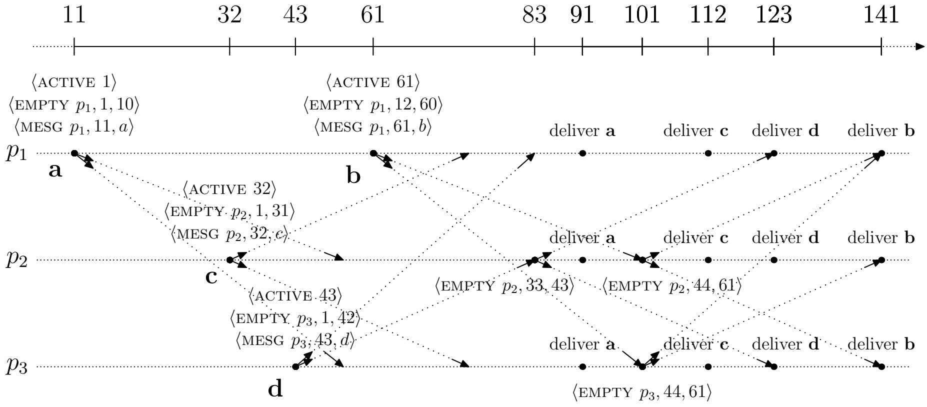


Fig. 4. A example run of the fault-tolerant algorithm from Fig. 3. Generic Broadcast of statements ⟨MESG⟩ takes two steps; the related messages are not shown.

abcast no messages between $t^{\max} + 1$ and t. If $t_i^{\max} \geq t$, then p_i has already reported its abcast states for time t and before, so no new statement is needed.

The condition $t \leq time_i$ makes sure that $t_i^{\max} < time_i$ at all times, so that the abcast function does not a issue a conflicting self-statement in line 5. This condition always holds in good runs: $t > time_i$ would mean that the message ⟨ACTIVE t⟩ arrived from a process whose clock was at least one communication step ahead of p_i's. In this case, p_i simply waits until $t \leq time_i$ holds, and then executes lines 7–9.

Lines 10–13 are executed when a leader process p_i experiences a change in $t_i^{\max}$ or in the output of its failure detector $\Diamond S$ or the leader oracle Ω. In these cases, p_i issues the appropriate negative statements on behalf of all processes it suspects to have crashed.

4.2 Delivery Part (Lines 14–28)

The delivery tasks delivers messages in the order of their timestamps. Each process p_i maintains two variables: $todeliver_i$ and $known_i$. Variable $todeliver_i$ contains all timestamped messages (m, t) that have been received but not atomically delivered yet.

Variable $known_i$ is an array of sets. Each $known_i[j]$ is the set of all times for which the abcast state of p_j is known, initially $\emptyset$. For example, at time 80, $known_2[1] = [1, 10]$, where $[t_1, t_2] = \{\, t \mid t_1 \leq t \leq t_2 \,\}$. Time $11 \notin known_2$ because ⟨MESG $p_1, 11, \mathbf{a}$⟩, sent using Generic Broadcast, will arrive at p_2 two communication steps after being sent, that is, at time 91. Each set $known_i[j]$ can be compactly represented as union of intervals $[t_1, t_2]$; most of the time $known_i[j] = [1, t]$ for some t.

In each iteration of the infinite loop (lines 16–28), the delivery task waits for a statement delivered by ordinary or Generic Broadcast. After receiving

$\langle \text{EMPTY } p_j, t_1, t_2 \rangle$, process p_i updates its knowledge about p_j by adding the interval $[t_1, t_2]$ to $known_i[j]$. For $\langle \text{MESG } p_j, t, m \rangle$, p_i first checks whether it has received any information about the abcast state of p_j at time t before. If not, p_i schedules message m for delivery by adding the pair (m, t) to $todeliver_i$. Process p_i also adds $\{t\}$ to $known_i[j]$ to reflect its knowledge of the abcast state of p_j at time t. Sending $\langle \text{ACTIVE } t \rangle$ in line 22 is necessary to ensure Uniform Agreement on messages abcast by faulty processes. Messages $\langle \text{ACTIVE } t \rangle$ broadcast by such processes in line 3 can get lost, so line 22 serves as a backup.

If $t \in known_i[j]$, then the leader suspected p_j to have crashed, and broadcast a conflicting negative statement on p_j's behalf, which was delivered before $\langle \text{MESG } p_j, t, m \rangle$. Since message m cannot be delivered with timestamp t, its sender tries to abcast it again, with a new timestamp. In the future, if m cannot be delivered with the new timestamp, its sender will re-abcast it yet again, and so on. If the failure detector is $\Diamond P$, all correct processes will eventually be permanently not suspected, so some re-abcast of m will eventually result in delivering m. For dealing with $\Diamond S$, see Sect. 4.3.

After processing the statement received in line 17, p_i attempts to deliver abcast messages. It first computes the largest time t_i^{known} for which it knows the abcast states of all processes up to time t_i^{known}. This ensures that $todeliver_i$ contains all non-yet-delivered messages abcast by time t_i^{known}. Process p_i delivers all these messages in the order of increasing timestamps and removes them from $todeliver_i$.

4.3 Dealing with $\Diamond S$ by Leader-Controlled Retransmission

With the $\Diamond S$ failure detector, the algorithm from Fig. 3 might fail to deliver messages abcast by senders that are correct but permanently suspected by the leader (this cannot happen with $\Diamond P$). This problem can be solved by letting the leader re-abcast all messages, instead of each process doing so itself.

In this scheme, each process p_i maintains a set $\mathcal{B}_i$ of messages it abcast but not delivered yet. Periodically, it sends $\mathcal{B}_i$ to the current leader, who re-abcasts all $m \in \mathcal{B}_i$ on p_i's behalf. Since Ω guarantees an eventual stable leader, each message abcast by a correct p_i will eventually be delivered (Validity). Some messages might be delivered twice, so an explicit duplicate elimination must be employed.

5 Cheap Generic Broadcast

The Atomic Broadcast algorithm from Sect. 4 assumes that, in failure-free runs, the underlying Generic Broadcast delivers all non-conflicting messages in two communication steps. Achieving this with existing Generic Broadcast algorithms requires $n > 3f$ [1, 12, 14].

Figure 5 presents a Generic Broadcast algorithm, which is similar to [1] but requires only $n > 2f$ (cheapness). As opposed to other Generic Broadcast algorithms, it achieves a two-step latency only in failure-free runs (otherwise $n > 3f$

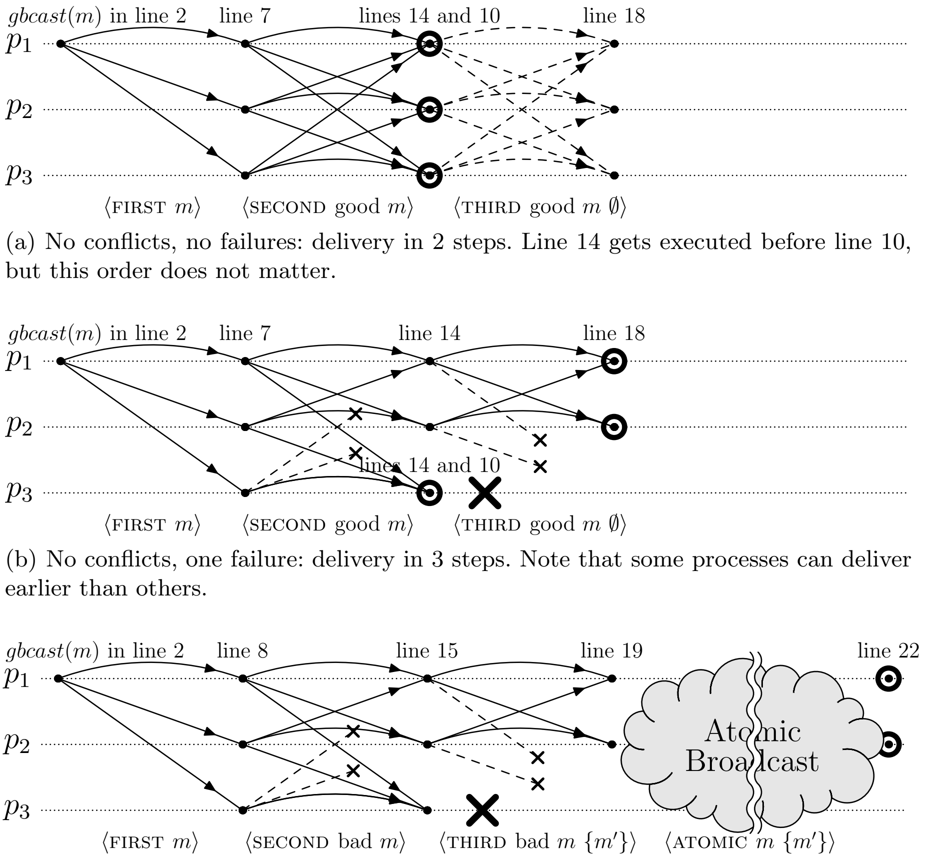
```
$seen_i^1 \leftarrow \emptyset$; $seen_i^2 \leftarrow \emptyset$; $quick_i \leftarrow \emptyset$
when $p_i$ executes $gbcast(m)$ do                { broadcast $m$ using Generic Broadcast }
  broadcast ⟨FIRST $m$⟩ using (non-uniform) Reliable Broadcast
when $p_i$ receives ⟨FIRST $m$⟩ do
  add $m$ to $seen_i^1$
  if $seen_i^1 \cap C(m) = \emptyset$              { $C(m)$ is the set of messages conflicting with $m$ }
    then FIFO broadcast ⟨SECOND good $m$⟩
    else FIFO broadcast ⟨SECOND bad $m$⟩
when $p_i$ receives ⟨SECOND good $m$⟩ from all processes do
  deliver $m$ if not delivered already                        { two-step delivery }
when $p_i$ receives ⟨SECOND * $m$⟩ from $n - f$ processes do
  add $m$ to $seen_i^2$
  if all "*" are "good" and $seen_i^2 \cap C(m) = \emptyset$
    then add $m$ to $quick_i$; broadcast ⟨THIRD good $m$ $\emptyset$⟩
    else broadcast ⟨THIRD bad $m$ $conflicts_i$⟩ where $conflicts_i = quick_i \cap C(m)$
when $p_i$ receives ⟨THIRD * $m$ $conflicts_j$⟩ from $n - f$ processes $p_j$ do
  if all "*" are "good"
    then deliver $m$ if not delivered already                 { three-step delivery }
    else atomically bcast ⟨ATOMIC $m$ $conflicts$⟩ with $conflicts = \bigcup_j conflicts_j$
when a process atomically delivers ⟨ATOMIC $m$ $conflicts$⟩ do
  deliver all $m' \in conflicts$ if not delivered already
  deliver $m$ if not delivered already
```

Fig. 5. Generic Broadcast algorithm that achieves a two-step latency in good runs and requires only $n > 2f$ (cheapness)

would be a lower bound [11]). As a bonus, all non-conflicting messages are delivered in three steps even in runs with failures, so the algorithm in Fig. 5 could be seen as a generalization of three-step Generic Broadcast protocols that require $n > 2f$ [1, 12]. To deal with conflicting messages, the algorithm employs an auxiliary Atomic Broadcast protocol, such as [2]. Real-time clocks are not used.

To execute *gbcast*(m), the sender sends ⟨FIRST m⟩ using Reliable Broadcast [6]. When a process p_i receives ⟨FIRST m⟩, it first checks whether any messages conflicting with m have reached this stage before. ($C(m)$ is the set of messages conflicting with m.) Process p_i then broadcasts ⟨SECOND good m⟩ or ⟨SECOND bad m⟩ accordingly. In failure-free runs, p_i receives ⟨SECOND good m⟩ from all processes, and delivers m immediately (two steps in total, Fig. 6a).

When p_i receives $n - f$ ⟨SECOND * m⟩ messages, it checks whether all of them are "good" and no conflicting m' has reached this stage before. The appropriate ⟨THIRD good m *⟩ or ⟨THIRD bad m *⟩ message is broadcast. In the "good" case, p_i adds m to $quick_i$, the set of messages that *may* be delivered without using the underlying Atomic Broadcast protocol. In the "bad" case, the ⟨THIRD⟩ message also contain the set of "quick" messages conflicting with m.



(a) No conflicts, no failures: delivery in 2 steps. Line 14 gets executed before line 10, but this order does not matter.

(b) No conflicts, one failure: delivery in 3 steps. Note that some processes can deliver earlier than others.

(c) Message m conflicts with a previously gbcast m', the latency depends on the underlying Atomic Broadcast protocol.

Fig. 6. Several runs of the Generic Broadcast protocol from Fig. 5 with $n = 3$ and $f = 1$. The apparent synchrony in the diagrams is only for illustrative purposes.

When p_i receives $n - f$ messages ⟨THIRD good m $\emptyset$⟩, it delivers m straight away (three steps in total, Fig. 6b). Thus, if m is delivered in this way, $m \in \mathit{quick}_j$ for at least $n - f$ processes p_j. As a result, for any $m' \in C(m)$, all messages ⟨THIRD * m' $\mathit{conflicts}_j$⟩ broadcast by these processes have $m \in \mathit{conflicts}_j$. Since $n > 2f$, any two groups of $n - f$ processes overlap, so any process p_i receiving $n - f$ messages ⟨THIRD * m' $\mathit{conflicts}_j$⟩, will have $m \in \mathit{conflicts}_j$ for at least one of them.

When p_i receives $n - f$ messages ⟨THIRD * m' $\mathit{conflicts}_j$⟩, not all "good", it atomically broadcasts m' along with the union *conflicts* of all $n - f$ received sets $\mathit{conflicts}_j$. As explained above, *conflicts* contains all messages $m \in C(m')$ that might be delivered in lines 10 or 18. Processes deliver m' in line 22 only after delivering all $m \in \mathit{conflicts}$ in line 21. Otherwise, different processes could deliver m' (line 22) and messages $m \in \mathit{conflicts}$ (line 18) in different orders.

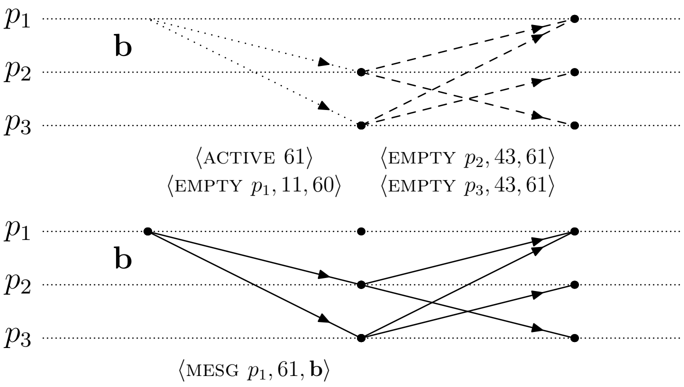


Fig. 7. Typical message patterns involved in a single abcast

In conflict-free runs, no "bad" messages are sent, so all messages are delivered in at most three steps (Figs. 6ab). Figure 6c shows that runs with conflicting messages can have a higher latency, which depends on the latency of the underlying Atomic Broadcast. Finally, observe that FIFO broadcast used for ⟨SECOND⟩ messages ensures that, if some process delivered m in line 10, no $m' \in C(m)$ will reach line 12 before m, so all correct processes will deliver m line 10 or 18.

6 Discussion

6.1 Message Complexity

Figure 7 shows messages related to abcasting a single message **b**. The upper diagram shows ordinary broadcast traffic related to ⟨ACTIVE⟩ and ⟨EMPTY⟩. The lower one presents a typical message pattern involved in Generic Broadcast of ⟨MESG $p_1, 61, \mathbf{b}$⟩. All messages in the upper diagram can therefore be piggybacked on the corresponding messages in the lower diagram. Thus, the message complexity of the Atomic Broadcast algorithm described here is the virtually the same as that of the underlying Generic Broadcast algorithm. This holds despite Atomic Broadcast achieving a two-step latency in *all* good runs, even in those with conflicting messages.

6.2 Inaccurate Clocks

In Fig. 4, assume p_3's clock is skewed and it timestamps **d** with 23 instead of 43. Message **c**, timestamped 32, will be delivered after **d**, timestamped with 23. If **d** is delivered in exactly two steps (time $123 = 43 + 2 \cdot 40$), then **c** will be delivered in two steps and $43 - 32 = 11$ units of time ($123 = 32 + 2 \cdot 40 + 11$). In general, the latency will be at most two communication steps plus the maximum clock difference Δ between two processes. By updating the clock whenever a process receives a message "from the future", one can ensure that Δ is at most one communication step. In the worst case, this reduces real-time clocks to scalar clocks [7], and ensures the latency of at most three steps in failure-free runs with inaccurate clocks.

6.3 Latency in Runs with Initial Failures

Consider a run in which all incorrect processes crash at the beginning and failure detectors do not make mistakes. In such runs, the current leader issues negative statements on behalf of the crashed processes. As opposed to negative *self*-statements, which use ordinary broadcast and are delivered in one step, the leader-issued ones use Generic Broadcast and take three steps to be delivered (or two if $n > 3f$). Therefore, the total latency in runs with initial/past failures grows from two to four steps (or three if $n > 3f$).

One communication step can be saved by eliminating line 11 from Fig. 3, so that every process (not only the current leader) can gbcast negative ⟨EMPTY⟩ statements on behalf of processes it suspects. In particular, any sender can now execute line 13 immediately after abcasting in lines 2–6, without waiting for the leader to receive its ⟨ACTIVE⟩ message in line 7. This saves one communication step and reduces the total latency to that of Generic Broadcast (two steps if $n > 3f$ and three otherwise).

6.4 Latency in Runs with General Failures

We have just seen that in runs with reliable failure detection, the total latency can be increased by at most one step. The picture changes considerably when failure detectors start making mistakes. First, if a crashed process is not (yet) suspected, no new messages can be delivered because the required ⟨EMPTY⟩ statements are not broadcast. Second, if a correct process is wrongly suspected, Generic Broadcast must deliver conflicting statements (from both the sender and the leader), which can significantly slow it down (Fig. 6c).

The above two problems cannot be solved at the same time: short failure detection timeouts motivated by the first increase the frequency of the other. To reduce the resultant high latency, the following technique can be used. When a process believes to be suspected, it should use the leader to abcast messages on its behalf rather than abcast them directly (a scheme similar to that in Sect. 4.3).

7 Lower Bounds

The Atomic Broadcast algorithm presented in this paper requires a majority of correct acceptors ($n > 2f$) and the $\Diamond S$ failure detector. These requirements are optimal [2], as is the latency of two communication steps [11]. Additional lower bounds hold for algorithms, such as this one or [13], which guarantee the latency lower than three communication steps in all good runs: the extended version of this paper [15] proves that no Atomic Broadcast algorithm can guarantee latency lower than three steps in runs in which (i) processes do not have access to real time clocks, or (ii) external processes are allowed to abcast (the open-group model [4]), or (iii) a (non-leader) process fails.

Conditions (ii) and (iii) represent a trade-off between two-step and three-step algorithms. The latter usually allow external processes to broadcast, and guarantee good performance if at most f non-leader processes fail. On the other hand, two-step delivery requires synchronized real-time clocks, all processes correct, and no external senders. (External processes can still abcast by using the current leader as a relay, but this incurs an additional step.)

The algorithm shown here relies on $\Diamond S$ to ensure that faulty processes do not hamper progress. I suspect that Ω alone is insufficient to achieve a two-step latency in all good runs, however, this is still an open question.

8 Conclusion

The Atomic Broadcast algorithm presented in this paper uses local clocks to timestamp all abcast messages, and then delivers them in order of these timestamps. Processes broadcast both positive and negative statements ("message abcast" vs. "no messages abcast"). For fault-tolerance, the leader can communicate negative statements on behalf of processes it suspects to have crashed.

Negative self-statements do not conflict with anything, so they are announced using ordinary broadcast. Other statements are communicated using a new Generic Broadcast protocol, which ensures a two-step latency in conflict-and-failure-free runs, while requiring only $n > 2f$. Since no statements conflict in good runs, the Atomic Broadcast protocol described in this paper delivers all messages in two steps. Interestingly, this speed-up is achieved with practically no message overhead over the underlying Generic Broadcast. As opposed to [13], no network traffic is generated if no messages are abcast.

Although the presented algorithm is always correct (safe and live), it achieves the optimum two-step latency only in runs with synchronized clocks, no external processes, and no failures. These three conditions are required by any two-step protocol, which indicates an inherent trade-off between two- and three-step Atomic Broadcast implementations. It also poses an interesting question: how much power exactly do (possibly inaccurate) real-time clocks add to the asynchronous model?

Acknowledgements

I would like to thank the referees for many insightful comments and suggestions that allowed me to improve this paper.

References

1. Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty Generic Broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 268–282, Toledo, Spain, 2000.
2. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

3. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
4. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.
5. Paul Ezhilchelvan, Doug Palmer, and Michel Raynal. An optimal Atomic Broadcast protocol and an implementation framework. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 32–41, January 2003.
6. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.
7. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
8. Achour Mostéfaoui and Michel Raynal. Low cost Consensus-based Atomic Broadcast. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, pages 45–52. IEEE Computer Society, 2000.
9. NTP. Network Time Protocol, 2006. URL http://www.ntp.org/.
10. Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.
11. Fernando Pedone and André Schiper. On the inherent cost of Generic Broadcast. Technical Report IC/2004/46, Swiss Federal Institute of Technology (EPFL), May 2004.
12. Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 94–108, 1999.
13. Pedro Vicente and Luís Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of 21st Symposium on Reliable Distributed Systems*, Osaka, Japan, 2002. IEEE Computer Society.
14. Piotr Zieliński. Optimistic Generic Broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.
15. Piotr Zieliński. Low-latency Atomic Broadcast in the presence of contention. Technical Report UCAM-CL-TR-671, Computer Laboratory, University of Cambridge, July 2006. Available at http://www.cl.cam.ac.uk/TechReports/.

Oblivious Gradient Clock Synchronization

Thomas Locher and Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK),
ETH Zurich, 8092 Zurich, Switzerland
{lochert, wattenhofer}@tik.ee.ethz.ch

Abstract. We study the *gradient clock synchronization* (GCS) problem, in which the worst-case clock skew between neighboring nodes has to be minimized. In particular, we consider *oblivious* clock synchronization algorithms which base their decision on how to adapt the clock solely on the most accurate timing information received from each neighbor. For several intuitive clock synchronization algorithms, which attempt to minimize the skew at all times, we show that the clock skew between neighboring nodes can be significantly larger than the proven lower bound of $\Omega(\frac{\log D}{\log \log D})$, where D denotes the diameter of the network. All of these natural algorithms belong to the class of oblivious clock synchronization algorithms. Additionally, we present an oblivious algorithm with a worst-case skew of $O(d + \sqrt{D})$ between any two nodes at distance d.

Keywords: Distributed algorithms, synchronization protocols, asynchronous computation.

1 Introduction

Due to the rapidly growing popularity of distributed systems, such as the Internet or wireless networks, and the sizable amount of applications running on those systems, the classical problem of synchronizing distributed clocks has further gained in importance in the last few years. The objective of a distributed clock synchronization algorithm is to ensure that all participating nodes in the system acquire a common notion of time. In a distributed system, nodes can accomplish this goal by perpetually sending messages containing information about the current clock value to the neighboring nodes.

Nodes are equipped with a hardware clock with bounded drift. According to the current hardware clock value and the messages received from all neighboring nodes, a logical clock value is computed. The skew between the logical clocks is to be minimized. Previous work has focused primarily on minimizing the clock skew between any two nodes in the system, while inducing a moderate message overhead. Hence, the goal of past work was to ensure that clocks are well synchronized *globally*. The resilience of these algorithms in the presence of node and network failures is another aspect of distributed clock synchronization that has been studied extensively.

For several distributed applications, such as TDMA, it is mandatory that the clocks between any node and all nodes in its vicinity do not deviate considerably

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 520–533, 2006.

from each other. This so called *gradient* property for clock synchronization was introduced in [2]. The gradient property requires that nodes that are close by have to be closely synchronized, whereas the skew between clocks of faraway nodes is allowed to be larger.

The main question is what bound on the skew between nearby nodes can be achieved by any clock synchronization algorithm. It can be shown that the skew between two nodes at distance d cannot be synchronized better than $\Omega(d)$ by using simple indistinguishability type arguments. Surprisingly, the skew between two neighboring nodes, i.e. nodes at distance 1, cannot be guaranteed to be constant. The lower bound on the worst-case clock skew proven in [2] is $\Omega(\frac{\log D}{\log \log D})$, where D is the diameter of the network.

This lower bound holds even if all nodes have full knowledge of the complete message history. However, for practical algorithms it is reasonable to assume that nodes cannot store the entire history of messages ever received. As time progresses, nodes will be forced to delete outdated information. We study clock synchronization in a restricted model in which each node is only allowed to store the largest clock value ever received from each neighbor. It is natural to restrict the stored information to these values because any algorithm attempting to minimize the skew at all times will set the clock in accordance with these clock values only, due to the lack of information about the message delays and the progress each node might have made in the meantime. Since these algorithms are unaware of the communication process and determine the local clock strictly by considering the largest clock values received from all neighboring nodes, we call these algorithms *oblivious*. Studying oblivious algorithms has a long tradition in distributed computing and computer science in general. Oblivious algorithms are examined for various reasons, one being that they can give valuable insights into problems that are hard to tackle in the general case. A more practical reason is that they are normally easier to realize in hardware. Another motivation to explore oblivious algorithms is that several oblivious algorithms perform well in their respective domains, for example routing or sorting algorithms. It is therefore worthwhile to determine the effect of obliviousness on clock synchronization.

Several fundamentally different strategies can be employed in order to minimize the skew at all times. As nodes must generally strive to catch up with the faster nodes, nodes can minimize the skew to the fastest node[1] by setting the clock to the largest clock value. A different approach is to minimize the skew to all neighbors at all times. A third method that is worth investigating is minimizing the skew to the slowest node. If every node waits under certain circumstances for the slower nodes to catch up, the skew might be kept within reasonable bounds. We will study algorithms devoted to each of these objectives and show that all of them fail to provide a low bound on the worst-case skew between neighbors.

However, these observations enable us to devise an oblivious algorithm with a worst-case skew of $O(d+\sqrt{D})$ between any two nodes at distance d, if the nodes

[1] The nodes with the currently largest and smallest logical clock values are called the fastest and the slowest node, respectively.

are aware of the diameter D of the network and adjust their clock synchronization mechanism accordingly. To the best of our knowledge, it is the first gradient clock synchronization algorithm guaranteeing a worst-case skew of $o(D)$ between neighboring nodes.

After briefly summarizing related work on clock synchronization in Section 2, we formally specify the model used in this paper in Section 3. Subsequently, we propose gradient clock synchronization algorithms which minimize the skew to the neighboring nodes according to the aforementioned strategies and analyze their behavior in specific executions. This is the subject of Section 4. In Section 5, the $O(d + \sqrt{D})$-GCS algorithm is presented and analyzed.

2 Related Work

The fundamental problem of clock synchronization has been studied extensively and many theoretical results have been published. Srikanth and Toueg [7] presented a clock synchronization algorithm which minimizes the maximum skew between any pair of nodes, given the hardware clock drift. In every possible execution, their algorithm ensures that the skew between any two nodes is at most $\Theta(D)$, which is asymptotically optimal. However, their algorithm also incurs a skew of $\Theta(D)$ between neighboring nodes in the worst case.

While there has been a lot of research on bounds for the skew and the communication costs [3,5,6] and also on the capability of clock synchronization algorithms to cope with both node and network failures [7], the gradient property has not been studied until the remarkable work by Fan and Lynch [2].

An algorithm $\mathcal{A}$ is said to be an f-GCS algorithm, if for all nodes i and j the clock skew between node i and j is at most $f(d_{i,j})$ at all times, where $d_{i,j}$ denotes the *distance* between node i and j.

Fan and Lynch prove that there is an execution after which the skew between two neighboring nodes is at least $\Omega(\frac{\log D}{\log\log D})$, independent of the chosen algorithm. They consider a linear network of n nodes, thus $D = n - 1$. Their proof relies on the fact that a node cannot increase its clock too quickly, i.e. by more than $O(f(1))$ in $O(1)$ time, otherwise it would violate the gradient property in a different execution that is indistinguishable from the original execution. The skew between all neighbors among k nodes can be increased by $O(1)$ in $O(k)$ time, which can be shown using again an indistinguishability type argument. In their execution, they build up a constant skew c_1 in time $O(n)$ between all n nodes. In the next step, the execution continues to run for $O(\frac{n}{f(1)})$ time, which means that the average skew during this time can only be reduced by a constant c_2 between each pair of neighboring nodes. The parameters can be chosen such that $c_1 > c_2$ and thus the skew is still larger than before this round. During the same time span, the skew between neighbors of a set of $O(\frac{n}{f(1)})$ nodes can again be increased by a constant. This procedure can be repeated recursively $\log_{f(1)} n$ times and since $n = D - 1$ and $f(1) \in \Omega(\log_{f(1)} n)$, it follows that $f(1) \in \Omega(\frac{\log D}{\log\log D})$. Meier and Thiele [4] showed that this bound also holds for a different model in which the delay of each message is 0, but the communication frequency is bounded.

An open problem for gradient clock synchronization is whether this lower bound is *tight* or whether an algorithm cannot even reach this bound asymptotically. We will show that many natural clock synchronization algorithms do not even come close to this bound. In the context of clock synchronization for wireless networks, Fan et al. show that when nodes occasionally receive a message informing them of the correct real time using a GPS service, the skew between any two nodes can be bounded by a small constant $\epsilon > 0$ "some of the time" [1]. Thus, this algorithm only satisfies a weakened version of the gradient property. We show that there is a general gradient clock synchronization algorithm for which it holds that the skew between nodes at distance d is bounded by $O(d + \sqrt{D})$ at all times. This is the first non-trivial upper bound for gradient clock synchronization.

3 Model

We consider an arbitrary graph $G = (V, E), V = \{1, \ldots, n\}$, where $|V| = n$ and $E \subseteq V \times V$. Any node i can communicate with any node j to which node i is directly connected, i.e. $\{i, j\} \in E$. These nodes are referred to as the neighboring nodes or neighbors of node i. Let $\mathcal{N}_i$ denote the set of all neighboring nodes of node i. The communication between neighboring nodes is reliable, but messages can have variables delays in the range $[0, 1]$. The *distance* between nodes i and j is defined as the length of the shortest path between those two nodes. The *diameter* D of G is the maximum distance between any two nodes.

Each node $i \in V$ is equipped with a *hardware clock* $H_i(\cdot)$ whose value at time t is $H_i(t) := \int_0^t h_i(\tau)d\tau$, where $h_i(\tau)$ is the *hardware clock rate* of node i at time τ. For all nodes $i \in V$ and all times t, it holds that $h_i(t) \in [\mathcal{L}, \mathcal{U}]$, where $0 < \mathcal{L} < \mathcal{U}$. The degree of synchronization that can be achieved is related to the maximum message delay. It is therefore reasonable to assume that a fast processor can increase its clock by at least 1, if it takes up to 1 time before a message arrives, therefore we assume $\mathcal{U} \geq 1$.

In addition to the hardware clock, each node i further has a *logical clock* $L_i(\cdot)$. As long as no new messages arrive, the logical clock value increases at the rate of the hardware clock. This implies that all nodes steadily make progress. A clock synchronization algorithm $\mathcal{A} : L \times \Psi \rightarrow L$ specifies how the logical clock $L_i(t)$ of node i at time t is adapted according to the current value of the logical clock and the message history $\Psi_i(t)$ of node i at time t. The algorithms are *reactive* in that they perform this update on the logical clock whenever a message from a neighboring node arrives. Let $\tilde{L}_j(t)$ denote the maximum logical clock value ever received from neighboring node j.[2] For every algorithm $\mathcal{A}$ it must hold that

$$\forall i \forall t : L_i(t) \leq \mathcal{A}(L_i(t), \Psi_i(t)) \leq \max_{j \in \mathcal{N}_i} \tilde{L}_j(t).$$

[2] Note that this might not be the *last* message received from node j, as the communication network does not necessarily satisfy the FIFO condition.

As the logical clock is not allowed to run backwards, the algorithm can either increase the logical clock or leave it at the current value. Moreover, the algorithm can set the logical clock at most to the maximum logical clock value it has ever received. If a node i set its logical clock to a value exceeding any value it has ever received from a neighbor, a neighboring node j could potentially increase its logical clock value even more, based on the new clock value of node i etc. resulting in a large clock skew between some nodes. We further assume that the adaptation of the logical clock through $\mathcal{A}$ requires 0 time.

As mentioned before, we focus on the class of oblivious clock synchronization algorithms. Nodes only store the reduced history $\tilde{\Psi}_i(t) := \{\tilde{L}_j(t)\}^{j \in \mathcal{N}_i}$, i.e. the largest clock values received from each neighboring node is stored. Whenever a message is received, these values are updated and, subsequently, the logical clock is computed according to a function on $\tilde{\Psi}_i(t)$. Naturally, the old logical clock value also has to be considered, since clocks can only make progress. In case the computed value does not exceed the old logical clock value, the logical clock simply remains unchanged, otherwise the logical clock is updated and the new logical clock value is broadcast immediately to all neighbors.

An *execution* is a tuple $\mathcal{E} = (\mathcal{M}, \mathcal{R})$, where $\mathcal{M} : \mathcal{T} \times V \times V \rightarrow [0,1]$ defines the message delays and the integrable function $\mathcal{R} : \mathcal{T} \times V \rightarrow [\mathcal{L}, \mathcal{U}]$ determines the hardware clock rates of each node. Hence $\mathcal{M}(t,i,j)$ specifies how long it takes for the message sent by node i at *real time* t to arrive at j and $h_i(t) := \mathcal{R}(t,i)$.

For any gradient clock synchronization algorithm, the goal is to ensure a small logical clock skew between neighboring nodes, i.e. a gradient clock synchronization algorithm strives to minimize $\max_{i,j \in \mathcal{N}_i, t} |L_i(t) - L_j(t)|$ over all possible executions $\mathcal{E}$ for every graph G. In the following section, we present natural clock synchronization algorithms and bounds on the induced worst-case skew.

4 Algorithms and Bounds

Throughout this section, we consider the graph G^{list} consisting of a linear list of n nodes, i.e. $|V| = n$ and $E^{list} = \{\{1,2\},\{2,3\},\ldots,\{n-1,n\}\}$. This simple graph is suitable to show that there are executions for all presented algorithms leading to a large skew between neighboring nodes.

Initially, all logical clock values are 0. We assume that, at real time 0, node n sends a *start message* to its neighbor and starts its logical clock. Every node that receives a start message for the first time also starts its clock and broadcasts the start message. For the purpose of synchronization, each node regularly informs all neighbors about its current logical clock value. In particular, we assume that every node transmits a message containing its ID and its logical clock value to all neighboring nodes when its logical clock reaches an integer value or when the logical clock is updated due to a received message. If a node has not yet received a message from a certain neighbor, it assumes that this neighbor's logical clock is still at 0.

Note that this initialization process is used for the sake of simplicity. The same bounds also hold asymptotically for other start-up procedures. By proving that the proposed algorithms incur a large skew among neighboring nodes in G^{list} using the stated initialization process, we can conclude that they are poor gradient clock synchronization algorithms in general.

4.1 Minimizing the Skew to the Fastest Neighbor

A straightforward algorithm, denoted $\mathcal{A}^{max}$, always sets the logical clock to the largest clock value ever received, if this value exceeds the current logical clock value. More formally, the logical clock value of node i is set to the value $\max(L_i(t), \max_{j \in \mathcal{N}_i} \tilde{L}_j(t))$, if it receives a message at time t. Thus, the skew to the fastest node is minimized by simply adopting the maximum clock value. It has been pointed out in [2] that $\mathcal{A}^{max}$ potentially incurs a large skew between neighboring nodes, due to the fact that a skew of $\Theta(n)$ between node 1 and n cannot be avoided and a *fast message*, i.e. a message that is transmitted with 0 delay, which is forwarded along the chain causes a skew of $\Theta(n)$ between two neighboring nodes. We will now briefly dwell on this simple algorithm in order to introduce our notation. The following execution $\mathcal{E} = (\mathcal{M}, \mathcal{R})$ induces a skew of $\Theta(n) = \Theta(D)$ between the nodes 1 and 2:

$$\mathcal{M}(t,i,j) := \begin{cases} 0 \text{ if } t \geq n-1, j \neq 1 \\ 1 \text{ else} \end{cases}$$

and $\forall t \forall i : \mathcal{R}(t,i) := 1 - \epsilon_i$, where $\epsilon_n = 0$ and $\epsilon_i > 0$ for all $i \in \{1, \ldots, n-1\}$.[3] It holds that $L_j(n-1) = n-1$ for all $j \in \{2, \ldots, n\}$, as the logical clock of node n has reached $n-1$ and this value is forwarded along the chain with a delay of 0. Since node 1 receives the start message at time $n-1$, its logical clock is still at 0 and thus the skew between node 1 and 2 is $\Theta(n)$. Note that this effect does not occur due to the fact that node 1 has merely received its start message. If there was no fast message, the logical clock of node i at time $t \in \mathbb{N}$, where $t > n-1$, would be $L_i(t) = t - (n-i)$. Setting the message delay to 0, except between nodes 1 and 2, at this point in time would still incur a skew of $\Theta(n)$ between nodes 1 and 2.

Before the fast message is sent, the clock value of node i at time $n-1$ is $i-1$. If each node allowed a slack of 1 between the clock of the fastest node and its own, the fast message would not alter any clock values. By setting the logical clock to $\max(L_i(t), \max_{j \in \mathcal{N}_i} \tilde{L}_i(t) - \gamma)$ for a particular $\gamma > 0$, it seems that the effect a fast message has in $\mathcal{A}^{max}$ can be avoided. Unfortunately, this is not the case. Let $\mathcal{R}(t,n) := \mathcal{U}$ and $\mathcal{R}(t,i) := \mathcal{L}$ for all $i \neq n$. The message delays are $\mathcal{M}(t,i,j) := 0$ for all $i, j \neq 1$ and $t \geq \vartheta$ for a specific time ϑ, and $\mathcal{M}(t,i,j) := 1$ otherwise. In this scenario, it holds that $\lim_{t \to \infty} L_i(t) - L_i(t-1) = \mathcal{U}$, as all nodes are paced by the fastest node. If node i receives the value x from node $i+1$, i sets its logical clock to $x - \gamma$. In this time, node $i+1$ has increased its clock by $\mathcal{U}$, therefore $\lim_{t \to \infty} L_{i+1}(t) - L_i(t) = \mathcal{U} + \gamma$ for all $i \in \{1, \ldots, n-1\}$. Assume that at time ϑ, this stabilization has occurred and that $L_n(\vartheta) = x$. The

[3] Node n is the fastest node and therefore sets the pace for the other nodes. The clock rates can be viewed as *relative* rates compared to the clock rate of the fastest node.

message delay at this time is reduced to 0 and thus node $n-1$ can increase its clock to $x-\gamma$. The skew between $n-1$ and $n-2$ is then $2\mathcal{U}+\gamma$, therefore node $n-2$ can increase its clock by $2\mathcal{U}$. In general, node $n-i$ will increase its logical clock by $i\mathcal{U}$ and thus the skew between node 1 and 2 at time ϑ is $(n-2)\mathcal{U} \in \Theta(n)$. Hence, this variation of $\mathcal{A}^{max}$ does not reduce the worst-case skew between neighboring nodes asymptotically.

As it is not an effective strategy to strictly minimize the skew to the fastest node, we will analyze the effect of taking the values $\tilde{L}_j(t)$ from all neighbors j into account.

4.2 Minimizing the Skew to All Neighbors

We will now consider the algorithm that sets the logical clock to the *average value* of all the neighbors' clock values in an attempt to minimize the clock skew to all neighbors at all times, i.e. node i sets its logical clock to the value

$$L_i(t) := \max(L_i(t), \frac{1}{|\mathcal{N}_i|} \sum_{j \in \mathcal{N}_i} \tilde{L}_j(t))$$

upon receiving a message from a neighbor at time t. We call this algorithm $\mathcal{A}^{avg}$.

In a very simple execution, the skew in G^{list} can become very large. The execution $\mathcal{E}^{nice} = (\mathcal{M}^{nice}, \mathcal{R}^{nice})$ is defined as follows. $\forall t \forall i \forall j : \mathcal{M}^{nice}(t,i,j) := 1$ and $\forall t \forall i : \mathcal{R}^{nice}(t,i) = 1-\epsilon_i$, where again $\epsilon_n = 0$ and $\epsilon_i > 0$ for all $i \in \{1,\ldots,n-1\}$. Since the hardware clock rates never change, the message delays are the same at any point in time and identical between any two neighboring nodes, one might assume that the skew between neighbors cannot become exceedingly large. Surprisingly, this is not true, as we will prove now.

Lemma 1. *Let $\mathcal{A}^{avg}$ be the clock synchronization algorithm in use. When executing $\mathcal{E}^{nice}$, it holds that $\forall t \forall i \in \{2,\ldots,n\} : L_i(t) - L_{i-1}(t) \leq 2i-3$, independent of the choices of $\epsilon_i > 0$.*

Proof. First, we define $\Delta L_i(t) := L_i(t) - L_i(t-1)$. It holds that $\forall i \forall t : \Delta L_i(t) \leq 1$, as the average speed is upper bounded by the maximum hardware clock rate, which is 1 in this particular execution. It immediately follows that $L_i(t+k) \leq L_i(t) + k$.

We have that $L_1(t) = L_2(t-1)$, as node 2 is the only neighbor of node 1. If node 1 is informed about a higher value, it can increase its logical clock immediately to this value. Since $\Delta L_2(t) \leq 1$ for all t, it follows that $L_2(t) - L_1(t) \leq 1$ for all t. Assume that it holds for all t and all $j \leq i$ that $L_j(t) - L_{j-1}(t) \leq 2j-3$. We will now prove a bound on the skew between node i and $i+1$. For $t=0$, it is trivially true that $L_{i+1}(t) - L_i(t) \leq 2(i+1)-3$. Assume that it holds for all $t' \leq t$. For $t+1$, we have that

$$\begin{aligned} L_i(t+1) &\geq \frac{L_{i+1}(t) + L_{i-1}(t)}{2} \\ &\geq \frac{L_{i+1}(t) + L_i(t) - (2i-3)}{2} \end{aligned}$$

$$\geq \frac{L_{i+1}(t) + L_i(t+1) - 1 - (2i-3)}{2}$$
$$\geq L_{i+1}(t+1) - (2(i+1) - 3).$$

The first inequality holds because the logical clock value is always at least the average value of its neighbors. The second inequality follows by induction and the third and fourth inequalities hold because $\Delta L_i(t) \leq 1$. □

Lemma 1 shows that the skew between any two nodes is bounded, when executing $\mathcal{E}^{nice}$. In order to prove that the skew can in fact become large, we need another lemma.

Lemma 2. $\forall i \in \{1, \ldots, n\} : \lim_{t\to\infty} \Delta L_i(t) = 1$.

Proof. Assume that $\Delta L_{n-1}(t)$ does not converge to 1. In this case, either there is an $\epsilon > 0$ such that for all t it holds that $\Delta L_{n-1}(t) \leq 1-\epsilon$ or $\Delta L_{n-1}(t) = 1$ only for some t. By definition of $\mathcal{E}^{nice}$, $\Delta L_n(t)$ is always 1. If there is such an $\epsilon > 0$, this would imply that $\lim_{t\to\infty} L_n(t) - L_{n-1}(t) = \infty$, which is a contradiction to Lemma 1. If for some t we have $\Delta L_{n-1}(t) = 1$, but the value never converges to 1, there is an unbounded number of times t' where $\Delta L_{n-1}(t') < 1$, which also implies that $\lim_{t\to\infty} L_n(t) - L_{n-1}(t) = \infty$, again a contradiction. Hence, $\lim_{t\to\infty} \Delta L_{n-1}(t) = 1$ and, applying the same argument to the other nodes, it follows inductively that $\lim_{t\to\infty} \Delta L_i(t) = 1$ for all nodes $i \in \{1, \ldots, n\}$. □

We are now in the position to prove the following theorem.

Theorem 1. *Let $\mathcal{A}^{avg}$ be the clock synchronization algorithm in use. When executing $\mathcal{E}^{nice}$, the largest skew between neighbors in G^{list} is $2n - 3 \in \Theta(n)$.*

Proof. In particular, we show that $\lim_{t\to\infty} L_i(t) - L_{i-1}(t) = 2i - 3$ for all $i \in \{2, \ldots, n\}$. Since $L_1(t) = L_2(t-1)$, it holds that $\lim_{t\to\infty} L_2(t) - L_1(t) = \lim_{t\to\infty} \Delta L_1(t{+}1) = 1$, according to Lemma 2. We assume now that $\lim_{t\to\infty} L_j(t) - L_{j-1}(t) = 2j - 3$ for all $j \leq i$. Lemma 4.2 states that $\lim_{t\to\infty} L_{i+1}(t) - L_i(t) = \mathcal{Q}$ for a constant $\mathcal{Q}$ which is upper bounded by $2(i+1) - 3$, due to Lemma 1. If $\mathcal{Q} < 2(i+1) - 3$, we get that

$$\lim_{t\to\infty} L_i(t) = \lim_{t\to\infty} \frac{L_{i-1}(t-1) + L_{i+1}(t-1)}{2}$$
$$= \lim_{t\to\infty} \frac{2L_i(t-1) - (2i-3) + \mathcal{Q}}{2}$$

and thus $\lim_{t\to\infty} \Delta L_i(t) < 1$, a contradiction to Lemma 2. □

Note that the skew between node 1 and n is $\Theta(n^2)$, which is worse than the tight upper bound of $\Theta(n)$ skew between any two nodes for $\mathcal{A}^{max}$. This execution shows that the neighboring node with the fastest clock must have a weight larger than the weight of all other neighbors together. To see this, consider a *k-ary tree* where the root is the fastest node. The skew between neighboring nodes will also be large when executing $\mathcal{E}^{nice}$ if all k children together have a weight that is at least the weight of the parent node, as this is equivalent to performing this execution on the linear list where the weight of the higher indexed node, i.e. the faster node, is not larger than the weight of the lower indexed node. We proved that in this case the skew is $\Theta(D)$ between neighboring nodes.

4.3 Minimizing the Skew to the Slowest Neighbor

In this section, we present a different approach which actively tries to bound the skew between neighbors. As proven in [2], a constant bound between neighboring nodes cannot be maintained. However, introducing a constant bound might still result in a significantly improved worst-case behavior.

The algorithm $\mathcal{A}^{bound}$ increases the logical clock proactively to the maximum value of all neighbors as long as the clock skew between its own logical clock and the clock of any of its neighbors does not exceed a predefined constant bound $\mathcal{B}$. Once the skew between node i and a neighbor j is at least $\mathcal{B}$ according to the state information about node j, i.e. $L_i(t) - \tilde{L}_j(t) \geq \mathcal{B}$, node i does not increase its logical clock due to an external message again until node j has caught up.[4] $\mathcal{A}^{bound}$ is immune to both the execution $\mathcal{E}^{nice}$ and also the execution that incurred a large clock skew when $\mathcal{A}^{max}$ is used. Nevertheless, $\mathcal{A}^{bound}$ is not better asymptotically in the worst case. The idea behind the adversarial schedule for this specific algorithm is the following. Using fast messages, a chain of nodes within the graph is constructed such that the skew between all neighboring nodes in this chain is $\mathcal{B}$, creating a chain of dependency. Consequently, each node has to wait for his slower neighbor to catch up, resulting in long waiting times before the logical clocks can again be increased.

The execution $\mathcal{E}^{fast} = (\mathcal{M}^{fast}, \mathcal{R}^{fast})$, given the constant bound $\mathcal{B}$, is defined as follows. We set

$$\mathcal{M}^{fast}(t,i,j) := \begin{cases} 0 \text{ if } t = n-1, j \neq 1 \\ 1 \text{ else.} \end{cases}$$

Let $\hat{\imath} := \lfloor \frac{n-1}{\mathcal{B}} \rfloor + 1$. The hardware clock rates are $\forall t \forall i \neq \hat{\imath}+1 : \mathcal{R}^{fast}(t,i) := 1 - \epsilon_i$, where again $\epsilon_n = 0$ and $\epsilon_i > 0$ for all $i \in \{1, \ldots, n-1\}$. $\forall t < n-1 : \mathcal{R}^{fast}(t, \hat{\imath}+1) := 1 - \epsilon_{\hat{\imath}+1}, \forall t \geq n-1 : \mathcal{R}^{fast}(t, \hat{\imath}+1) := 1$. Thus, the hardware clock of node $\hat{\imath}+1$ is sped up at time $n-1$.

Note that the delay of messages is 0 at time $n-1$, unless they are sent to node 1. As local computation requires 0 time in this asynchronous computation model, some nodes can potentially communicate an unbounded number of times, while other nodes wait for the arrival of some particular messages. As far as clock synchronization is concerned, this entails that the clock values of nodes whose communication links have a delay of 0 at a specific time t will *stabilize* according to the clock synchronization algorithm in use. Such a stabilization always occurs independent of the clock synchronization algorithm, since logical clocks can only make progress, but the logical clock values cannot exceed the maximum clock value. This characteristic of asynchronous communication is exploited in the execution $\mathcal{E}^{fast}$.

[4] Note that node i still makes progress at the rate of its hardware clock.

Lemma 3. *Let $\vartheta := n + \lfloor \frac{n-1}{\mathcal{B}} \rfloor - \kappa$, $0 < \kappa < 1$. When executing $\mathcal{E}^{fast}$, parameterized by $\mathcal{B}$, the skew between nodes $\hat{\imath} := \lfloor \frac{n-1}{\mathcal{B}} \rfloor + 1$ and $\hat{\imath} + 1$ at time ϑ is at least $\left(\lfloor \frac{n-1}{\mathcal{B}} \rfloor + 1 - \kappa\right) \epsilon_{\hat{\imath}}$.*

Proof. For any $\mu < 1$ and $i \geq 2$, we have $L_i(n-2+\mu) = i-2+\mu\epsilon_i$. Node 1 does not start its logical clock before time $n-1$. At time $n-1$, all communication between the nodes $i \in \{2, \ldots, n\}$ requires 0 time. Note that we do not need to specify which messages are handled first. The outcome of this stabilization process solely depends on the clock synchronization algorithm. In this case, after the clocks have stabilized, it holds that

$$L_i(n-1) := \begin{cases} (i-1)\mathcal{B} & i \in \{1, \ldots, \hat{\imath}\} \\ n-1 & \text{else} \end{cases}$$

as node 1 can increase its logical clock to $\mathcal{B}$ and consequently, node 2 can raise its clock to $2\mathcal{B}$ etc. At this point, the clock rate of node $\hat{\imath} + 1$ is set to 1, which means that $L_{\hat{\imath}+1}(\tau) = \tau$ for all $\tau \geq n-1$. Node 1 receives the message that the logical clock of node 2 is already at $\mathcal{B}$ at time n and subsequently increases its own clock to this value. In general, node j has to wait until time $n+j-1$ before it can increase its logical clock by $\mathcal{B}$. Accordingly, node $\hat{\imath}$ has to wait until time $n + \lfloor \frac{n-1}{\mathcal{B}} \rceil > \vartheta$ before it can increase its logical clock by $\mathcal{B}$.

Since $L_{\hat{\imath}}(n-1) \leq n-1$, we have that $L_{\hat{\imath}}(\vartheta) \leq n-1+(1-\epsilon_{\hat{\imath}})(1-\kappa+\lfloor \frac{n-1}{\mathcal{B}} \rfloor)$. Hence $|L_{\hat{\imath}+1}(\vartheta) - L_{\hat{\imath}}(\vartheta)| \geq \epsilon_{\hat{\imath}} \left(\lfloor \frac{n-1}{\mathcal{B}} \rfloor + 1 - \kappa\right)$. □

The following theorem is immediate from Lemma 3.

Theorem 2. *Let $\mathcal{A}^{bound}$ be the clock synchronization algorithm in use. When executing $\mathcal{E}^{fast}$, the skew between neighbors in G^{list} can be at least $n\frac{\epsilon_{\hat{\imath}}}{\mathcal{B}} - (\frac{\epsilon_{\hat{\imath}}}{\mathcal{B}} + 1) \in \Theta(n)$.* □

It is a strong assumption that some nodes can communicate an unbounded number of times while other nodes are not making any progress. If only a constant number of communication rounds were possible, the skew would be constant in this execution. However, the result is quite counterintuitive, as one might assume that the more and the faster nodes can communicate, the better clocks can be synchronized in general.

This theorem shows that the resulting skew can be large even though a constant bound has been specified. It turns out that the constant bound potentially results in waiting times that incur a skew of much more than the specified bound. In general, for any bound $\mathcal{B}$, it holds that $\exists t, i, j \in \mathcal{N}_i : |L_i(t) - L_j(t)| = \mathcal{B}$. If the delay is 0 between a set of nodes in a particular execution, this results in a chain of nodes with clock values $x, x+\mathcal{B}, x+2\mathcal{B}, \ldots, O(D)$. If the length of this chain is λ, then the worst-case skew between two neighbors can be at least $\Omega(\lambda)$, because all nodes in the chain are constrained to wait for the slower node in the chain. The length of this chain can be $\Theta(\frac{D}{\mathcal{B}})$ when the entire skew of $\Theta(D)$ is allocated. Hence it follows that the skew between neighboring nodes can be at least $\Omega(\frac{D}{\mathcal{B}})$. According to the bounds maintained by neighboring nodes, a node might adapt its bound in order to adjust to this situation. Using a smaller bound than the current bound of one of the neighbors does not help, as the chain becomes even longer in the worst case, resulting in a larger worst-case skew. A node might allow a skew of $\delta\mathcal{B}$ for any $\delta > 1$, if the maximum skew between any of its neighbors and one of this node's neighbors has already reached $\mathcal{B}$. In an execution such as $\mathcal{E}^{fast}$, the length of the chain is at most $\Theta(\log_\delta \frac{D}{\mathcal{B}})$. However, the maximum skew between neighbors is then $\Omega(\delta^{\log_\delta \frac{D}{\mathcal{B}}}) = \Omega(\frac{D}{\mathcal{B}})$, thus adapting the bounds does not improve the worst-case behavior either. Consequently, when minimizing the skew to the slowest node while increasing the clock whenever possible, the worst-case skew between neighboring nodes is always at least $\Omega(\frac{D}{\mathcal{B}} + \mathcal{B}) = \Omega(\sqrt{D})$.

5 A $O(d + \sqrt{D})$-GCS Algorithm

The idea is that the knowledge of the diameter D can be exploited by setting the bound to $O(\sqrt{D})$. If the algorithm can ensure that the skew between any two nodes is always at most $O(D)$, the skew between neighbors will be at most $O(\sqrt{D})$, because nodes do not allow a larger skew than $O(\sqrt{D})$ and the waiting time until they can raise their logical clocks again considerably is also bounded by $O(\sqrt{D})$. The algorithm which achieves this goal is described in greater detail in the following section.

5.1 Description of the Algorithm

As in the previous algorithms, the algorithm presented here, denoted by $\mathcal{A}^{root}$, also mandates the forwarding of the clock value to all neighboring nodes when the logical clock reaches an integer value. Apart from the diameter D, the algorithm must also know an upper bound on the hardware clock rate. Usually, the hardware clock rates will only differ slightly, therefore one could simply set $\mathcal{U}$ to a realistic upper bound on the maximum clock rate, if the true bound is unknown. Algorithm 1 depicts the steps taken upon receipt of a message from a neighbor.

```
if $\tau > \tilde{L}_j(t)$ then
  $\tilde{L}_j(t) := \tau$
end if
if $\max_{j \in \mathcal{N}_i}(\tilde{L}_j(t)) > L_i(t)$ and $\min_{j \in \mathcal{N}_i}(\tilde{L}_j(t)) + \mathcal{U}\sqrt{D+1} > L_i(t)$ then
  $L_i(t) := \min(\max_{j \in \mathcal{N}_i}(\tilde{L}_j(t)), \min_{j \in \mathcal{N}_i}(\tilde{L}_j(t)) + \mathcal{U}\sqrt{D+1})$
  send $\langle i, L_i(t) \rangle$ to all $j \in \mathcal{N}_i$
end if
```

Algorithm 1: Node i calls this procedure when a message $\langle j, \tau \rangle$ from node j with time stamp τ is received at time t

The information about the corresponding neighbor is updated if the newly arrived message indicates progress. Subsequently, the logical clock is increased if the slowest neighbor is not more than $\mathcal{U}\sqrt{D+1}$ behind. The logical clock is raised at most to the maximum logical clock value of all neighbors. Any message that does not cause a change of the logical clock is simply dropped.

5.2 Analysis of the Algorithm

First, we prove that the worst-case skew between any two nodes is at most $O(D)$, which is asymptotically optimal. This *global* property is further used to derive the gradient property of $\mathcal{A}^{root}$. Note that both properties hold independent of the underlying network structure, thus the algorithm effectively bounds the skew between neighboring nodes in arbitrary graphs.

Theorem 3 (Global Property). *Let $\mathcal{A}^{root}$ be the clock synchronization algorithm in use. For all executions and for any graph, it holds that $\forall i, j, t$: $|L_i(t) - L_j(t)| < \mathcal{U}D + 1 \in O(D)$.*

Proof. The crucial observation is that for the slowest node $\mathcal{A}^{root}$ is identical to $\mathcal{A}^{max}$. Recall that the slowest node is the node with the currently lowest clock value, and the node with the largest clock value is denoted the fastest node. After at most D time, the slowest node starts its clock. The progress of the fastest node is at most $\mathcal{U}D$ during this time, resulting in a skew not larger than $\mathcal{U}D$. Before the next message reaches the slowest node, the fastest node can increase its logical clock by less than 1, resulting in a skew of less than $\mathcal{U}D + 1$. Once this message reaches the slowest node, the skew drops back to at most $\mathcal{U}D$. The slowest node can increase its logical clock at least at the same speed of the fastest node, thus the skew cannot grow any further. By reducing the messages delays, the slowest node can even catch up, as it can increase its clock earlier. If the messages are sped up such that the skew between the slowest node and any of its neighbors reaches $\mathcal{U}\sqrt{D+1}$, the slowest node can instantaneously raise its logical clock by $\mathcal{U}\sqrt{D+1} > \mathcal{U}$, hence the skew again decreases in this case. □

Using this bound on the *global* skew, we can limit the waiting time for any node and thereby guarantee that the skew between neighbors is always at most $O(\sqrt{D})$.

Theorem 4 (Gradient Property). *Let $\mathcal{A}^{root}$ be the clock synchronization algorithm in use. For all executions and for any graph, it holds that $\forall i, \forall j \in \mathcal{N}_i, t$: $|L_i(t) - L_j(t)| < 2\mathcal{U}\sqrt{D+1} \in O(\sqrt{D})$.*

Proof. It is evident that the skew can only be larger than $\mathcal{U}\sqrt{D+1}$ when nodes are forced to wait for other nodes to increase their logical clocks. Let the skew between node i and j be $\mathcal{U}\sqrt{D+1}$ at time t. Without loss of generality, let $L_i(t) = L_j(t) + \mathcal{U}\sqrt{D+1}$. If there is a node $k \in \mathcal{N}_j$ such that $L_j(t) = L_k(t) + \mathcal{U}\sqrt{D+1}$, node j has to wait for node k to increase its logical clock. Node k can again have a neighbor whose logical clock is $\mathcal{U}\sqrt{D+1}$ behind etc. If this chain of dependent nodes has length λ, it takes at most λ time steps until node j can increase its logical clock by $\mathcal{U}\sqrt{D+1}$. The length λ is upper bounded by the maximum skew between any two nodes divided by $\mathcal{U}\sqrt{D+1}$. Hence, using Theorem 3, $\lambda \leq \frac{\mathcal{U}D+1}{\mathcal{U}\sqrt{D+1}} \leq \sqrt{D+1}$, because $\mathcal{U} \geq 1$. Node i cannot increase its logical clock by more than $\mathcal{U}\sqrt{D+1}$ during this time, as the maximum hardware clock rate is $\mathcal{U}$, and node j increases its logical clock by at least $\mathcal{U}\sqrt{D+1}$, thus nodes can always catch up.

Before node j can raise its logical clock after $\sqrt{D+1}$ time, i can increase its logical clock by less than $\mathcal{U}\sqrt{D+1}$, thus, at all times, the skew between any two neighbors is less than $2\mathcal{U}\sqrt{D+1}$. □

6 Conclusion

We have shown that aiming at achieving a minimal skew at all times naturally translates to oblivious algorithms, due to the fact that nodes do not have any information about the message delays and the hardware clock rates. Focusing on the fastest nodes potentially incurs a large skew between neighbors, but the fastest node must nevertheless have a large weight, as proven in our analyses. By assigning a large weight to the fastest node, the clocks will converge quickly to a large value, even if some neighboring nodes do not make any significant progress during the same time span.

However, there is an oblivious clock synchronization algorithm with a worst-case skew of $O(d + \sqrt{D})$ between any two nodes at distance d, which answers the question whether there is an GCS algorithm with a skew of $o(D)$ between neighboring nodes. This algorithm further guarantees a skew of $\Theta(D)$ between any two nodes, which is globally asymptotically optimal.

A challenging open problem is whether the bound of $\Theta(\sqrt{D})$ skew between neighbors is asymptotically optimal for oblivious algorithms. Additionally, it is also worth investigating how much more knowledge, e.g. the times when messages arrived or a larger message history in general, is required in order to substantially improve the worst-case skew. Another important aspect of clock synchronization is the number of messages required in order to effectively bound the skew between nodes. Analyzing the message complexity of gradient clock synchronization algorithms is another demanding problem which has not been studied so far.

Acknowledgements

We would like to thank Regina O'Dell and Thomas Moscibroda for many fruitful discussions and all the anonymous reviewers for their constructive comments and suggestions which helped to improve this paper.

References

1. R. Fan, I. Chakraborty, and N. Lynch. Clock Synchronization for Wireless Networks. In *Proc. 8th International Conference on Principles of Distributed Systems (OPODIS)*, pages 400–414, 2004.
2. R. Fan and N. Lynch. Gradient Clock Synchronization. In *Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 320–327, New York, NY, USA, 2004. ACM Press.
3. J. Lundelius and N. Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2/3):190–204, 1984.

4. L. Meier and L. Thiele. Brief Announcement: Gradient Clock Synchronization in Sensor Networks. In *Proc. 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.
5. R. Ostrovsky and B. Patt-Shamir. Optimal and Efficient Clock Synchronization under Drifting Clocks. In *Proceedings 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 1999.
6. B. Patt-Shamir and S. Rajsbaum. A Theory of Clock Synchronization. In *Proceedings 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 810–819, 1994.
7. T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *J. ACM*, 34(3):626–645, 1987.

Brief Announcement: Abortable and Query-Abortable Objects

M.K. Aguilera[1], S. Frolund[2], V. Hadzilacos[3], S.L. Horn[3], and S. Toueg[3]

[1] HP Laboratories, Palo Alto, CA, USA
[2] Gatehouse A/S, Denmark; work done while author was at HP Laboratories
[3] Dept. of Computer Science, University of Toronto, Toronto, ON, Canada

The setting for this paper is a distributed system in which asynchronous processes interact by accessing linearizable, wait-free shared objects [2]. The typical task in such a system is to design an implementation of a target object given certain "base objects". The complexity of such implementations is mostly due to the fact that multiple processes may access the target object concurrently. It has been observed that in some systems contention on shared objects is rare [3]. It is therefore tempting to try to simplify the task of implementing shared objects by allowing them to exhibit degraded behaviour in the (hopefully rare) event of contention.

In this paper we first introduce *abortable* objects. An abortable object behaves like an ordinary one when accessed sequentially, but may return the special value $\perp$ when accessed concurrently. An operation that returns $\perp$ is said to *abort*. An aborted operation may or may not have taken effect, but the object cannot be left in an inconsistent state, reflecting the partial application of the operation. A caller that invoked an operation and received the response $\perp$ does not know whether the operation took effect. We use the term "traditional object" to signify objects where operations cannot abort.

We show that any *abortable* object can be implemented using only single-writer single-reader *abortable* registers. We also prove that abortable registers are *strictly weaker* than traditional registers. In contrast, it is well-known that many useful traditional objects cannot be implemented using traditional registers [2]. Thus, abortable objects are fundamentally easier to implement than traditional ones.

Although easier to implement, abortable objects are harder to use than traditional ones, because they provide weaker semantics. An application using an abortable object must cope not only with the prospect that an operation it applied did not take effect, but also with the potentially more serious problem of not knowing *whether* it did.

To help the user of an abortable object cope with the uncertainty regarding the fate of aborted operations, we introduce *query-abortable* objects. These are abortable objects with an additional query operation that allows each process to find out whether the last aborted operation (other than a query) that it previously applied took effect and, if so, what the appropriate response to that operation would have been. For instance, if a process p applies a dequeue operation on a query-abortable queue and the operation aborts, p can subsequently apply the query operation to find out whether its aborted dequeue took effect and, if did, what element it actually dequeued. Of course, a query operation can itself return $\perp$, if it happens to be concurrent with other operations. In this case, the user of a query-abortable object can apply the query operation repeatedly until it returns a satisfactory response.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 534–536, 2006.

We prove that any *query-abortable* object can also be implemented using only (single-writer single-reader) abortable registers. Note that in this construction the base objects are only abortable registers, not query-abortable ones.

Abortable objects and obstruction-free implementations. To avoid the complexity and expense of wait-free implementations, Herlihy, Luchangco and Moir proposed replacing wait freedom by the weaker requirement of obstruction freedom. An implementation of a traditional object is *obstruction-free* if the object returns a response to each operation that eventually executes *in isolation* [3].

It is easy to see that query-abortable objects can be used to implement obstruction-free objects. The converse is not true, because an obstruction-free object may "hang", i.e., it may never return a response to operations that encounter persistent contention.

Related work. Our definition of abortable objects was inspired by an alternative proposal by Attiya, Guerraoui and Kouznetsov [1]. We first explain their proposal and its relation to ours, and then we compare the results of the two papers.

By definition, in an obstruction-free implementation, an operation that always encounters contention may never terminate. Attiya et al. were the first to suggest that, instead of starving, it may be better for an operation facing such persistent contention to return control to the caller. In their proposal, an operation that returns to the caller does so either with the value $\emptyset$, indicating that the operation surely did not take effect, or with the value $\perp$, indicating that the operation may or may not have taken effect and is now considered "paused". A paused operation causes no contention, giving other operations a chance to terminate. The caller of a paused operation on object O can access O again only by resuming the paused operation. A paused and subsequently resumed operation may terminate and return a "normal" response; or, if it encounters contention again, it may return $\emptyset$ or $\perp$ with the meanings explained before.

An important difference between these "pausable objects" and query-abortable ones is that the caller of a paused operation can access the object again *only* by resuming that operation. In contrast, the caller of an aborted operation is not restricted in any way on how it may access the object again. If it wishes to do so, the caller can simulate the semantics of pausable objects by calling the query operation (perhaps repeatedly) to find the outcome of the aborted operation. But if applying a different operation on the object better suits its objectives, it is free to do so. Thus, query-abortable objects are more versatile than pausable ones.

Another difference between pausable objects and query-abortable ones is the status of an aborted operation that returns control to the caller. In pausable objects, the aborted operation is still pending and may take effect later, when it is resumed. In query-abortable objects, the aborted operation is not pending: it has either taken effect or not, but this will not change in the future, even if the caller of the aborted operation later applies a query operation.

Thus, the behaviour of query-abortable objects is consistent with that of traditional objects: Processes (including those whose last operation on the object aborted) are not restricted in what operation they can apply to the object, and an operation that returns control to the caller cannot remain pending. Abortable and query-abortable objects are simply objects that, in addition to wait freedom and linearizability, satisfy one other property: an operation may abort only if it encounters contention.

Attiya *et al.* prove that any pausable object can be implemented using *traditional* registers; in that implementation, the base objects have stronger properties than the implemented object in the face of contention. In contrast, we prove that any query-abortable object can be implemented using *abortable* registers; here the base and implemented objects have the same properties in the face of contention. Furthermore, starting with strictly weaker registers we show how to implement more versatile objects.

Interval vs. step contention. Obstruction-free implementations, pausable objects, and abortable objects all allow an object to exhibit weak behaviour in the presence of contention. Several different notions of contention have been defined, two of which are particularly relevant to our discussion: interval contention and step contention. Informally, an operation encounters interval contention if its execution interval intersects that of another operation; an operation encounters step contention if its execution interval contains steps of another operation. Thus, step contention implies interval contention, but the converse is not necessarily true.

Attiya *et al.* base their specification of desirable behaviour for pausable objects on step, rather than interval, contention. This would appear to be the better choice since, by doing so, one restricts the circumstances under which an object can return "unusual" responses (such as $\emptyset$ or $\perp$). We could have followed this approach in our specification of desirable behaviour for abortable objects; indeed, initially we did so. We have since come to realize, however, that this choice can lead to anomalies.

Specifically, we demonstrate an implementation of a target object O from a set of base objects B that meets the step-contention-based specification, but such that if we replace the objects in B by a *correct* implementation using "finer" base objects, the resulting implementation of O fails to meet the step-contention-based specification.

We consider this lack of composability of implementations undesirable. To avoid this problem, we define abortable objects based on interval rather than step contention: with this definition, abortable object implementations are indeed composable.

Summary of contributions. We define abortable and query-abortable objects as a simple and clean extension of standard wait-free linearizable objects. We show how to implement any abortable and query-abortable object using (single-writer single-reader) abortable registers. We prove that abortable registers are strictly weaker than traditional ones.

It was known that traditional registers can be used to implement obstruction-free objects [3]. In this paper we show something stronger: query-abortable objects (which are strictly stronger than obstruction-free objects) can be implemented using abortable registers (which are strictly weaker than traditional registers).

References

1. H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proc. of DISC '05*, pages 122–136, Krakow, Poland, 2005.
2. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
3. M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of ICDCS '03*, pages 522–529, Washington, DC, USA, 2003.

Brief Announcement: Fault-Tolerant SemiFast Implementations of Atomic Read/Write Registers

Chryssis Georgiou[1], Nicolas C. Nicolaou[2], and Alexander A. Shvartsman[2,3]

[1] Dept. of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus
[2] Dept. of Computer Science, University of Connecticut, USA
[3] Computer Science and Artificial Intelligence Laboratory, MIT, USA

Problem and Motivation. Atomic (linearizable) read/write memory is one of the fundamental abstractions in distributed computing. Atomic object services in message-passing systems allow processes to share information with precise consistency guarantees in the presence of asynchrony and failures. A seminal implementation of atomic memory of Attiya *et al.* [1] gives a single-writer, multiple reader (SWMR) solution where each data object is replicated at n message-passing nodes. Following this development, a folklore belief developed that in messaging-passing atomic memory implementations "atomic reads must write". However, recent work by Dutta *et al.* [2] established that if the number of readers is appropriately constrained with respect to the number of replicas, then single communication round implementations of reads are possible. Such an implementation given in [2] is called *fast*. Furthermore it was shown that any implementation with a larger set of readers cannot have only the single round-trip reads. Thus when the number of readers can be large, it is interesting to consider *semifast* implementations where the writes involve a single communication round and where the reads may involve one or two rounds with the goal of having as many as possible single round reads.

Our Contributions. Our goal is to develop atomic memory algorithms where a large number of read and write operations are fast. In particular, we want to remove constraints on the number of readers while preserving atomicity. We say that an atomic SWMR implementation is *semifast* if write operations take a single communication round and where read operations take one or two rounds. We show that one can obtain semifast implementations with unbounded number of readers, where in many cases reads take a single round. Our approach is based on forming groups of processes where each group is given a unique virtual identifier. The algorithm is patterned after the general scheme of the algorithm in [2]. We show that for each write operation at most one complete read operation returning the written value may need to perform a second communication round. Furthermore, our implementation enables non-trivial executions where both reads and writes are fast. We also provide simulation results for our algorithm, and we consider semifast implementations for multiple writers.

Semifast Implementations and Virtual Nodes. We consider the single writer, multiple reader (SWMR) model, where a distinguished process w is the writer, the set of R readers are processes with unique identifiers from the set $\mathcal{R} = \{r_1, \ldots, r_R\}$, and where the object replicas are maintained by the set of S servers with unique identifiers from the set $\mathcal{S} = \{s_1, \ldots, s_S\}$ such that at most t servers can crash.

To accommodate arbitrarily many readers, we introduce the notion of *virtual identifiers*. We allow multiple readers to share the same virtual identifier, thus forming groups

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 537–539, 2006.

of nodes that we call *virtual nodes*. More formally each virtual node has a unique identifier from a set $\mathcal{V} = \{\nu_1, \ldots, \nu_V\}$, and each reader r_i that is a member of a virtual node ν_j maintains its own identifier r_i and its virtual identifier $\nu(r_i) = \nu_j$; we identify such process by the pair $\langle r_i, \nu_j \rangle$. We assume that some external service is used to create virtual nodes by assigning virtual identifiers to reader processes. For a read operation, the determination of the proper return value is based on the cardinality of a set maintained by the servers, known as *seen* set, which contains virtual node identifiers and probably the writer identifier. Thus we use virtual nodes to set the boundary limits of the *seen* set even though arbitrarily many readers may use the service. To ensure the correctness of our algorithm we restrict the cardinality of the seen set to be less than $\frac{S}{t} - 1$ and hence the number of virtual nodes $|\mathcal{V}|$ to be less than $\frac{S}{t} - 2$.

A semifast atomic implementation, as suggested in [2], is an implementation that either has all reads that are fast *or* all writes that are fast. We formalize the definition of *semifast* implementations that requires all writes to be fast and that specifies which atomic reads are required to perform a second communication round. In this brief announcement we present an informal version of our definition: here, for each write operation, only *one complete* read operation is allowed to perform two communication rounds. In more detail a SWMR implementation I is *semifast* if the following properties are satisfied: (1) All writes are *fast*, (2) all complete read operations perform *one* or *two* communication rounds, (3) if a read operation ρ_1 performs two communication rounds, then all read operations that precede or succeed ρ_1 and return the same value as ρ_1 are fast, and (4) there exists some execution of I which contains only fast read and write operations.

Implementation SF. We now overview a semifast implementation, called SF, that supports one writer and arbitrarily many readers. We use timestamps to impose a partial order on the read and write operations, as in [1]. Of interest is the way in which timestamps are associated with the values.

To perform a write operation, the writer increases the timestamp and sends the new value to $S - t$ servers. The timestamps impose a natural order on the writes since there is only one writer.

The server processes maintain object replicas and do not invoke any read or write operations. To implement the fast operation behavior, the servers use a bookkeeping mechanism to record all the processes to whom they sent their latest timestamp. Therefore when a server s_i receives a message $\langle msgType, ts, *, vid \rangle$ from a non-server process p_j, it updates its local timestamp ts_ℓ to be equal to ts, if $ts > ts_\ell$, and it initializes the recording set called $seen$, to $\{vid\}$. Otherwise, if $ts \leq ts_\ell$, s_i sets its *seen* set to be equal to $seen \cup \{vid\}$ declaring that p_j inquired s_i's local timestamp. When a reader performs a second communication round, then the server s_i updates its postit value ps if $ts \geq ps$. This declares that the timestamp ts is about to be returned by some reader.

When a reader process invokes a read operation it sends messages to all servers and waits for $S - t$ responses. It determines the maximum timestamp $maxTS = ts'$ and the maximum postit $maxPS = ps'$ value contained among the received messages, and it computes the set of the messages that contain the discovered $maxTS$ ($maxTSmsg$). Then the following key predicate is used to decide on the return value:

$$\exists \alpha \in [1, V+1]\ \exists \mathit{MS} \subseteq maxTSmsg\ s.t.\ |\mathit{MS}| \geq S - \alpha t \wedge |\cap_{m \in \mathit{MS}} m.seen| \geq \alpha.$$

The above predicate is derived from the observation that for any two read operations ρ_1 and ρ_2 that witness the same $maxTS$ and compute $maxTSmsg_1$ and $maxTSmsg_2$ respectively, the difference $||maxTSmsg_1| - |maxTSmsg_2||$ is less than or equal to t. If the predicate is true or if $maxPS = maxTS$ the reader returns $maxTS$ otherwise it returns $maxTS - 1$. If a reader observes that $| \cap_{m \in MS} m.seen| = \alpha$ or less than $2t + 1$ messages containing $maxPS$ are discovered in the system, then it performs a second communication round before returning $maxTS$.

To associate the timestamps with the values we maintain a triple $\langle ts, v_{ts}, v_{ts-1} \rangle$, where ts is the current timestamp, v_{ts} the value written with this timestamp, and v_{ts-1} the value written with the previous timestamp. We prove the correctness (atomicity) of the new implementation. Note that SF is not a straightforward extension of [2]. The introduction of virtual nodes raises new challenges such as ensuring consistency within groups so that atomicity is not violated by processes sharing the same virtual identifier.

Impossibility and MWMR model. We consider two families of algorithms, one that does not use reader grouping mechanisms and the other that assumes grouping mechanisms (such as our algorithm [3]). For both families we show that there is no semifast atomic implementation when $\frac{S}{t} - 2$ or more virtual nodes exist in the system. The idea behind the proof is to show by contradiction that if $V \geq \frac{S}{t} - 2$, then the fast behavior of the system violates atomicity, and as a result property (4) of the semifast definition cannot hold. Additionally it is shown that any semifast algorithm must inform no less than $3t + 1$ server processes during a second communication round — otherwise either property (3) of the semifast definition does not hold, or atomicity is violated.

Examining the applicability of the result in the multiple writer multiple reader (MWMR) model, we show that there does not exist semifast atomic implementations even in the case of 2 writers, 2 readers and $t = 1$. Our proof assumes that the read operations are allowed to perform one or more communication rounds.

Simulations. We simulated our SWMR implementation using the NS2 network simulator and we obtained preliminary results demonstrating that only a small fraction of read operations need to perform a second communication round. Specifically, under reasonable execution conditions in our simulations no more than 10% of the read operations required a second round. Our simulations assume that the readers are uniformly distributed among the virtual nodes. Furthermore we assume both stochastic and static environments and several plausible frequencies on the invocation of read and write operations. (The details can be found in [3].)

Acknowledgment. We thank Rachid Guerraoui and Peter Musial for insightful discussions. This work is supported in part by the NSF Grants 0121277 and 0311368.

References

1. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. of the ACM*, 42(1):124–142, 1996.
2. P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proc. of ACM PODC 2004*, pages 236–245.
3. C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. Fault-tolerant semifast implementation of atomic read/write registers. In *Proc. of ACM SPAA 2006*, to appear.

Brief Announcement: Convergence Analysis of Scalable Gossip Protocols

Stacy Patterson[1], Bassam Bamieh[2], and Amr El Abbadi[1]

[1] Department of Computer Science, University of California, Santa Barbara
{sep, amr}@cs.ucsb.edu
[2] Department of Mechanical Engineering, University of California, Santa Barbara
bamieh@engineering.ucsb.edu

Introduction. We present a simple, deterministic gossip protocol for solving the distributed averaging problem. Each node has an initial value and the objective is for all nodes to reach consensus on the average of these values using only communication between neighbors in the network. We first give an analysis of the protocol in structured networks, namely d-dimensional discrete tori and lattices, and show that in an n node network, the number of rounds required for the protocol to converge to within ϵ of the average is $O(|\log(\epsilon)|\ n^{2/d})$. We then extend our results to derive upper and lower bounds on convergence for arbitrary graphs based on the dimensions of spanning supergraphs and subgraphs.

Analyzing Convergence. The network is represented by an undirected graph $G = (V, E)$ where V is the set of nodes in the network, with $|V| = n$, and E is the set of communication channels between them. The neighbor set of a node i, denoted N_i, is the set of nodes $j \in V$ such that $(i, j) \in E$. Every node i has an initial value $x_i(0)$, and the average of all values in the system is $x_{ave} = \frac{1}{n}\sum_{i=1}^{n} x_i(0)$. In each round k, every node sends an equal fraction β of its current value to each of its neighbors and sends the remaining fraction, $\alpha = 1 - |N_i|\beta$, to itself. Each node updates its current value $x_i(k+1)$ to be the sum of all values received in round k. The desired goal of the protocol is for the system to converge to an equilibrium where $x_i(k) = x_{ave}$ for all $i \in V$.

We measure how far the current state of the system is from the average state using the "deviation from average" vector, with each component defined by

$$\tilde{x}_i(k) \ := \ x_i(k) \ - \ \frac{1}{n}\left(x_1(k) \ + \ \ldots \ + \ x_n(k)\right).$$

When the vector $\tilde{x}$ equals $\mathbf{0}$, the vector of all 0's, the vector x equals $x_{ave}\mathbf{1}$, where $\mathbf{1}$ is the vector of all 1's. The rate at which $||\tilde{x}||$ approaches 0 determines the rate at which the nodes reach consensus at x_{ave}. We define the *ϵ-consensus time* to be the number of rounds required for $\frac{||\tilde{x}(k)||}{||\tilde{x}(0)||} \leq \epsilon$ for a given ϵ.

If $x(k)$ denotes the vector of current values in round k, the distributed averaging protocol can be represented as an $n \times n$ matrix $A = [a_{ij}]$ that transforms $x(k)$ to $x(k+1)$, where $x(k+1)$ is the vector of values in round $k+1$. a_{ij} is the fraction of j's current value that j sends to node i in each round. The evolution of the vector x is given by following recursion equation.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 540–542, 2006.

$$x(k+1) \;=\; A\, x(k)$$

It can be shown that the evolution of the vector x depends on the second largest eigenvalue of A, λ_2. More specifically, an upper bound on the number of rounds required for ϵ-consensus is given by

$$k \geq \frac{\log(\epsilon)}{\log(|\lambda_2|)}. \tag{1}$$

Therefore, the ϵ-consensus time depends upon the inverse of $|\lambda_2|$. In general, it is not possible to determine λ_2 analytically. However, for certain graph structures, we can derive λ_2 and thus derive an asymptotic bound for the protocol.

In particular, we consider the protocol matrix for a d-dimensional discrete torus $\mathbb{Z}_s^d$. In each round, every node sends an equal fraction $\beta < \frac{1}{2d}$ of its current value to each of its $2d$ neighbors. In the 1-dimensional case, the protocol matrix A is a circulant matrix. In the general d-dimensional case, A is known as a circulant operator. The eigenvalues of A can be explicitly obtained using the multi-dimensional Discrete Fourier Transform [1]. Using this and Equation (1), we can determine the ϵ-consensus time of the protocol in any d-dimensional discrete torus. For large n, the asymptotic convergence of the protocol in a d-lattice is the same as for a d-dimensional torus, so the same analysis applies.

Theorem 1. *The ϵ-consensus time of the distributed averaging protocol in a discrete d-dimensional torus or d-lattice with n nodes is $O(|\log(\epsilon)|\; n^{2/d})$.*

This result shows that the dimensionality of the torus determines the convergence rate of the averaging protocol. In tori, the dimension is closely related to the connectivity of the graph; a higher dimensional torus has greater connectivity and a faster convergence rate than a lower dimensional torus. If β is chosen carefully, a similar relationship between connectivity and convergence can be derived for arbitrary graphs.

Theorem 2. *Let $G_1 = (V_1, E_1)$ be an undirected, connected graph. Let $G_2 = (V_2, E_2)$ be a spanning subgraph of G_1. For $\beta \leq \frac{1}{2\Delta(G_1)}$, where $\Delta(G_1)$ is the maximum degree of G_1, the convergence rate of the protocol on G_1 is greater than or equal to that on G_2.*

According to the Theorem 2, if we start with a d-dimensional torus and add edges, the convergence rate of the protocol on the new graph will be at least as fast as that on the original graph. Similarly, if we take an arbitrary graph and add edges to form a d-dimensional torus, the convergence rate of the protocol on the original graph will be less than or equal to that of the protocol on the torus. This result is stated precisely in the following corollary.

Corollary 1. *Let $G = (V, E)$ be an arbitrary graph with $|V| = n$.*

1. *If D is the dimension of the largest dimensional torus that is a spanning subgraph of G, the consensus protocol reaches ϵ-consensus on G in $O\left(|\log(\epsilon)|\; n^{2/D}\right)$ rounds.*

2. If d is the dimension of the smallest dimensional torus for which G is a spanning subgraph, the consensus protocol reaches ϵ-consensus on G in $\Omega\left(|\log(\epsilon)|\ n^{2/d}\right)$ rounds.

This corollary is a direct result of Theorems 1 and 2.

Details on the above results can be found in [2].

Related Work. Stochastic gossip protocols for solving the distributed averaging problem have been proposed [3,4]. In these protocols, each node selects a neighbor at random in each round and averages its own value with the neighbor's value. Bounds derived for convergence rates of these protocols are probabilistic. A deterministic protocol in which a node can send a different fraction of its current value to each of its neighbors in each round have also been studied [5]. The authors use offline analysis based on global information to determine the optimal fraction to send along each edge and give localized online heuristics. However, the selection of fractions does not affect the asymptotic convergence rate of the protocol. Distributed averaging has also been studied in the context of anonymous networks [6]. In this work, each node must know the size and topology of the network. The role of dimensionality in optimal error bounds in sensor networks was studied in [7]. Our work characterizes the relationship between network dimensionality and the convergence rate of a deterministic gossip protocol for distributed averaging that requires only local information, making it well-suited for large scale P2P systems, sensor networks, and mobile ad-hoc networks.

References

1. Salapaka, S., Peirce, A., Dahleh, M.: Analysis of a circulant based preconditioner for a class of lower rank extracted systems. In: Numerical Linear Algebra with Applications. Volume 12. (2005) 9–32
2. Patterson, S., Bamieh, B., El Abbadi, A.: Convergence analysis of scalable gossip protocols. Technical Report 2006-09, Department of Computer Science, University of California, Santa Barbara (2006)
3. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: IEEE Symposium on Foundations of Computer Science. (2003) 482
4. Boyd, S., Ghosh, A., Prabhakar, B., Shah, D.: Gossip algorithms: Design, analysis and applications. In: INFOCOM. Volume 3. (2005) 1653–1664
5. Xiao, L., Boyd, S.: Fast linear iterations for distributed averaging. In: Systems and Control Letters. Volume 53. (2005) 65–78
6. Kranakis, E., Krizanc, D., van den Berg, J.: Computing boolean functions on anonymous networks. Information and Computation **114** (1994) 214–236
7. Barooah, P., Hespanha, J.: Graph effective resistances and distributed control: Part ii – electrical analogy and scalability. In: 45th IEEE Conference on Decision and Control. Feb. 2006 (submitted). (2006)

Brief Announcement: Computing Automatically the Stabilization Time Against the Worst and the Best Schedules

Joffroy Beauquier, Colette Johnen, and Stéphane Messika

L.R.I./C.N.R.S., Université Paris-Sud 11

Abstract. In this paper, we reduce the problem of computing the convergence time for a randomized self-stabilizing algorithm to an instance of the stochastic shortest path problem (SSP). The solution gives us a way to compute automatically the stabilization time against the worst and the best policy. Moreover, a corollary of this reduction ensures that the best and the worst policy for this kind of algorithms are memoryless and deterministic. We apply these results here in a toy example. We just present here the main results, to more details, see [1].

By their very nature, distributed algorithms have to deal with a non-deterministic environment. Speeds of the different processors or the message delays are generally not known in advance and may vary substantially from one execution to the other. For representing the environment in an abstract way, the notion of scheduler (also called demon or adversary) has been introduced. The scheduler is in particular responsible of which processors take a step in a given configuration or of which among the messages in transit arrives first. It is well known that the correctness of a distributed algorithm depends on the considered scheduler. This remark also holds for self-stabilizing distributed algorithm.

In this paper, we restrict our attention to probabilistic self-stabilization. Classically self-stabilization requires convergence (each execution reaches a legitimate configuration) and correctness (each execution starting from a legitimate configuration satisfies the specification). Probabilistic self-stabilization requires that convergence is probabilistic. It appears that the convergence property of a given algorithm depends on the chosen policy. With some policies the algorithm can converge in a finite number of steps (the stabilization time) while with others it can not converge at all. Even if the stabilization time is finite, it can differ according to the policy. It is thus interesting to know the best policy (the policy that gives the smaller expected stabilization time) and the worst. Note that the best policy can possibly give a finite stabilization time and the worst an infinite one. In some cases best and worst both give finite stabilization time (it is then said that the algorithm is self-stabilizing under the distributed scheduler: scheduler that produce any k-bounded policy).

In a distributed system, all the machines are finite state machines. A configuration X of the distributed system is the N-tuple of all the states of the machines. The code is a finite set of guarded rules. The guard of a rule on p is a boolean expression involving the state of p and its neighbors. A machine p

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 543–547, 2006.

is *enabled* in a configuration c, if a rule guard of p is true, in c. The execution simultaneously by several enabled machines of rules is call a *computation step*.

A randomized distributed algorithm can be seen as a Markov Decision Process. Informally, a Markov Decision Process is a generalization of a Markov chain in which a set of possible actions is associated to each state. To each state-action pair corresponds a probability distribution on the states, which is used to select the successor state. A Markov chain corresponds thus to a Markov Decision Process in which there is exactly one action associated with each state. The formal definition is as follows.

Definition 1 (Markov Decision Process). *A Markov Decision Process (MDP)* $(S,\ Act,\ A,\ p)$ *consists of a finite set* S *of states, a finite set* Act *of actions, and two components* A, p *that specify the transition structure.*

- *For each* $s \in S$, $A(s)$ *is the non-empty finite set of actions available at* s.
- *For each* $s, t \in S$ *and* $a \in A(s)$, $p_{st}(a)$ *is the probability of a transition from* s *to* t *when action* a *is selected. Moreover,* p *verifies the following property* $\forall s, \forall a \in A(s)$ *we have* $\sum_{t \in S} p_{st}(a) = 1$.

The MDP associated with a distributed algorithm is defined by (i) S is the set of configurations, (ii) *Act* is the set of machine sets, (iii) $A(c)$ is the set of enabled machines in c, (iv) $p_{st}(a)$ is the probability to reach the configuration t from a configuration s by a computation step where all processors in a execute a rule.

Policies are closely related to the adversaries of Segala and Lynch, and to the schedulers of Lehman and Rabin, Vardi, Pnueli and Zuck , and to the strategies of Beauquier, Delaët, Granidariu, and Johnen.

Definition 2 (Policy). *A policy* η *is a set of conditional probabilities* $Q_\eta(a|s_0 s_1 ... s_n)$, *for all* $n \geq 0$, *all possible sequences of states* $s_0, ..., s_n$ *and all* $a \in A(s_n)$ *such that* $0 \leq Q_\eta(a|s_0, s_1 ..., s_n) \leq 1$ *and* $\sum_{a \in A(s_n)} Q_\eta(a|s_0, s_1 ..., s_n) = 1$.

Definition 3 (Probability measure under a policy). *Let* η *be a policy. Let* $h = s_0 a_0 s_1 a_1 ... s_n$ *be a sequence of computation steps.* $P_s^\eta(C_h) = \prod_{k=0}^{n-1} p_{s_k s_{k+1}}(a_k) Q_\eta(a_k|s_0, s_1 ..., s_k)$.

Under a given policy, the randomized distributed algorithm can be seen as a Markov chain. In this Markov chain, we denoted by X_n the configuration reached after n computation steps (X_n is the random variable).

Definition 4 (Probabilistic convergence). *Let Leg be the legitimate predicate defined on configurations. A probabilistic distributed algorithm A under a policy* η *probabilistically converges to Leg iff : from any configuration* s, *the probability to reach a legitimate configuration is equal to 1 under the policy* η. *Formally,* $\lim_{n \to \infty} P_c^\eta(\exists m \leq n \mid X_m \in L) = 1$ *where* X_m *is the reached state after* m *computation steps in the Markov chain defined by : MDP associated to* A, c *and* η.

The convergence time under a policy is the expectation value of the random variable Y under this policy (Y being the number of computation steps to reach a legitimate configuration). The best policy (resp. the worst policy) is the policy having the smallest (resp. largest) convergence time.

Informally, the Stochastic Shortest Path problem consists in computing the minimum expected cost for reaching a given subset of destination states, from any state of a Markov Decision Process in which a cost is associated to each action.

Definition 5 (Instance of Stochastic Shortest Path Problem). *An instance of the stochastic shortest path problem is (M, U, c, g) in which (i) M is a Markov Decision Process, (ii) U is the set of destination states, (iii) c is the cost function, which associates to each state $s \in S - U$ and action $a \in A(s)$ the cost $c(s, a)$, and (iv) g is the terminal cost function which associates to each $s \in U$ its terminal cost $g(s)$.*

In [1], we have define two instances of SSP (called SSP_b and SSP_w). SSP_b allows us to compute the best convergence time. SSP_w is designed to compute the worst convergence time.

- SSP_b: M is the MDP associated to the algorithm. U is the set of legitimate configurations. c is always equal to 1. g is function null.
- SSP_w: M is the MDP associated to the algorithm. U is the set of legitimate configurations. c is always equal to -1. g is function null.

Definition 6 (Bellman operator). *Let (M, U, c, g) be an instance of the SSP problem. We denote $v = (v_s)_{s \in S-U}$ a vector of real numbers. We define the Bellman operator L by*

$$L(v_s) = min_{a \in A(s)}\{c(s, a) + \sum_{t \in S-U} p_{st}(a)v_t + \sum_{t \in U} p_{st}(a)g(t)\}$$

Theorem 1. *Computing the convergence time for a randomized self-stabilizing algorithm under the best and worst policy can be reduced to an instance of SSP.*

Thus, one can apply the result on the SSP problem. For instance, the Bellman operator has a fixpoint in SSP_b and in SSP_w. Thus, the convergence time under the best and worst policy can be compute automatically (via the fixpoint of Bellman operator). We can also obtain the corresponding policies. In SSP_b, (resp. SSP_w) on each configuration, the selected action is the best choice (resp. worst choice) to converge from this configuration.

Toy Example. The presented algorithm achieves token circulation on unidirectional ring, it was defined by Beauquier and al. in [BDC95].

A processor is said to have a token iff it is enabled. The algorithm to be self-stabilizing should converge from a configuration with several tokens to a configuration with one token. Notice that the algorithm ensures that in any configuration, the ring has at least one token.

Configurations are gathered in class. Configurations in which the tokens are at the same (clockwise) distance d, belong to the same class denoted d. If a

Algorithm 1. token circulation on anonymous and unidirectional rings

```
Constant:
$N$ is the ring size. $m_N$ is the smallest integer not dividing $N$.
Variables on $p$:    $v_p$ is a variable taking value in [0, $m_N$ -1].
Random Variables on $p$:
    $rand\_bool_p$ taking value in {1, 0}. Each value has a probability 1/2.
Action on $p$: $lp$ is the clockwise preceeding neighbor of $p$
    $\mathcal{R}$:: $(v_p - v_{lp} \neq 1 \bmod m_N) \rightarrow$ if $rand\_bool_p = 1$ then $v_p := (v_{lp} + 1) \bmod m_N$;
.
```

token $T1$ is at distance d of the other token $T2$ then $T2$ is at distance N-d of $T1$. Therefore class d and N-d are identical. Notice that if $d = 0$ then the ring has only one token. The best convergence time is easy to guess here, from the configuration of the class $0 < d \leq N/2$ is $2 \times d$, whatever is the ring size. This convergence time is achieved under the following policy: in any configuration, the token at distance d of the other token tries to catch up the unmoving token.

Let us take $N = 5$, then there are 3 classes c_0, c_1, c_2 There are also three kinds of policy in each different configuration, one can choose only the closest token (action c), the furthest (f), or both of them (b). In the corresponding SSP instance we have $U = c_0$ and we can start from the vector $v^0 = (0, 0)$ then applying one time the Bellman operator L we obtain that $v^1 = (1, 1)$ then $v^2 = (3/2, 2)$, $v^3 = (7/4, 11/4)$, $v^4 = (15/8, 26/8)$, $v^5 = (31/16, 57/16)$, $v^6 = (63/32, 120/32)$, $v^7 = (127/64, 247/64)$, $v^8 = (255/128, 502/128)$ and the sequence converges towards $v^\bullet = (2, 4)$ which is effectively a fixpoint. Then, by getting the action in which the minimum is reached we obtained that the best policy is the one that chose always action c, that is conform to the intuition.

In [1], we study two other self-stabilizing algorithms : vertex coloring, and naming in grids. There is a self-stabilizing deterministic algorithm giving distinct name to nodes in grids. Using this technique we show that the convergence time under the worst and best policy are better with our randomized algorithm that with the deterministic one.

One could think that the best policy and the worst policy are intricate and difficult to describe, or that their simulation would use a lot of resources. In fact it is not true because there are always a best policy and a worst policy that are memoryless (meaning that the choice they make in a given configuration depends only on the configuration, and not on the past of the execution). This results extend a result in [3] and, although in a different context, a result of [2].

Corollary 1. *The best and the worst policy (considering the convergence time) for a randomized self-stabilizing algorithm are memoryless and deterministic.*

This last result allows us to only consider memoryless policies when studying self-stabilizing algorithms. Indeed, the worst convergence time is always given by a memoryless policy. This result considerably simplifies the verification of the correctness and the computation of the complexity of these algorithms.

References

1. J. Beauquier, C. Johnen, and S. Messika. Computing automatically the stabilization time against the worst and the best schedulers. Technical Report 1448, L.R.I, 2006.
2. D.P. Bertsekas and J.N. Tsitsiklis. An analysis of stochastic shortest path problems. *Math of Op. Res.*, 16(2):580–595, 1991.
3. L. de Alfaro. *Formal Verification of Probabilistic systems.* PhD Thesis, Stanford University, 1997.

Brief Announcement: Many Slices Are Better Than One

Vinit A. Ogale and Vijay K. Garg*

PDS Laboratory, Dept. of Electrical and Computer Engineering, University of Texas at Austin
{ogale, garg}@ece.utexas.edu

1 Introduction and Background

In this paper, we present a new technique called *multislicing* to efficiently verify whether a distributed program has executed correctly. Our algorithm supports a class of temporal predicates (Multislicing Temporal Logic or MTL [1]) which allows properties based on local predicates and arbitrarily placed negations, disjunctions and conjunctions along with the *possibly* ($\Diamond$, EF) temporal operator. We show that multislicing makes it possible to detect any MTL predicate in polynomial time with respect to the number of processes (n) in the system and the number of events ($|E|$) in the distributed computation (though, as expected, it is not polynomial with respect to the size of the predicate) [1]. We do not know of any other algorithm that allows detection of a similar class of predicates in polynomial time in n (or $|E|$).

We model a distributed computation ($\langle E, \rightarrow \rangle$) as a partial order on the set of events E, based on the happened before relation ($\rightarrow$) [2]. A partial order captures all the possible causally consistent interleavings and ensures that bugs in any possible consistent interleaving of the computation will be visible. The drawback of using a partial order model is that the number of global states of the computation is exponential in the number of processes in the system.

A *consistent cut* C is a set of events in the computation which satisfies the following property: if an event e is contained in the set C, then all events in the computation that happened before e are contained in C, i.e., $\forall e_1, e_2 \in E : (e_2 \in C) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in C$. Detecting a predicate in a distributed computation is determining if the initial cut of the computation satisfies the predicate. Using a model checking approach for predicate detection is inefficient and requires exponential time in the worst case.

The set of all consistent cuts in a computation forms a distributive lattice [3]. Birkhoff's representation theorem [4] states that a distributive lattice can be completely characterized by the set of its join irreducible elements. Join irreducible elements are elements of the lattice that cannot be expressed as the join of any two elements.

The *slice* $s[p]$ of a computation with respect to a predicate p is the poset of the join irreducible consistent cuts of the smallest sublattice that contains all consistent cuts satisfying p. If every consistent cut in the slice $s[p]$ satisfies the predicate p, then the slice is said to be a *lean slice*. A lean slice is obtained whenever the consistent cuts that satisfy the predicate have a lattice structure (closed under joins and meets). Such predicates are called *regular predicates*.

* Supported in part by the NSF Grants ECS-9907213, CCR-9988225, CNS-0509024, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 548–550, 2006.

Predicate detection using computation slicing traditionally involves computing the slice with respect to the predicate and examining the slice [5]. The drawback of this approach is that, if the computed slice is not lean, then we are forced to examine each global state in the slice. This may require exponential time with respect to the number of processes in the system.

In *multislicing*, instead of computing a single non-lean slice, we compute a set of slices that are guaranteed to be lean. We show that it is possible to restrict the size of this set, thus making it possible to efficiently detect the predicate.

2 Multislicing Overview

Definition 1. *(Multislice) A multislice S of a computation $\mathcal{C}$ with respect to a predicate P is defined as a set of lean slices $\{s_1, s_2, ..., s_l\}$ corresponding to regular predicates $p_1, p_2, ..., p_l$ such that $P = p_1 \vee p_2 \vee ... \vee p_l$.*

Note that, the multislice of a computation for a predicate is not necessarily unique [1]. *Multislicing* is the operation of computing a multislice of the computation with respect to the given predicate. It follows from the definition that given a predicate P, a multislice $S[P]$ satisfies the following properties:

1. If P is true at consistent cut C in the computation, then there exists a slice s in the multislice $S[P]$, such that, cut C belongs to s
2. $\forall s \in S[P]$: C belongs to $s \Rightarrow C$ satisfies the predicate P

A multislice represents all the cuts in the computation at which the predicate holds. To determine if the given MTL predicate holds at the initial state of the computation, we can simply check if the initial cut belongs to any slice of the multislice. A multislice is nonempty if the predicate is true in any consistent cut of the computation.

Definition 2. *A predicate P in MTL is defined recursively as follows:*

1. *All local predicates are MTL predicates*
2. *If P and Q are MTL predicates then $P \vee Q$, $P \wedge Q$, $\Diamond P$ and $\neg P$ are MTL predicates*

Note that $\vee$, $\wedge$, $\neg$, $\Diamond$ have their usual meanings [1]. $\Box P$ (or $AG(P)$) can rewritten in MTL as $\neg\Diamond\neg P$.

In the following discussion, we denote regular predicates and lean slices by small case letters (e.g. s or $s[p]$ denotes the slice of the computation for a regular predicate p). We use uppercase letters to represent multislices and MTL predicates.

Efficient algorithms to compute the slice for local predicates can be found in [3]. The multislice of a local predicate is the single element set containing its slice. It is trivial to compute the multislice for the disjunctions of two predicates($S[P \vee Q] \equiv S[P] \cup S[Q]$). The conjunction of two lean slices, $s_1 \wedge s_2$, can be computed by using the grafting algorithm described in [3]. The multislice for $S[P \wedge Q]$ is obtained by computing $s_p \wedge s_q$ for all $s_p \in S[P], s_q \in S[Q]$. Since $\Diamond$ distributes over $\vee$, the multislice $S[\Diamond P]$ is obtained by computing the slices $\Diamond s$ for all $s \in S[P]$. Note that if s_1 and s_2 are lean slices, then $s_1 \wedge s_2$ and $\Diamond s_1$ are also lean [3,5].

We assume that the negation operators are pushed inside as far as possible (to the local predicates or to the $\Diamond$ operator). Using existing slicing approaches to calculate a multislice for a predicate with nested negations can quickly lead to an exponential blowup. To avoid this, we present a new algorithm to efficiently compute $\neg\Diamond p$, where p is a regular predicate. To compute the multislice of $\neg\Diamond P$ where $P = p_1 \vee p_2 \vee \ldots \vee p_l$ is a MTL predicate, we compute the multislices for $\neg\Diamond p_1$, $\neg\Diamond p_2$ etc. and take their conjunction. The detailed algorithm to efficiently compute the multislice for $\neg\Diamond$ of a regular predicate is presented in the technical report [1]. The main idea is, to find along each process, the least cut that does not satisfy $\Diamond P$. We get a set of at most n such cuts (called the *least cuts*) when the computation has n processes. Let $S' = \{C_1, C_2, \ldots, C_n\}$ be the set of least cuts. The output multislice S is given by the set of n slices $S = \{\texttt{intervalLatticeBetween}(C_i, E) | i = 1 \text{ to } n\}$ where E is the final cut. Note that, $\texttt{intervalLatticeBetween}(C_1, C_2)$ returns the slice which with the contains all cuts between C_1 and C_2 with C_1, C_2 as the initial and final cuts respectively.

Though it may appear that nested $\neg\Diamond$ operators could lead to an exponential blowup (in n, the number of processes), we show that the number of slices does not explode due to the property that the multislice for $\neg\Diamond P$ is a join-closed structure in the original computational lattice [1].

Theorem 1. *The time complexity of our multislicing algorithm is polynomial in the number of events ($|E|$) and the number of processes (n) in the computation.*

In the future, it will be interesting to extend multislicing for non-terminating computations. Another promising avenue is to explore ways to define and efficiently compute a canonical version of a multislice.

A detailed explanation of the algorithms, proofs of correctness, complexity analysis and the experimental evaluation can be found in the technical report [1].

References

1. Ogale, V.A., Garg, V.K.: Multislicing a computation. Technical Report TR-PDS-2006-002, University of Texas at Austin (2006) Available as `http://maple.ece.utexas.edu/TechReports/2006/TR-PDS-2006-02.ps`.
2. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565
3. Mittal, N., Garg, V.K.: Slicing a distributed computation: Techniques and theory. In: 5th International Symposium on DIStributed Computing (DISC'01). (2001) 78 – 92
4. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge, UK (1990)
5. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: 7th International Conference on Principles of Distributed Systems. (2003)

Brief Announcement: On Augmented Graph Navigability*

Pierre Fraigniaud, Emmanuelle Lebhar, and Zvi Lotker

[1] CNRS and U. Paris-Sud
[2] ENS Lyon
[3] CWI, Amsterdam

Abstract. It is known that graphs of doubling dimension $O(\log\log n)$ can be augmented to become navigable. We show that for doubling dimension $\gg \log\log n$, an infinite family of graphs cannot be augmented to become navigable. Our proof uses a counting argument which enable us to consider any kind of augmentations. In particular we do not restrict our analysis to the case of symmetric distributions, nor to distributions for which the choice of the long range link at a node must be independent from the choices of long range links at other nodes.

1 Statement of the Problem

The *doubling dimension* of a graph G is the smallest d such that, for any integer $r \geq 1$, and for any node $u \in V(G)$, the ball $B(u, 2r)$ centered at u and of radius $2r$ can be covered by at most 2^d balls $B(u_i, r)$ centered at nodes $u_i \in V(G)$. The doubling dimension has an impact on the analysis of the small world phenomenon, precisely on the expected performances of greedy routing in *augmented* graphs. An augmented graph is a pair (G, φ) where G is an n-node graph, and φ is a collection of probability distributions $\{\varphi_u, u \in V(G)\}$. Every node $u \in V(G)$ is given an extra link pointing to some node v, called the *long range contact* of u. The link from a node to its long range contact is called a *long range link*. The original links of the graph are called *local links*. The long range contact of u is chosen at random according to φ_u as follows: $\Pr\{u \to v\} = \varphi_u(v)$. Greedy routing in (G, φ) is the oblivious routing protocol where the routing decision taken at the current node u for a message of destination t consists in (1) selecting a neighbor v of u that is the closest to t according to the distance in G (this choice is performed among all neighbors of u in G and the long range contact of u), and (2) forwarding the message to v. This process assumes that every node has a knowledge of the distances in G. On the other hand, every node is unaware of the long range links added to G, except its own long range link. Hence the nodes have no notion of the distances in the augmented graph.

* The complete version of this paper is available as: P. Fraigniaud, E. Lebhar, and Z. Lotker. A Doubling Dimension Threshold $\Theta(\log\log n)$ for Augmented Graph Navigability. In 14th European Symp. on Algorithms (ESA), ETH Zürich, Switzerland, September 11-15, 2006.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 551–553, 2006.

An infinite family of graphs $\mathcal{G} = \{G^{(i)}, i \in I\}$ is *navigable* if there exists a family $\Phi = \{\varphi^{(i)}, i \in I\}$ of collections of probability distributions, and a function $f(n) \in O(\text{polylog}(n))$ such that, for any $i \in I$, greedy routing in $(G^{(i)}, \varphi^{(i)})$ performs in at most $f(n^{(i)})$ expected number of steps where $n^{(i)}$ is the order of the graph $G^{(i)}$. More precisely, for any pair of nodes (s,t) of $G^{(i)}$, the expected number of steps $\mathbb{E}(\varphi^{(i)}, s, t)$ for traveling from s to t using greedy routing in $(G^{(i)}, \varphi^{(i)})$ is at most $f(n^{(i)})$. In his seminal paper, Kleinberg (STOC, 2000) proved that, for any fixed integer $d \geq 1$, the family of d-dimensional square meshes is navigable. Slivkins (PODC, 2005) recently related navigability to doubling dimension by proving that any metric with doubling dimension at most $O(\log\log n)$ is navigable. All these results naturally lead to the question of whether all graphs are navigable.

2 Our Results

Let $\delta : \mathbf{N} \mapsto \mathbf{N}$, let $\mathcal{G}_{n,\delta(n)}$ be the class of n-node graphs with doubling dimension at most $\delta(n)$, and let $\mathcal{G}_\delta = \cup_{n\geq 1}\mathcal{G}_{n,\delta(n)}$. Slivkins result restricted to the context of non-weighted graphs implies that $\mathcal{G}_\delta$ is navigable for any function δ bounded from above by $c\log\log n$ for some constant $c > 0$. We prove a threshold of $\delta(n) = \Theta(\log\log n)$ for the navigability of $\mathcal{G}_\delta$:

Theorem 1. *Let* $\delta : \mathbf{N} \mapsto \mathbf{N}$ *be such that* $\lim_{n\to\infty} \frac{\log\log n}{\delta(n)} = 0$. *Then* $\mathcal{G}_\delta$ *is not navigable.*

Hence, the result by Slivkins is essentially the best that can be achieved by considering only the doubling dimension of graphs. Note that Theorem 1 does not assume independent trials for the long range links.

3 Sketch of the Proof

Our negative result requires to prove that for an infinite family of graphs in $\mathcal{G}_\delta$, *any* distribution of the long range links leaves the expected number of steps of greedy routing above any polylogarithmic for some pairs of source and target. For this purpose, we exhibit an infinite family of graphs presenting a very high number of possible "directions" for a long range link to go, implying that for any trial of the long range links, there always exist a direction for which these long links do not help. Precisely, by a counting argument, we show that there exists a pair of source and target at distance greater than any polylogarithm, between which greedy routing does not use any long range link, whatever their distribution is.

Let $\{G^{(n)}, n \geq 1\}$ be an infinite family of graphs indexed by their number of vertices as depicted on Figure 1. Precisely, let $d : \mathbf{N} \mapsto \mathbf{N}$ be such that $d \leq \delta$, $\lim_{n\to\infty} \frac{\log\log n}{d(n)} = 0$, and $d(n) \leq \varepsilon\sqrt{\log n}$ for some $0 < \varepsilon < 1$. For the sake of simplicity, assume that $p = n^{1/d(n)}$ is integer. $G^{(n)}$ is the graph of n nodes consisting of $p^{d(n)}$ nodes labeled $(x_1, \ldots, x_{d(n)})$, $x_i \in \mathbf{Z}_p$. Node $(x_1, \ldots, x_{d(n)})$

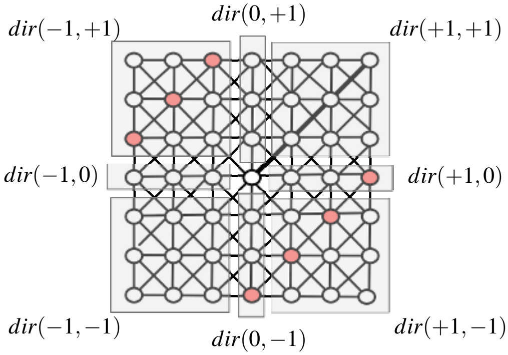


Fig. 1. Example of graph $G^{(n)}$ with $d(n) = 2$

is connected to all nodes $(x_1 + a_1, \ldots, x_{d(n)} + a_{d(n)})$ where $a_i \in \{-1, 0, 1\}$, $i = 1, \ldots, d(n)$, and all operations are taken modulo p. One can check that the diameter of $G^{(n)}$ is $\lfloor p/2 \rfloor$, and $G^{(n)} \in \mathcal{G}_{n,\delta(n)}$.

Let $u = (u_1, \ldots, u_{d(n)})$ be any node. For $D = (\nu_1, \ldots, \nu_{d(n)}) \in \{-1, 0, +1\}^{d(n)}$, we call *direction* the set of nodes $\mathrm{dir}_u(D) = \{v = (v_1, \ldots, v_{d(n)}) : v_i = (u_i + \nu_i \cdot x_i) \bmod p,\ 1 \leq x_i \leq \lfloor p/2 \rfloor\}$. On Figure 1, grey areas represent the various *directions* for the central node. For any $D = (\nu_1, \ldots, \nu_{d(n)}) \in \{-1, +1\}^{d(n)}$, we call *diagonal* the set of nodes $\mathrm{diag}_u(D) = \{v = (v_1, \ldots, v_{d(n)}) : v_i = (u_i + \nu_i \cdot x) \bmod p,\ 1 \leq x \leq \lfloor p/2 \rfloor\}$. On Figure 1, the bold line represents a *diagonal* for the central node. We can prove that for any node u and its long range contact v for some distribution $\varphi^{(n)}$ of the long range links, if $v \in \mathrm{dir}_u(D)$ and $t \in \mathrm{diag}_u(D')$ for $D, D' \in \{-1, +1\}^{d(n)}$, $D \neq D'$, then greedy routing from u to t does not use the long range link (u, v). Consider now a distribution of long range links for $G^{(n)}$. We prove that routing on the diagonals is hard. For this purpose, we define an *interval* I as a connected subgraph of a diagonal. We say that an interval I of $\mathrm{diag}_u(D)$ is *good* if there exists $x \in I$ such that the long range contact y of x satisfies $y \in \mathrm{dir}_x(D)$.

A *line* L of $G^{(n)}$ in direction $D \in \{-1, +1\}^{d(n)}$ is a maximal subset of $V(G^{(n)})$ such that for any two nodes $u, v \in L$, we have $\mathrm{diag}_u(D) \cap \mathrm{diag}_v(D) \neq \emptyset$. On Figure 1, dark grey nodes belongs to a *line*. The set of all the lines in the same direction D partitions $G^{(n)}$ into n/p lines of size p. Let us partition each line into p/X disjoint intervals of same length X. This results into n/X intervals per direction, thus in total into a set S of $\frac{n}{X} \cdot 2^{d(n)}$ intervals of length X. There is a one-to-one mapping between intervals and nodes because every good interval $I \in S$ must contain a node u (called the *certificate*) whose long range contact v satisfies $v \in \mathrm{dir}_u(D)$. We have $2^{d(n)} \cdot \frac{n}{X}$ intervals in S. Since $2^{d(n)} \cdot \frac{n}{X} \leq n$, we get $X \geq 2^{d(n)}$. By the pigeonhole principle, if $X < 2^{d(n)}$, there is one interval $I = [s, t] \in S$ which is not good. Choosing $X = 2^{d(n)} - 2$ implies that greedy routing from s to t takes $2^{d(n)} - 3 \notin O(\text{polylog } n)$ steps. □

Brief Announcement: Decoupled Quorum-Based Byzantine-Resilient Coordination in Open Distributed Systems*

Alysson Neves Bessani[1,2], Miguel Correia[2],
Joni da Silva Fraga[1], and Lau Cheuk Lung[3]

[1] DAS/PGEEL, Universidade Federal de Santa Catarina, Brazil
[2] LaSIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
[3] PPGIA, Pontificia Universidade Católica do Paraná, Brazil

1 Introduction

The *tuple space coordination model*, originally introduced in the LINDA programming language [2], uses a shared memory object called a *tuple space* to support coordination that is decoupled both in time – processes do not have to be active at the same time – and space – processes do not need to know each others' addresses. The tuple space can be considered to be a kind of storage that stores *tuples*, i.e. finite sequences of values. The operations supported are essentially three: inserting a tuple in the space, reading a tuple from the space and removing a tuple from the space.

In this paper we propose an efficient Byzantine fault-tolerant implementation of a tuple space called *LBTS (Linearizable Byzantine Tuple Space)*. LBTS is implemented by a set of distributed servers and behaves according to its specification if up to a number of these servers fail in a Byzantine way. Moreover, LBTS also tolerates accidental and malicious faults in an unbounded number of the clients that use its services and satisfies two important properties: linearizability and wait-freedom (with respect to client failures). In LBTS, most operations on the tuple space are implemented by pure Byzantine quorum protocols [3,4]. However, since a tuple space is a shared memory object with consensus number 2, it cannot be implemented using only quorum protocols. In this paper we identify the tuple space operations that require stronger protocols, and show how to implement them using a modified *Byzantine Paxos* consensus protocol [1]. The philosophy behind our design is that simple operations are implemented by "cheap" quorum-based protocols, while stronger operations are implemented by more expensive protocols based on consensus.

2 LBTS Protocols

We assume an eventually synchronous system model composed by an infinite set of clients and $n \geq 4f + 1$ servers. An unbounded number of clients and at most f servers can fail in a Byzantine way. The tuple space is implemented by the

* Work supported by CNPq (project 506639/2004-5) and LaSIGE.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 554–556, 2006.

servers organized as a *f-masking Byzantine quorum system*, where each quorum contains $q = \lceil \frac{n+2f+1}{2} \rceil$ servers, ensuring an intersection of $2f+1$ servers between every two quorums of the system [3]. An important assumption of our protocols is that each tuple is unique. This assumption can be enforced in practice appending a nonce to each tuple.

In this paper we briefly describe the protocols that implement the three (non-blocking) operations of LBTS: *out*, for tuple insertion in the space; *rdp*, for tuple reading from the space; and *inp*, for tuple removal from the space. A fundamental property of the *inp* operation is that no two clients can remove the same tuple from the space.

Tuple Insertion (out). The tuple insertion protocol comprises a single access to a quorum giving the tuple being inserted. Each server that receives the tuple stores it in its local copy of the space if the tuple is not already stored and was not removed before. Notice that we are implementing a multi-writer storage but are not using timestamps. The protocol requires only two communication steps and has linear message complexity.

Tuple Reading (rdp). This operation is implemented by a protocol that executes in two phases. At first, the reading client accesses the servers requesting the tuples that match a given template. A server sends a response to the client together with the number of tuple removals it previously executed. The client is registered in a listener set and receives an update message every time a tuple that matches the given template is inserted in the server or some removal is executed. The client keeps collecting information from the servers until it receives matching tuples from a quorum of servers that removed the same number of tuples. If there is a tuple t that appears in at least $f+1$ of these responses, t is the read tuple. If this tuple appears in less than q servers, the client has to write it back to the system to ensure that it will be read in subsequent reads and satisfy linearizability. Notice that the listener communication pattern is used for a different purpose than in [4]: the reader wants to "take a photo" of the system between removals in order to define the result of the read operation. This protocol requires 2 and 4 (when write-back is needed) communication steps and has linear message complexity. Its correctness relies on the fact that all removals (*inp*) are executed in all correct servers in the same total order.

Tuple Destructive Reading (inp). The approach to implement the semantics of this operation (no two clients can remove the same tuple) is to execute all *inp* operations in the same order in all servers. This can be implemented using a total order multicast protocol based on the *Byzantine Paxos* algorithm, e.g. *BFT* [1]. BFT works briefly as follows. When a client wants to multicast a message m, it sends m to all servers. When the leader server s receives m, it gives it the next *sequence number* i and sends it to all the other servers. If server s' receives $\langle m, i \rangle$ from the leader, it has previously received m from the client, and it accepted no previous message with sequence number i, then s' *accepts* m. When this happens, s' engages in two rounds of message exchange with the other servers to do agreement on the association $\langle m, i \rangle$. When agreement is reached, m is defined as the i-th message to be delivered by correct servers. If some servers detect that the leader is faulty (e.g. because it leaves gaps in the sequence numbers), they elect another leader. LBTS' *inp* protocol is a modified version of BFT. It differs from BFT in three aspects: *(1)* when the leader s receives a $inp(\overline{t})$ request, it sends to the other servers not only the sequence number for this message but also a tuple $t_{\overline{t}}$ from its local tuple space that matches $\overline{t}$; *(2)* each server s' accepts to remove the tuple $t_{\overline{t}}$ received from the leader if the BFT conditions for acceptance are met, s' did not previously accepted the removal of $t_{\overline{t}}$, the tuple $t_{\overline{t}}$ matches the given template $\overline{t}$, and $t_{\overline{t}}$ is not forged (s' has $t_{\overline{t}}$ in its local tuple space or s' received $f+1$ signed messages from different servers ensuring that they have $t_{\overline{t}}$ in their local tuple spaces); *(3)* when a new leader l' is elected, each server sends it its protocol state and a signed set with the tuples in its local tuple space that match $\overline{t}$. This information is used by l' to build a proof for a proposal with a tuple t (in case it gets that tuple from $f+1$ servers) or $\perp$ (in case it does not). This modified version of BFT ensures total order in all *inp* executions and that the result of a *inp* is the same in all correct servers.

3 Discussion

This paper presents the first quorum-based construction for a shared memory object strictly stronger than a register. This construction is based on a combination of common quorum techniques plus three novel ones: *(i.)* instead of using timestamps, it uses a novel technique suited for collection objects, i.e., objects that store collections of elements, where the elements space is partitioned between all clients, and every element is unique; *(ii.)* it uses the listener communication pattern to capture the state of the system between executions of the read-write operations, and then apply the usual quorum-based reasoning to define the result for a read operation; and *(iii.)* it uses a modified Byzantine Paxos algorithm to do total order multicast and reach agreement about the result of an operation in a single execution. LBTS is more efficient in terms of message complexity and communication steps than a similar object would be if implemented directly on top of a BFT.

As future work, we expect to generalize the techniques used to design LBTS to define a replication algorithm that can be used to implement any shared object with consensus number greater than 1.

References

1. M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.
2. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.
3. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.
4. J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proc. of the 16th Int. Symposium on Distributed Computing, DISC 2002*, volume 2508 of *LNCS*, pages 311–325, Oct. 2002.

Brief Announcement: Optimistic Algorithms for Partial Database Replication*

Nicolas Schiper[1], Rodrigo Schmidt[2], and Fernando Pedone[1]

[1] University of Lugano, Switzerland
[2] EPFL, Switzerland

1 Introduction

Database replication protocols based on group communication have recently received a lot of attention. The main reason for this stems from the fact that group communication primitives offer adequate properties, namely agreement on the messages delivered and on their order, to implement synchronous database replication. Most of the complexity involved in synchronizing database replicas is handled by the group communication layer. Previous work on group-communication-based database replication has focused mainly on full replication. However, full replication might not always be adequate. First, sites might not have enough disk or memory resources to fully replicate the database. Second, when access locality is observed, full replication is pointless. Third, full replication provides limited scalability since every update transaction should be executed by each replica.

2 Partially-Replicated Database State Machine

In this paper, we discuss an extension of the Database State Machine (DBSM) [3], a group-communication-based database replication technique, to partial replication. The DBSM is based on the deferred update replication model [1]. Transactions execute locally on one database site and their execution does not cause any interaction with other sites. Read-only transactions commit locally only; update transactions are atomically broadcast at commit time for certification. The certification test ensures *one-copy serializability* [1] and requires every database site to keep the *writesets* of committed transactions. The certification of a transaction T consists in checking that T's *readset* does not contain any outdated value, i.e., no committed transaction T' wrote a data item x after T read x.

A straightforward way of extending the DBSM to partial replication consists in executing the same certification test as before but having database sites only process update operations for data items they replicate. But as the certification test requires storing the writesets of all committed transactions, this strategy

* The work presented in this paper has been partially funded by the Hasler Foundation, Switzerland (project #1899) and SNSF, Switzerland (project #200021-107824).

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 557–559, 2006.

defeats the whole purpose of partial replication since replicas may store information related to data items they do not replicate. Ideally, sites would only store transaction information related to the data items they replicate. However, we do not want to rule out solutions that rely on building blocks (e.g., consensus) that may be oblivious to the data items replicated by the sites. In such cases, sites may momentarily store transactions unrelated to the data items they replicate. Moreover, we want to make sure each transaction is handled by a site at most once. If sites are allowed to completely forget past transactions, this constraint cannot obviously be satisfied. We capture these constraints with the following property:

- *Quasi-Genuine Partial Replication:* For every submitted transaction T, correct database sites that do not replicate data items read or written by T permanently store not more than the identifier of T.[1]

Consider now the following modification to the DBSM, allowing it to ensure Quasi-Genuine Partial Replication. Besides atomically broadcasting transactions for certification, database sites periodically broadcast "garbage collection" messages. When a garbage collection message is delivered, a site deletes all the writesets of previously committed transactions. When a transaction is delivered for certification, if the site does not contain the writesets needed for its certification, the transaction is conservatively aborted. Since all sites deliver both transactions and garbage collection messages in the same order, they will all reach the same outcome after executing the certification test. This mechanism, however, may abort transactions that would be committed in the original DBSM. In order to rule out such solutions, we introduce the following property:

- *Non-Trivial Certification:* If there is a time after which no two submitted transactions conflict, then eventually transactions are not aborted by certification.

In [4], we present two algorithms for partial database replication for clusters of servers. Our algorithms satisfy both Quasi-Genuine Partial Replication and Non-Trivial Certification and are optimistic, i.e., we assume *spontaneous total order*: with high probability messages sent to all servers in the cluster reach all destinations in the same order.

To the best of our knowledge, [2] and [5] are the only papers addressing partial database replication using group communication primitives. In [2], every operation of a transaction on data item x is multicast to its replicas and a final atomic commit protocol ensures transaction atomicity. In [5], the authors extend the DBSM for partial replication by adding an extra atomic commit protocol. Both of our algorithms compare favorably to [2,5]: they either have a lower latency or make weaker assumptions about the underlying model, i.e., they do not require perfect failure detection.

[1] Notice that even though transaction identifiers could theoretically be arbitrarily large, in practice, 4-byte identifiers are enough to uniquely represent 2^{32} transactions.

3 Final Remarks

This short paper defines two properties, Quasi-Genuine Partial Replication and Non-trivial Certification. These properties characterize our view of partial replication in the DBSM. The first property forbids replicas to permanently store information about data items they do not replicate; the second property prevents trivial solutions that would unnecessarily abort transactions in an attempt to satisfy the first property. Two algorithms for partial replication in the DBSM that ensure these two properties are presented in [4]. In the future, we intend to better characterize partial replication and devise efficient algorithms that satisfy a stronger property than Quasi-Genuine Partial Replication. Intuitively, Genuine Partial Replication should be defined such that only database sites that replicate data items touched by a transaction T are involved in its certification.

References

1. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
2. Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 284–291, 2001.
3. F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.
4. N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. Technical Report 2006, University of Lugano, 2006.
5. A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the 1st International Symposium on Network Computing and Applications (NCA)*, October 2001.

Brief Announcement: Performance Analysis of Cyclon, an Inexpensive Membership Management for Unstructured P2P Overlays

François Bonnet[1], Frédéric Tronel[2], and Spyros Voulgaris[3]

[1] École Normale Supérieure de Cachan/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[2] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[3] Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Motivations. Unstructured overlays form an important class of peer-to-peer networks, notably for content-based searching algorithms. Being able to build overlays with low diameter, that are resilient to unpredictable joins and leaves, in a totally distributed manner is a challenging task. Random graphs exhibit such properties, and have been extensively studied in literature. Cyclon algorithm is an inexpensive gossip-based membership management protocol described in detail in [1] that meets these requirements.

An Overview of Cyclon. For a detailed description of Cyclon algorithm, the reader should refer to [1]. Briefly, Cyclon supports two different modes of operation : a basic shuffling mode, and an enhanced one. The basic mode, the only one to be studied in this article is a purely random mode, while the second mode uses a timestamp mechanism to improve performance with respect to node failures behavior. Each node maintains a cache of neighbor nodes of size c, hence each node knows exactly c nodes in the overlay. To correctly initialize nodes caches, we assume the existence of a predefined set of well-known supernodes. During the execution of the protocol, each node performs periodically a shuffle step. For a given node p, a shuffle step consists in contacting one node q among its neighbors. Then p and q exchange $\ell \leq c$ nodes from their respective caches. ℓ is a parameter of the algorithm. Counterintuitively, simulations in [1], have shown that the influence of parameter ℓ is negligible (except for limit cases, when ℓ is close to 1 or c). One of the most fundamental operation performed by the shuffling step is that p sends its own identity to q, and remove q from its own cache. Consequently, the edge from p to q is reversed by the shuffling step. This guarantees the connectivity of the underlying overlay. In this paper, we propose two models to analyse Cyclon performances in term of convergence speed, and quality of the obtained overlay. In our work we evaluate this quality from the distribution of the in-degree[1] of nodes. We are interested by this distribution since it is highly related to the robustness of the overlay in the presence of failures. This gives also an indication of the distribution of resource

[1] The number of nodes that have an edge directed to the considered node. It is an integer in $[0, n-1]$ since we do not authorized loop edges.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 560–562, 2006.

usage (processing, bandwidth) across nodes. We are looking for a distribution as uniform as possible.

Model #1. We assume that there are n nodes in the system, gossip exchanges are atomic, and are triggered by a global scheduler which picks at random the next process to perform a shuffle operation with uniform probability. This is of course, a rather coarse model of reality, where each process would certainly locally triggers its shuffling operation, using timeout expiration. In fact this model corresponds to a complete asynchronous system; even though this model is questionable (see [2]) it has been introduced by [3] and [4]. With this random scheduling a given process has probability $1-e^{-1}$ of taking at least one computation step, when exactly n steps are triggered. This must be compared to a real system based on local uniform time triggers, where the same process would have probability 1 to perform a step, every n steps counted globally. We are interested in evaluating the in-degree evolution of a given node since it is a good measure of the quality of the obtained overlay. We model Cyclon algorithm by a discrete time Markov chain (DTMC) whose states space S is the possible in-degrees for a given process. Note that we can focus on a particular process because they are all equivalent with respect to the scheduler. The evolution of the in-degree of a node after a shuffle step depends only on its value before the step; the impact of the detailed structure of the network is negligible. Moreover during a step, the in-degree can only change by one. Thus, the n-square matrix M_1 associated to this DTMC is a tridiagonal one. Its upper diagonal is $m^1_{i,i+1} = -\frac{i}{n(n-1)} + \frac{1}{n}$, while its lower diagonal is $m^1_{i,i-1} = \frac{i}{n(n-1)} \frac{n-1-c}{c}$. Being a stochastic tridiagonal matrix, diagonal elements $m_{i,i}$ of M_1 are equal to $1 - m^1_{i,i+1} - m^1_{i,i-1}$. We show that the generating function $G_\lambda(z) = \sum_{i=0}^{n-1} v_\lambda[i] z^i$ associated to the eigenvector v_λ[2] satisfies a first-order differential equation whose solutions are equal to $\left(\frac{z-1}{cz+n-c-1}\right)^{cn(\lambda-1)} \left(\frac{cz-n-c-1}{n-1}\right)^{n-1}$. Since by definition, this function must be a polynomial of degree $n-1$, we can conclude that eigenvalues are $1-\frac{k}{nc}$ with $k \in [0, n-1]$. In particular we obtain for $k=0$, a closed form for the generating function $\pi(z)$ associated with the stationary distribution $\pi = v_1$, namely $\pi(z) = \left(\frac{cz+n-c-1}{n-1}\right)^{n-1}$. Thus $\pi[i] = \binom{n-1}{i} \left(\frac{n-c-1}{n-1}\right)^{n-1-i} \left(\frac{c}{n-1}\right)^{i}$. This corresponds exactly to the in-degree distribution of a purely random directed graph where each vertex has exactly c outgoing edges, a highly desirable property for unstructured overlays. Using well-known properties on generating functions we can establish that the mean value $\overline{\pi}$ of stationary distribution is equal to c, which naturally satisfies the balance equation in a directed graph[3]. Similarly we can establish that standard deviation of π is equal to $c + O(1/n)$. Since we have access to the eigenvalues, and in particular the second largest one, namely $1-\frac{1}{nc}$, we can establish an upper bound on the convergence speed of the DTMC. Using classical linear algebra, we can show that $\max_{X_0} \frac{1}{2} \|X_0 M_1^t - \pi\|_1 \leq \left(1 - \frac{1}{nc}\right)^t$

[2] v_λ is such that $v_\lambda M_1 = \lambda v_\lambda$ for a given λ called an eigenvalue of M_1.

[3] The number of outgoing edges, nc in this particular case, is equal to the number of ingoing edges, which is equal to $n\overline{\pi}$.

where X_0 denotes the initial distribution. Mixing time $\tau_1(\epsilon)$ as defined in [5] is thus bounded by $nc \log \epsilon^{-1} + O(1)$, which shows that Cyclon is a fast mixing algorithm [4].

Model #2. We consider a more refined model, where processes are fairly scheduled. We consider now that a step in our model corresponds to a whole cycle of the protocol, i.e. in a step every node performs one and exactly one shuffle. Contrary to model #1, this model corresponds to a synchronous system: all nodes execute the same number of exchanges and at the same time. This refinement comes at the price of a more complex stochastic matrix M_2. The evolution of the in-degree of a node after a (model) step still depends only on its value before the step, but its variation may now be larger than one. In particular, M_2 is an lower hesselberg[5] matrix whose general term is $m^2_{0 \le j-1 \le i \le n-1} = \binom{i}{j-1} \left(\frac{1}{c}\right)^{i+1-j} \left(\frac{c-1}{c}\right)^{j-1}$. We show that the generating function $G_\lambda(z)$ is solution of the functional equation : $\lambda G_\lambda(\frac{cz-1}{c-1}) = \frac{cz-1}{c-1} G_\lambda(z) + \frac{c(1-z)}{c-1} \left(\frac{cz-1}{c}\right)^{n-1} v_\lambda[n-1]$. We can extract and solve a recurrence equation giving $G_\lambda(z)$ and all of its successive derivatives $G_\lambda^{(k\ge0)}(z)$ at point $z = 1$. For $\lambda = 1$, by using a Taylor series expansion, and by the fact that $G_1(z)$ is a polynomial of degree $n-1$, we have access to a closed formula for $\pi(z)$ the generating function associated to the stationary distribution of the considered DTMC, namely $\pi(z) = \sum_{k=0}^{n-1} \frac{G_1^{(k)}(1)}{k!}(z-1)^k$. For $\lambda \ne 1$, the fact that $G_\lambda(z)$ is a polynomial of degree $n-1$ can be expressed as a set of constraints on the successive derivatives at point $z = 1$, namely $G_\lambda^{(k\ge n)}(1) = 0$. These constraints reduce to a polynomial of degree $n-1$ whose roots are exactly the $n-1$ eigenvalues $\lambda < 1$. We show that the second largest eigenvalue is smaller than $1 - \frac{1}{c}$. Hence mixing time $\tau_2(\epsilon)$ is bounded by $c \log \epsilon^{-1} + O(1)$. Note that this is compatible with model #1. Indeed in model #2, each step of the Markov process corresponds to n steps of the previous Markov process. This explains why $\frac{\tau_1(\epsilon)}{\tau_2(\epsilon)} = n$. The reader is invited to refer to [6] for a detailed version of the results.

References

1. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: Inexpensive membership management for unstructured P2P overlays. J. Network Syst. Manage **13** (2005)
2. Aspnes, J.: Randomized protocols for asynchronous consensus. DISTCOMP: Distributed Computing **16** (2003)
3. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. JACM: Journal of the ACM **32** (1985)
4. Aspnes, J.: Fast deterministic consensus in a noisy environment. In: PODC. (2000) 299–308
5. Randall, D.: Rapidly mixing Markov chains with applications in computer science and physics. Computing in Science and Engineering **8** (2006) 30–41
6. Bonnet, F., Tronel, F., Voulgaris, S.: Performance analysis of cyclon, an inexpensive membership management for unstructured p2p overlays. Technical Report 1807, IRISA (2006)

[4] $\tau_1(\epsilon)$ is bounded by a polynomial of $\log \epsilon^{-1}$ and n. See [5] for a precise justification.

[5] A matrix whose terms above the upper diagonal are zero.

Brief Announcement: Decentralized, Connectivity-Preserving, and Cost-Effective Structured Overlay Maintenance

Yu Chen and Wei Chen

Microsoft Research Asia
{ychen, weic}@microsoft.com

Since their introduction, structured overlays have been used as an important substrate for many peer-to-peer applications. In a structured peer-to-peer overlay, each node maintains a partial list of other nodes in the system, and these partial lists together form an overlay topology that satisfies certain structural properties (e.g., a ring). Various system conditions, such as node joins and leaves, message delays and network partitions, affect overlay topology, so overlay topology should adjust itself appropriately to maintain structural properties. Topology maintenance is crucial to the correctness and the performance of applications built on top of the overlay.

Most structured overlays are based on a logical key space, and they can be conceptually divided into two components: leafset tables and finger tables.[1] The leafset table of a node keeps its logical neighbors in the key space, while the finger table keeps relatively faraway nodes in the key space to enable fast routing along the overlay topology. The leafset tables are the key for maintaining a correct overlay topology since finger tables can be constructed efficiently from the correct leafset tables. Therefore, our study focuses on leafset maintenance. In particular, we focus on one-dimensional circular key space and the ring-like leafset topology in this space, similar to many studies such as [1] and [2].

Leafset maintenance is a continuously running protocol that needs to deal with various system conditions. An important criterion for leafset maintenance is convergence, that is, the leafset topology can always converge back to the desired structure after the underlying system stabilizes (but without knowing about system stabilization), no matter how adverse the system conditions were before system stabilization.

Existing studies on overlay maintenance have various limitations. Some investigate system level improvements without formal proofs on protocol guarantees [3,4,5]; some provide formal proofs to their protocols but do not address fault tolerance and convergence [6]; and some propose one-shot protocols for fast overlay construction under known system stabilization conditions without considering adverse effects before system stabilization [7]. There are some studies on self-stabilizing protocols, which guarantee convergence with arbitrary initial states of processes and communication channels. However, among these protocols, some rely on a continuously available bootstrap system to actively participate in the self stabilization process [8,9]; some incur a significant amount of cost by maintaining a large membership list [10,11]; and some only provides a special case for self stabilization [12].

[1] The term leafset is originally used in Pastry [1] while the term finger is originally used in Chord [2].

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 563–565, 2006.

In contrast, we have designed a leafset maintenance protocol [13] that removes the limitations in existing protocols. In particular, we provide a precise specification for leafset maintenance protocols with cost effectiveness requirements. All properties of the specification are desired by applications, while together they prohibit protocols with the above limitations. We then provide a complete protocol with proof showing that it satisfies the specification.

Our study is based on a decentralized and symmetric system model in which any node may join and leave the system or crash, and there is no special group of nodes that is always available to act as a bootstrap system. We assume the system eventually stabilizes, but the protocol does not know the system stabilization time, so it cannot easily nullify the impact of system conditions before system stabilization.

Based on the system model, we provide a set of properties as a rigorous specification of the leafset maintenance protocol. One important property is Connectivity Preservation: If the underlying system stabilizes and the topology is still connected, the maintenance protocol should not break the connectivity of the topology while it evolves the topology towards the correct configuration. Moreover, we explicitly put requirements on cost effectiveness: The messaging and local state costs on a node could depend on the size of its leafset table, but should not depend on the size of the system.

To deal with topology partitions caused by network partitions, we define a simple add$(contacts)$ interface, through which an application can add new contact nodes into a leafset to heal topology partitions. Our specification makes it clear that after the underlying system stabilizes, the add$(contacts)$ interface only needs to be invoked once at one node to bridge the partitioned topology. Afterwards, the protocol should continue to preserve connectivity and converge the topology by itself without further help. Hence, the reliance on an outside mechanism such as a bootstrap system is kept at the minimum.

Our specification prohibits protocols that either rely on a continuously available bootstrap system, or maintain large local membership lists that is related to the size of the system, or assume that the system stabilization condition is known. Therefore, it only permits protocols that remove these limitations existed in previous protocols.

We have designed a protocol and proved that it satisfies our specification. The core of protocol is to preserve the connectivity of the topology during its convergence. To be cost-effective, the protocol needs to remove extra entries in leafsets (as in many other protocols), but such removals may break topology connectivity. To avoid such situation, in our protocol if a node x wants to remove another node z from its leafset, it has to find a replacement node y such that x can still reach z in the topology through y. However, the task of finding a replacement that guarantees connectivity is not trivial, because these tasks are run concurrently by all nodes and they may interfere with each other. In particular, some tasks may be started before system stabilization and cause incorrect replacements, and thus breaking the topology even long after the system stabilizes. We employed a round-number mechanism to eliminate such interference.

Besides connectivity preservation, the protocol also needs to guarantee progress. A subtle issue in guaranteeing progress is that our protocol execution is not closed: Because of the interface add$(contacts)$, an application may continue to invoke this function to bridge potentially partitioned components. Without careful design, these invocations may interfere with leafset convergence and cause livelocks. We employed a couple of mechanisms to avoid such livelock scenarios.

Finally, we also studied some heuristics to significantly improve the convergence speed of our protocols. We verified these heuristics by simulations on several classes of initial topologies and showed that it improves convergence speed from $O(N)$ to $O(\log N)$, where N is the number of online nodes when the system stabilizes.

As a summary, our protocol maintains a small leafset independent of the system size, guarantees leafset convergence as long as the topology is still connected after the system stabilizes, and if it is not connected, one invocation of the add() interface is enough to bridge disconnected components and lead to convergence. Moreover, continuous invocations of add() interface will not interfere with the convergence process.

References

1. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of IFIP/ACM Middleware, Heidelberg, Germany (2001)
2. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the SIGCOMM'01 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, San Deigo, California, USA (2001)
3. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proceedings of the USENIX Annual Technical Conference, Boston, Massachusetts, USA (2004)
4. Castro, M., Costa, M., Rowstron, A.: Performance and dependability of structured peer-to-peer overlays. In: Proceedings of the International Conference on Dependable Systems and Networks 2004, Palazzo dei Congressi, Florence, Italy (2004)
5. Haeberlen, A., Hoye, J., Mislove, A., Druschel, P.: Consistent key mapping in structured overlays. Technical Report TR05-456, Rice Computer Science Department (2005)
6. Li, X., Misra, J., Plaxton, C.G.: Active and concurrent topology maintenance. In: Proceedings of the 18th International Symposium on Distributed Computing, Trippenhuis, Amsterdam, the Netherlands (2004)
7. Angluin, D., Aspnes, J., Chen, J.: Fast construction of overlay networks. In: Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures, Las Vegas, Nevada, USA (2005)
8. Dolev, S., Kat, R.I.: Hypertree for self-stabilizing peer-to-peer systems. In: Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications. (2004)
9. Shaker, A., Reeves, D.S.: Self-stabilizing structured ring topology p2p systems. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, Konstanz, Germany (2005)
10. Jelasity, M., Babaoglu, O.: T-Man: Gossip-based overlay topology management. In: Proceedings of the 3rd International Workshop on Engineering Self-Organising Applications, Utrecht, The Netherlands (2005)
11. Montresor, A., Jelasity, M., Babaoglu, O.: Chord on demand. In: Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing, Konstanz, Germany (2005)
12. Balakrishnan, H., Karger, D., Liben-Nowell, D.: Analysis of the evolution of peer-to-peer systems. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, Monterey, California, USA (2002)
13. Chen, Y., Chen, W.: Self-stabilizing, cost-effective, and fast-convergent structured overlay maintenance. Technical Report MSR-TR-2006-56, Microsoft Research (2006)

Brief Announcement
Monitoring of Linear Distributed Computations

Anton Esin[1], Rostislav Yavorskiy[1], and Nikolay Zemtsov[2]

[1] Steklov Mathematical Institute
Gubkina 8, Moscow, 119991, Russia
{esin, rey}@mi.ras.ru
[2] Department of Mathematical Logic and Theory of Algorithms
Faculty of Mechanics and Mathematics
Moscow State University
Moscow, 119992, Russia
Nikolay.Zemtsov@mtu-net.ru

Abstract. We consider a general formal model for monitoring of a local network behavior. Monitored behavior of the system corresponds to the dynamics induced on the observed state space by the corresponding projection function. An algorithm is developed for restoration of information about the system state on the base of the observations.

The theoretical work is backed by our experiments on monitoring Spanning Tree Protocol.

1 Formal Model of a Local Network Behavior

We assume a local network to be a state transition system of the following form. The network topology is characterized by a fixed finite graph $G = (V, E)$, where V denotes the set of vertices, E stands for edges. The state of the whole system, Ω, is a union of all the states of the vertices. Without loss of generality one can assume that all the vertices have the same state space S, so $\Omega = S^{|V|}$. A system transition is a nondeterministic update of the global state such as

- update of a single node (e.g. timer event or communication with environment);
- update of two adjacent nodes (message passing or so);
- simultaneous update of more than two nodes (different kinds of broadcasting).

It turns out that in many cases the state of a vertex is characterized by a fixed number of integer valued variables, so we assume $S = \mathbb{Z}^m$. Moreover, network protocols rarely make use of rather advanced mathematics; as a rule, linear arithmetic is enough to express the transition function. So, the transition relation τ is a relation definable over natural numbers by means of addition, multiplication by a constant, inequality and boolean connectives (note that case splitting is expressible, so min, max are also allowed).

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 566–568, 2006.

2 The Projecting Function

Given a nonempty set M *a projecting function* is a mapping $\pi : \Omega \to M$. Informally, M is the state space of a system monitor, π is the correspondence between the states of the whole system and the monitored ones.

In the context described above it is natural to consider the following two kinds of the projecting function that differ by the function range. Namely

1. $M = \mathbb{Z}^m$, where $m \leq n$. In practice it means that monitor observes some of the state variables and/or some aggregative integer valued parameters.
2. M is a finite set. Then, without loss of generality one can assume M to be an initial segment of natural numbers $M = \{1, \ldots, p\}$. This corresponds to observing fixed number of boolean (or other finitely valued) parameters of the system.

3 The Observed Dynamics

Let $c = (s_0, s_1, \ldots, s_l)$ be a computation, that is $s_{i+1} = \tau(s_i)$. Then, projection of the computation $\pi(c)$ is the sequence $(\pi(s_0), \pi(s_1), \ldots, \pi(s_l))$ over M.

The main goal of our research here is to study the relation between c and $\pi(c)$. The following tasks seem to be the most natural. First, given Ω, τ, π and the monitored behavior $(\pi(s_0), \pi(s_1), \cdots, \pi(s_l))$ to restore maximum information about the system dynamics $(s_0, s_1, \ldots, s_l)$. The second task corresponds to the situation when the user has more freedom in choosing the projection function. Then the goal is to find the "best quality" monitor π. Another interesting task is the following. Given two hypotheses about the system behavior, τ_1 and τ_2, to construct a monitor π such that the corresponding monitored behavior will be different.

4 The Theoretical Results

Theorem 1. *For a linear nondeterministic transition system (Ω, τ) and a piecewise linear projection function π the set of projections of all possible computations is decidable.*

The proof provides an algorithm, which takes a sequence γ over M as input and returns an answer, whether the given sequence could be a projection of some trajectory c for (Ω, τ).

The second theorem provides sufficient conditions for π to be the most informative in the sense that it uses all the expressive power of M.

Let $M = \{1, \ldots, p\}$ and M^* denote the set of all finite words over M.

Theorem 2. *If for any $i \in M$ one has $\tau(\pi^{-1}(i)) = \Omega$ then every word in M^* is a projection of some computation of (Ω, τ).*

Let $L(\Omega, \tau, \pi)$ denote the set of all words of the form $\pi(c)$, where c ranges over all possible computations of (Ω, τ). In these terms the previous theorem provides a criteria for $L(\Omega, \tau, \pi) = M^*$

The following theorem concerns the task to distinguish between two possible behaviors of the monitored system.

Theorem 3. *a) Let $|M| = 2$. Then there exist two different transition relations τ_1 and τ_2 such that for any monitor $\pi : \Omega \to M$ one has $L(\Omega, \tau_1, \pi) = L(\Omega, \tau_2, \pi)$.*

b) Let $|M| \geq 3$. Then for any different τ_1 and τ_2 one can construct monitor $\pi : \Omega \to M$ such that there exists at least one word of length 2 that distinguishes between $L(\Omega, \tau_1, \pi)$ and $L(\Omega, \tau_2, \pi)$.

5 Monitoring the Spanning Tree Protocol for Recognition of Topology Changes

To verify the framework described above we made experiments with monitoring the STP traffic through a single sensor in a local area network. (See e.g. [5] regarding the STP protocol.) During the experiment we disconnected switches one edge at a time and captured STP traffic observed by our sensor. The experiment showed that for a given configuration of the network topology and a sensor one can construct a monitor for recognizing the topology changes.

References

1. Vladimir Filatov and Rostislav Yavorskiy. *Scenario based analysis of linear computations.* Proceedings of 12th International Workshop on Abstract State Machines ASM'05, March 8-11, 2005, France, pp. 167-174.
2. Yuri Gurevich. *Abstract State Machines: An Overview of the Project.* in "Foundations of Information and Knowledge Systems" editors Dietmar Seipel and Jose Maria Turull-Torres. Springer Lecture Notes in Computer Science volume 2942 (2004), pages 6–13
3. Egon Börger and Robert Stärk, **Abstract State Machines. A method for High-Level System Design and Analysis.** Springer-Verlag 2003.
4. Uwe Glaesser, Yuri Gurevich and Margus Veanes, *Abstract Communication Model for Distributed Systems.* IEEE Transactions on Software Engineering Vol. 30, no. 7 (2004), pp. 458-472.
5. The OSI Reference Model http://www.cisco.com/warp/public/473/5.html

Brief Announcement: Communication-Optimal Implementation of Failure Detector Class $\Diamond\mathcal{P}$[*]

Mikel Larrea, Alberto Lafuente, and Joachim Wieland

The University of the Basque Country
20018 San Sebastián, Spain
{mikel.larrea, alberto.lafuente}@ehu.es, joachim.wieland@rwth-aachen.de

Abstract. Several algorithms implementing the failure detector class $\Diamond\mathcal{P}$ have been proposed in the literature. Regarding communication efficiency, a performance parameter based on the number of links that carry messages forever, algorithms using n links have been proposed, being n the number of processes in the system. In this paper, we show that communication-optimal $\Diamond\mathcal{P}$ algorithms, i.e., using only $\mathcal{C}$ links, being $2 \leq \mathcal{C} \leq n$ the number of correct processes, can be implemented. The price to pay for obtaining communication optimality is a higher number of messages exchanged when a failure suspicion occurs. However, one of the algorithms we propose shows that this cost can be linear in n.

1 Introduction

Unreliable failure detectors, proposed by Chandra and Toueg [2], are a mechanism providing (possibly incorrect) information about process failures. This mechanism has been used to solve several problems in asynchronous distributed systems, in particular the Consensus problem [6]. In this paper, we focus on the *Eventually Perfect* failure detector class, denoted $\Diamond\mathcal{P}$.

In [4] we have proposed a family of heartbeat-based algorithms implementing $\Diamond\mathcal{P}$ which are based on a ring arrangement of processes. The algorithms are *communication-efficient* [1], i.e., eventually only n unidirectional links carry messages forever, outperforming previously proposed algorithms for $\Diamond\mathcal{P}$. In this paper we show how *communication-optimal* [4] algorithms for $\Diamond\mathcal{P}$ are possible, in which eventually only $\mathcal{C}$ unidirectional links carry messages forever, being $2 \leq \mathcal{C} \leq n$ the number of correct processes.

In our system model, every pair of processes is connected by two unidirectional and reliable communication links. Processes can only fail by crashing and crashes are permanent. We consider that processes are arranged in a logical ring. We will use the functions $pred(p)$ and $succ(p)$ respectively to denote the predecessor and the successor of a process p in the ring. Concerning timing assumptions, we consider a partially synchronous model [2,3] for just the links between every correct process and its correct successor in the ring.

[*] Research partially supported by the Spanish Research Council (MCyT), under grant TIN2004-07474-C02-02, the Basque Government, under grant S-PE051K06, and the Gipuzkoa Provincial Council, under grant OF-204/2005.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 569–571, 2006.

Every process p executes the following:

```
procedure $update\_pred\_and\_succ()$
    $pred_p \leftarrow p$'s nearest predecessor $r$ in the ring such that $Balance_p(r) \leq 0$
    $succ_p \leftarrow p$'s nearest successor $r$ in the ring such that $Balance_p(r) \leq 0$
end procedure

    $pred_p \leftarrow pred(p)$
    $succ_p \leftarrow succ(p)$
    for all $q \in \Pi$: $\Delta_p(q) \leftarrow$ default time-out interval
    for all $q \in \Pi$: $Balance_p(q) \leftarrow 0$

    cobegin

    || Task 1: repeat periodically
        if $succ_p \neq p$ then
            send ($p$-is-alive) to $succ_p$

    || Task 2: repeat periodically
        if $pred_p \neq p$ and $p$ did not receive ($pred_p$-is-alive)
                                    during the last $\Delta_p(pred_p)$ ticks of $p$'s clock then
            $Balance_p(pred_p) \leftarrow Balance_p(pred_p) + 1$
            r-broadcast ($SUSPICION, pred_p$) to the rest of processes
            $update\_pred\_and\_succ()$

    || Task 3: when r-deliver ($SUSPICION, r$)
        if $r \neq p$ then
            $Balance_p(r) \leftarrow Balance_p(r) + 1$
            $update\_pred\_and\_succ()$
        else
            r-broadcast ($REFUTATION$, $p$) to the rest of processes

    || Task 4: when r-deliver ($REFUTATION$, $q$) for some $q$
        $Balance_p(q) \leftarrow Balance_p(q) - 1$
        $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
        $update\_pred\_and\_succ()$

    coend
```

Fig. 1. A communication-optimal $\diamond\mathcal{P}$ using reliable broadcast

2 A Basic Communication-Optimal Implementation of $\diamond\mathcal{P}$

Figure 1 presents a communication-optimal implementation of $\diamond\mathcal{P}$ using reliable broadcast. Every process p sends periodical heartbeats to its successor in the ring and monitors its predecessor in the ring. Besides this, p reliably broadcasts suspicions and, when it is erroneously suspected, refutations. $Balance_p$ accounts suspicions and refutations for every other process. If $Balance_p(q) > 0$, then p suspects q; else, q is trusted by p. $Balance_p$ provides the properties of $\diamond\mathcal{P}$.

3 Optimizations and Performance Analysis

In [5] we present two optimized versions of the previous algorithm that do not use reliable broadcast. In one of them, processes re-send suspicions, keeping messages simple. In the other, we try to maintain as low as possible the number of messages exchanged as a consequence of a suspicion, at the cost of increasing the size of messages by adding more information. Figure 2 summarizes the communication costs of the algorithms, which are expressed in terms of regular heartbeat messages exchanged (which corresponds to the number of unidirectional links used in stability), and extra messages exchanged to manage an erroneous suspicion (l stands for the size of the *local* list of suspected processes). The cost of the communication-efficient $\diamond\mathcal{P}$ algorithms proposed in [4] is also included. As it can be observed, communication optimality is obtained at the price of sending more messages during stabilization.

Algorithm	# regular heartbeats (communication efficiency)	# extra messages to manage an erroneous suspicion
Figure 1	C (communication-optimal)	$O(n^2)$
Optimized_1 [5]	C (communication-optimal)	$3(n-1)-1+(l-1)^2$
Optimized_2 [5]	C (communication-optimal)	$3(n-1)-1$
Algorithms in [4]	n (communication-efficient)	From 0 to $2n$

Fig. 2. Communication cost of algorithms implementing $\diamond\mathcal{P}$

References

1. M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC'2001)*, pages 108–122, Lisbon, Portugal, October 2001. LNCS 2180, Springer-Verlag.
2. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
3. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
4. M. Larrea, A. Lafuente, and J. Wieland. Communication-efficient implementation of $\diamond\mathcal{P}$ with reduced detection latency. Technical Report EHU-KAT-IK-02-06, The University of the Basque Country, February 2006. Available at http://www.sc.ehu.es/acwlaalm/.
5. M. Larrea, A. Lafuente, and J. Wieland. Communication-optimal implementation of failure detector class $\diamond\mathcal{P}$. Technical Report EHU-KAT-IK-08-06, The University of the Basque Country, May 2006. Available at http://www.sc.ehu.es/acwlaalm/.
6. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

Brief Announcement: Synchronous Distributed Algorithms for Node Discovery and Configuration in Multi-channel Cognitive Radio Networks

Srinivasan Krishnamurthy, R. Chandrasekaran, Neeraj Mittal, and S. Venkatesan

School of Engr. and Comp. Science, The University of Texas at Dallas, Richardson TX, USA

Introduction and Related Work. Cognitive Radios (CR) [4] enable flexible and improved radio spectrum utilization by allowing a group of CR nodes to utilize unused channels without interference with the channels' owners [1]. A CR node has a set of available (wireless) channels and this set varies with time, location and activities of the primary (licensed) users. Determining a node's neighborhood and common channels for communication is non-trivial. We propose several efficient distributed algorithms for *node discovery and configuration* (*NDC*) for various CR node models. Krishnamurthy et al. [3] proposed a solution for *NDC* without a control channel. In their model, each node has a single transceiver. If N is the maximum number of nodes and M is the maximum number of channels, their algorithm uses $2NM$ timeslots for *NDC*. Other works assume a common control channel, or every node is equipped with a separate radio interface for each channel or an identical frequency distribution at each node. Please see [2] for a thorough discussion of related work. Here, we extend the algorithms of [3] to other CR node models.

System Model. We assume that during *NDC*, the CR network topology is a static *multi-hop multi-channel* wireless network. Each node i has a unique id $\mathcal{UID}_i$ in the range $[1 \ldots N]$, where N is an upper bound on the total number of nodes. We assume $\mathcal{UID}_i = i$. All nodes know N. Let $\mathcal{A}_{univ} = \{c_1, c_2 \ldots c_M\}$ represent the universal set of available channels. All nodes know M and $\mathcal{A}_{univ}$. Every channel has a unique id and all nodes know the mapping between the channel id and its frequency *a priori*. Node i is aware of its channel availability set $\mathcal{A}_i$ and it can receive and transmit on any channel of $\mathcal{A}_i$. Communication is loss-free. Nodes i and j are said to be *neighbors* if i and j are within each other's radio range and $\mathcal{A}_i \cap \mathcal{A}_j \neq \emptyset$. Communication between non-neighbors is by multi-hop transmissions. The set of channels that is common to i and nodes that are within k hops from i is referred to as the *k-local channel set*, $\mathcal{L}_i^k$. Node i needs to know $\mathcal{L}_i^2$ for collision-free communication with its neighbors.

Nodes invoke *NDC* every T time units to account for changes in network topology and/or channel availability sets. The starting times of *NDC* are known to all. The clocks of the nodes are synchronized. *NDC* is said to be complete when each node i determines its set of neighbors and its 2-hop local channel set $\mathcal{L}_i^2$. Time is divided into *timeslots* of equal duration. A message transmitted by a node is *delivered* to all its neighbors in the same timeslot. A receiver successfully *receives* a message if and only if there is exactly one message being delivered at that timeslot on the channel it is tuned to. If two or more neighbors of a node i transmit on the same channel in a given timeslot, a *collision* occurs and i does not receive any of those transmitted messages. Nodes cannot distinguish between a collision and background noise.

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 572–574, 2006.

Our Contributions. *NDC* consists of two rounds of identical duration. A *round* is the number of timeslots required for each node to communicate with all of its neighbors. Each timeslot is assigned to one or more node-channel pairs depending on the model. When a node-channel pair (i, c) is assigned to a timeslot t, node i transmits on channel c during timeslot t only if $c \in \mathcal{A}_i$; i remains silent otherwise. During round 1, each node i communicates its channel availability set $\mathcal{A}_i$ and receives $\mathcal{A}_j$ from each neighbor j. Thus, node i finds its neighbors $\mathcal{NBR}_i$ and their respective channel availability sets by the end of round 1. Node i then computes $\mathcal{L}_i^1$. In round 2, i sends $\mathcal{L}_i^1$ to its neighbors.

Limited Channel Divergence with Single Transceiver: Suppose that channel availability across nodes does not vary drastically. Such a situation may arise in scenarios where the CR network is deployed in regions that have a scarce population of primary users and/or other sources of interference. We assume that the sets of channels available to any pair of neighboring nodes i and j differ by at most k i.e. $|\mathcal{A}_i - \mathcal{A}_j| \leq k$, and that the nodes know k. Since $\mathcal{A}_i$ and $\mathcal{A}_j$ differ by at most k channels, if i transmits on any subset of $(k + 1)$ channels, there is at least one channel on which j can receive the message *provided* j tunes to the correct channel(s) in the appropriate timeslot(s). Let $\mathcal{A}_i^{k+1}$ be the set of $(k+1)$ lower-most channels in $\mathcal{A}_i$. Let $c_x \in \mathcal{A}_i^{k+1}$. In our algorithm, i transmits on c_x for $(k + 1)$ timeslots while the other nodes listen on their respective $(k + 1)$ lower-most channels, one channel at a time. Node i repeats the transmission behavior for every channel in $\mathcal{A}_i^{k+1}$; the remaining nodes listen on their $(k + 1)$ lower-most channels as before. Thus, each node transmits for $(k + 1)^2$ timeslots and a round of this algorithm consists of $\left(N \times (k + 1)^2\right)$ timeslots. For $k \ll M$, this is a drastic improvement over the NM timeslots required for *NDC* in [3].

Nodes with Multiple Receivers: Multiple transmissions may be supported by increasing the number of receivers at each node. Each node has r receivers ($r > 0$) and a single transmitter. At any given time, a node can be (i) transmitting on one channel, (ii) receiving on up to r channels or (iii) turned off. Simultaneous transmission and reception (even on different channels) at a node is not allowed. Collisions in a timeslot t are avoided by assigning a unique channel for each node that is transmitting in t. Note that a node cannot receive when it is transmitting. As a result, if nodes i and j are assigned to transmit simultaneously on channels c_i and c_j in a timeslot t, then there must be at least two other timeslots t_i and t_j for the nodes to communicate with each other. We use $\oplus_b$ to denote the modulo b addition: $x \oplus_b y = (x + y - 1) \mod b + 1$. Therefore, given the sequence of numbers 1, 2, $\cdots$, b, the subsequence of j consecutive numbers starting from number i (with wrap-around) is given by i, $i \oplus_b 1, \cdots, i \oplus_b (j - 1)$.

Let $G = \{1, 2, \cdots, N\}$ be the set of nodes. We assume that r divides both N and M. G is partitioned into $\frac{N}{r}$ groups $G_1, G_2, \cdots, G_{\frac{N}{r}}$ each of size r. Let $G_1 = \{1, 2, \cdots, r\}$, $G_2 = \{r + 1, r + 2, \cdots, 2r\}$, and so on be the groups. Each round of the algorithm consists of two sub-rounds. The first sub-round handles inter-group communications among nodes and consists of $\frac{N}{r}$ blocks. Each block consists of M timeslots. In block i, nodes in group G_i transmit and all other nodes listen. Specifically, in timeslot j of block i, r nodes in G_i, given by $(i - 1) * r + 1$, $(i - 1) * r + 2, \cdots, i * r$, transmit on r channels c_j, $c_{j \oplus_M 1}, \cdots, c_{j \oplus_M (r-1)}$, respectively. The remaining nodes receive on these r channels simultaneously. The second sub-round employs a divide-and-conquer approach to achieve intra-group communication. For the sake of brevity, we will only describe the second sub-round for the special case of $M = N = x$, such that each node has at least x receivers and $x = 2y$ for some integer y. The algorithm for this special case is used as a subroutine to achieve inter-group communication for general M, N and r in $O(\max(N, M) \log r)$ timeslots. Thus, the time complexity of a round of the algorithm for *NDC* is $O\left(\frac{N \times M}{r} + \max(N, M) \log r\right)$ [2].

Algorithm DAC: Consider a group of nodes $A = \{a_1, a_2, \cdots, a_x\}$ and a group of channels $B = \{b_1, b_2 \cdots, b_x\}$. The algorithm ensures that every node in A listens to every other node in A on all x channels in $O(x \log x)$ timeslots. Let DAC(A_1, B_1) and DAC(A_2, B_2) be two instances of the algorithm DAC. We use DAC(A_1, B_1) || DAC(A_2, B_2) to denote the algorithm obtained by running the two instances concurrently. To run the two instances concurrently, they should not interfere with each other. Thus, $A_1 \cap A_2 = \emptyset$ and $B_1 \cap B_2 = \emptyset$. Further, we also ensure that $|A_1| = |A_2|$. As a result, the running time of the resulting algorithm is same as that of the either instance. Likewise, we use DAC(A_1, B_1) $\circ$ DAC(A_2, B_2) to denote the algorithm obtained by running the two instances serially, one-by-one.

Let $A = A_1 \cup A_2$ be a partition of A, where $A_1 = \{a_1, a_2 \cdots, a_y\}$ and $A_2 = \{a_{y+1}, a_{y+2}, \cdots, a_{2y}\}$ (recall that $x = 2y$). The algorithm consists of three blocks. In the first block consisting of x timeslots, all nodes in A_1 transmit on all channels in B whereas nodes in A_2 only listen. Specifically, in timeslot i of the first block, nodes a_1, a_2, $\cdots$, a_y transmit on channels b_i, $b_{i \oplus_x 1}$, $\cdots$, $b_{i \oplus_x (y-1)}$, respectively. The second block is similar to the first block except that roles of A_1 and A_2 are reversed. Finally, in the third block, we recursively invoke the algorithm. Let $B = B_1 \cup B_2$ be a partition of B into two equal halves. The third block is given by $\big($DAC(A_1,B_1) || DAC(A_2,B_2)$\big)$ $\circ$ $\big($DAC(A_1,B_2) || DAC(A_2,B_1)$\big)$. Let $T(x)$ denote the running time of DAC(A, B). Then, $T(x) = 2x + 2T(x/2)$, which yields $T(x) = O(x \log x)$.

Discussion: Details of the algorithms and the proofs of correctness appear in [2]. In addition, we have also proposed solutions for other CR node models – (i) Limited channel divergence with multiple receivers and (ii) Nodes with multiple transceivers [2]. We have proved that $O(NM)$ timeslots is a lower bound for oblivious deterministic algorithms for *NDC* for the single transceiver model. (The proof is to appear in a future article.) Hence, we believe that the solutions presented here are close to optimal for models considered in this paper. Future work will focus on proving lower bounds and investigating adaptive deterministic algorithms that could potentially be faster than the oblivious algorithms proposed in this paper.

References

[1] R. W. Broderson, A. Wolisz, D. Cabric, S. M. Mishra, and D. Willkomm. *Corvus: A Cognitive Radio Approach for Usage of Virtual Unlicensed Spectrum.*

[2] S. Krishnamurthy, R. Chandrasekaran, S. Venkatesan, and N. Mittal. Algorithms for Node Discovery and Configuration in Cognitive Radio Networks. Technical Report UTDCS-23-06, The Univ. of Texas at Dallas, Richardson, TX, May 2006.

[3] S. Krishnamurthy, M. Thoppian, S. Kuppa, R. Chandrasekaran, S. Venkatesan, N. Mittal, and R. Prakash. Time-efficient Layer-2 Auto-configuration for Cognitive Radios. In *PDCS 2005*, Phoenix, AZ, November 2005.

[4] J. Mitola. *Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio.*

Provably Unbreakable Hyper-encryption Using Distributed Systems

Michael O. Rabin

DEAS Harvard University
Cambridge, MA 02138
rabin@deas.harvard.edu

Abstract. Encryption is a fundamental building block for computer and communications technologies. Existing encryption methods depend for their security on unproven assumptions. We propose a new model, the Limited Access model for enabling a simple and practical provably unbreakable encryption scheme. A voluntary distributed network of thousands of computers each maintain and update random pages, and act as Page Server Nodes. A Sender and Receiver share a random key K. They use K to randomly select the same PSNs and download the same random pages. These are employed in groups of say 30 pages to extract One Time Pads common to S and R. Under reasonable assumptions of an Adversary's inability to monitor all PSNs, and easy ways for S and R to evade monitoring while downloading pages, Hyper Encryption is clearly unbreakable. The system has been completely implemented.

Modern encryption methods depend for their security on assumptions concerning the intractability of various computational problems such as the factorization of large integers into prime factors or the computation of the discrete log function in large finite groups. Even if true, there are currently no methods for proving such assumptions. At the same time, even if these problems will be shown to be of super-polynomial complexity, there is steady progress in the development of practical algorithms for the solution of progressively larger instances of the problems in question. Thus there is no firm reason to believe that any of the encryptions in actual use is now safe, or an indication as to how long it will remain so. Furthermore, if and when the current intensive work on Quantum Computing will produce actual quantum computers, then the above encryptions will succumb to these machines.

At present there are three major proposals for producing provably unbreakable encryption methods. Quantum Cryptography employs properties of quantum mechanics to enable a Sender and Receiver to create common One Time Pads (OTPs) which are secret against any Adversary. The considerable research and development work as well as the funding invested in this effort are testimony to the need felt for an absolutely safe encryption technology. At present Quantum Cryptography systems are limited in range to a few tens of miles, are sensitive to noise or disturbance of the transmission medium, and require rather expensive special equipment.

The Limited Storage Model was proposed by U. Maurer. It postulates a public intensive source of random bits. An example would be a satellite or a system of satellites containing a Physical Random Number Generator (PRNG) beaming down a stream a of random numbers, say at the rate of 100GB/sec. S and R use a small shared key, and use those bits and the key to form OTPs which are

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 575–577, 2006.

subsequently employed in the usual manner to encrypt messages. The Limited Storage Model further postulates that for any Adversary or group of Adversaries it is technically or financially infeasible to store more than a fraction, say half, as many bits as there are in a. It was proved by Aumann, Rabin, and Ding and later by Dziembowski-Maurer, that under the Limited Storage Model assumptions, one can construct schemes producing OTPs which are essentially random even for a computationally unbounded (but storage limited) Adversary. The critique of the Limited Storage Model is three-fold. It requires a system of satellites, or other distribution methods, which are very expensive. The above rate of transmission for satellites is right now outside the available capabilities. More fundamentally, with the rapid decline of cost of storage it is not clear that storage is a limiting factor. For example, at a cost of $ 1 per GB, storing the above mentioned stream of bytes will cost about $ 3 Billion per year. And the cost of storage seems to go down very rapidly.

The Limited Access Model postulates a system comprising a multitude of sources of random bytes available to the Sender and Receiver. Each of these sources serves as a Page Server Node (PSN) and has a supply ofrandom pages. Sender and Receiver initially have a shared key K. Using K, Sender and Receiver asynchronously in time access the same PSNs and download the same random pages. The Limited Access assumption is that an Adversary cannot monitor or compromise more than a fraction of the PSNs while the Sender or Receiver download pages. After downloading sufficiently many pages, S and are make sure that they have the same pages by employing a Page Reconciliation Protocol. They now employ the common random pages according to a common scheme in groups of, say, 30 pages to extract an OTP from each group. Let us assume that the extraction method is simply taking the XOR of these pages. The common OTPs are used for encryption in the usual manner.

A crucially important point is that a Page Server Node sends out a requested random page at most twice, then destroys and replaces it by a new page. Opportunity knocks only twice!

Why is this scheme absolutely secure? Assume that we have 5,000 voluntary participants acting as PSNs. Assume that a, possibly distributed, Adversary can eavesdrop, monitor or corrupt (including by acting as imposter) no more than 1000 of these nodes. Thus the probability that in the random choice of the 30 PSNs from which a group of 30 pages are downloaded and XORed, all 30 pages will be known to the Adversary is smaller than (1/5)30, i.e., totally negligible. But if an Adversary misses even one page out of the 30 random pages that are XORed into an OTP then the OTP is completely random for him.

The send at most twice, then destroy policy, prevents a powerful Adversary from asking for a large number of pages from each of the PSNs and thereby gain copies of pages common to S and R. The worst that can happen is that, say, S will down load a page P from PSNi and the Adversary (or another user of Hyper-Encryption) has or will download the same page P from PSNi. When R now requests according to the key K the same page from PSNi, he will not get it. So R and S never have a page P in common if P was also downloaded by a third party. The only consequence of an Adversary's down-loading from too many PSNs is denial of service to the legitimate users of the system. This is a problem for any server system and there are ways of dealing with this type of attack.

What if an Adversary eavesdrops onto the Sender and or Receiver while they are downloading pages from PSNs. Well, S and R can go to an Internet caf or one of those establishments allowing a customer to obtain an Internet connection. They can use a device that does not identify them and download thousands of pages from PSNs within a short time. The salient point is that S and R need not time-synchronize their access to the PSNs. Once S and R have common OTPs, they can securely communicate from their fixed known locations with immunity against eavesdropping or code breaking.

The initial key K is continually extended and updated by S and R using common One Time Pads. Each pair of random words from K is used to select a PSN and a page from that PSN only once and then discarded. This is essential for the absolute security of Hyper Encryption.

With all these provisions Hyper Encryption in the Limited Access Model also provides Ever Lasting Secrecy. Let us make a worst case assumption that the initial common key K or its later extensions were lost or stolen after their use to collect common random pages from PSNs. Those pages are not available any more as a result of the send only twice and destroy policy. Thus the extracted OTPs used to encrypt messages cannot be reconstructed and the encryption is valid in perpetuity. By contrast, all the existing public or private key encryption methods are vulnerable to the retroactive decryption attack if the key is lost or algorithms come up that break the encryption algorithm.

We shall also describe an additional scheme based on the use of search engines for the generation of OTPs and of unbreakable encryption.

Our systems were fully coded in Java for distribution as freeware amongst interested users. All the protocols described below are running in the background on the participating computers and impose negligible computational and storage overheads on the host computer.

Time, Clocks, and the Ordering of My Ideas About Distributed Systems

Leslie Lamport

Microsoft Corporation
1065 La Avenida
Mountain View, CA 94043
U.S.A.
lamport@microsoft.com

Abstract. A guided tour through the labyrinth of my thoughts, from the Bakery Algorithm to arbiter-free marked graphs. This exercise in egotism is by invitation of the DISC 20th Anniversary Committee. I take no responsibility for the choice of topic.

S. Dolev (Ed.): DISC 2006, LNCS 4167, p. 578, 2006.

My Early Days in Distributed Computing Theory: 1979–1982

Nancy Lynch

CSAIL, MIT
Cambridge, MA 02139
U.S.A.
lynch@theory.csail.mit.edu

Abstract. I first became involved in Distributed Computing Theory around 1978 or 1979, as a new professor at Georgia Tech. Looking back at my first few years in the field, approximately 1979-1982, I see that they were tremendously exciting, productive, and fun. I collaborated with, and learned from, many leaders of the field, including Mike Fischer, Jim Burns, Michael Merritt, Gary Peterson, Danny Dolev, and Leslie Lamport.

Results that emerged during that time included space lower bounds for mutual exclusion; definition of the k-exclusion problem, with associated lower bounds and algorithms; the Burns-Lynch lower bound on the number of registers needed for mutual exclusion; fast network-wide resource allocation algorithms; the Lynch-Fischer semantic model for distributed systems (a precursor to I/O automata); early work on proof techniques for distributed algorithms; lower bounds on the number of rounds for Byzantine agreement; definition of the approximate agreement problem and associated algorithms; and finally, the Fischer-Lynch-Paterson impossibility result for consensus.

In this talk, I will review this early work, trying to explain how we were thinking at the time, and how the ideas in these projects influenced later work.

S. Dolev (Ed.): DISC 2006, LNCS 4167, p. 579, 2006.

Panel on the Contributions of the DISC Community to Distributed Computing: A Historical Perspective

Eli Gafni[1], Jan van Leeuwen[2], Michel Raynal[3], Nicola Santoro[4], and Shmuel Zaks[5]

[1] UCLA, CA, USA
eli@cs.ucla.edu
[2] Utrecht University, The Netherlands
jan@cs.uu.nl
[3] IRISA, Université de Rennes, France
raynal@irisa.fr
[4] Carleton University, Ottawa, Canada
santoro@scs.carleton.ca
[5] The Technion, Haifa, Israel
zaks@cs.technion.ac.il

This panel discussed the contributions of the DISC community to distributed computing. The panelists (Eli Gafni, Jan van Leeuwen, Nicola Santoro, Shmuel Zaks) and the moderator (Michel Raynal) were the members of the program committee of the second DISC (called WDAG at that time), held in Amsterdam.

At the very beginning, WDAG was centered mainly on distributed algorithms on graphs. Subsequently, while keeping its main focus on distributed algorithms, WDAG evolved and adopted a more general view of the research area, changed its name and became DISC. In a continuous manner, new topics have always appeared in the DISC call for papers (and also in accepted papers!). These include ubiquitous computing, cryptography, autonomic computing to name only a few. The scientific DISC contributions are numerous. They are on distributed computing models, algorithm design, complexity, possibility/impossibility results, distributed computability, lower bounds, etc. The panel reviewed the status of many contributions to network protocol design and to the understanding of distributed computing in general. It also discussed the possible ways in which DISC may evolve in the future.

S. Dolev (Ed.): DISC 2006, LNCS 4167, p. 580, 2006.

DISC at Its 20th Anniversary: Past, Present and Future

Michel Raynal[1], Sam Toueg[2], and Shmuel Zaks[3]

[1] IRISA, Campus de Beaulieu, 35042 Rennes, France
[2] Department of Computer Science, University of Toronto, Toronto, Canada
[3] Department of Computer Science, Technion, Haifa, Israel

Prologue

DISC 2006 marks the 20th anniversary of the DISC conferences. We list below the special events that took place during DISC 2006, together with some information and perspectives on the past and future of DISC.

Present: Special 20th Anniversary Events

The celebration of the 20th anniversary of DISC consisted in three main events: invited talks by three of the brightest figures of the distributed computing community, and a panel involving all the people who were at the very beginning of DISC:

- An invited talk "*Time, clocks and the ordering of my ideas about distributed systems*" by Leslie Lamport.
- An invited talk "*My early days in distributed computing theory: 1979-1982*" by Nancy Lynch.
- An invited talk "*Provably unbreakable hyper-encryption using distributed systems*" by Michael Rabin.
- A panel on "*The contributions of the WDAG/DISC community to distributed computing: a historical perspective*" by Eli Gafni, Michel Raynal, Nicola Santoro, Jan van Leeuwen and Shmuel Zaks (who were the PC members of the second WDAG, Amsterdam, 1987).

Past: A Short History

The *Workshop on Distributed Algorithms on Graphs* (WDAG) was initiated by Eli Gafni, Nicola Santoro and Jan van Leeuwen in 1985. It was intended to provide a forum for researchers and other interested parties to present and discuss recent results and trends in the design and analysis of distributed algorithms on communication networks and graphs.

Then, more than 10 years later, the acronym WDAG was changed to DISC (the international symposium on DIStributed Computing). This change was made to reflect

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 581–583, 2006.

the expansion from a workshop to a symposium as well as the expansion of the research areas of interest. So, following 11 successful WDAGs, DISC'98 was the 12th in the series.

Since 1996 WDAG/DISC has been managed by a Steering Committee consisting of some of the most experienced members of the distributed computing community. The main role of this committee is to provide guidance and leadership to ensure the continuing success of this conference. To do so, the committee oversees the continuous evolution of the symposium's research areas of interest, it forges ties with other related conferences and workshops, and it also maintains contact with Springer-Verlag and other professional or scientific sponsoring organizations (such as EATCS). The structure and rules of the DISC Steering Committee, which were composed by Sam Toueg and Shmuel Zaks, can be found at *http://www.disc-conference.org*, along with information about the previous DISC conferences.

The location, program chairs, and proceedings of the past 20 WDAG/DISC meetings are summarized in Table 1, and the Steering Committee Chairs are listed in Table 2.

Table 1. The past Wdag/Disc

Year	Location	Program Chair(s)	Proceedings
1985	Ottawa	N. Santoro and J. van Leeuwen	Carleton Scientific
1987	Amsterdam	J. van Leeuwen	LNCS 312
1989	Nice	J.-Cl. Bermond and M. Raynal	LNCS 392
1990	Bari	N. Santoro and J. van Leeuwen	LNCS 486
1991	Delphi	S. Toueg and P. Spirakis	LNCS 579
1992	Haifa	A. Segall and S. Zaks	LNCS 647
1993	Lausanne	A. Schiper	LNCS 725
1994	Terschelling	G. Tel and P. Vitányi	LNCS 857
1995	Le Mont-Saint-Michel	J.-M. Hélary and M. Raynal	LNCS 972
1996	Bologna	Ö. Babaoglŭ and K. Marzullo	LNCS 1151
1997	Saarbrücken	M. Mavronicolas and Ph. Tsigas	LNCS 1320
1998	Andros	S. Kutten	LNCS 1499
1999	Bratislava	P. Jayanti	LNCS 1693
2000	Toledo	M. Herlihy	LNCS 1914
2001	Lisbon	J. Welch	LNCS 2180
2002	Toulouse	D. Malkhi	LNCS 2508
2003	Sorrento	F.E. Fich	LNCS 2848
2004	Amsterdam	R. Guerraoui	LNCS 3274
2005	Cracow	P. Fraigniaud	LNCS 3724
2006	Stockholm	S. Dolev	These proceedings

Table 2. Steering committee chairs

Period	1996-1998	1998-2000	2000-2002	2002-2004	2004-2006	2006-2008
SC chair	Sam Toueg	Shmuel Zaks	André Schiper	Michel Raynal	Alex Shvartsman	Paul Vitányi

Epilogue, and Future

Together with the whole DISC community, we congratulate DISC for its 20th anniversary. We feel proud to have taken part in this important and successful activity of our research community, and are confident that DISC will continue to play a central role in years to come.

We wish to thank all those who contributed over the years to the success of DISC. Each played an essential role, and each forms a vital link in the DISC chain:

- The local organizers, and their teams, who did everything to ensure a smooth and successful conference,
- The program committee chairs, program committee members, and external referees, who ensured the high academic level of the conference,
- The participants of the WDAG and DISC conferences,
- The steering committee members,
- The sponsor organizations, for their generous support over the years,
  and - last but not least -
- All the members of the distributed computing community who submitted papers to WDAG and DISC.

We are confident that the DISC community will continue to play a central role within the distributed computing and communication networks research areas for many years to come.

HAPPY ANNIVERSARY TO DISC!

This photo is from DISC 2005 in Cracow, Poland, and was taken during the banquet at *Wierzynek 1364* restaurant (one of the oldest restaurants in Europe). It shows the first five chairs of the DISC steering committee (from left to right: Shmuel Zaks, Alex Shvartsman, Michel Raynal, André Schiper and Sam Toueg).

Author Index

ERRATUM

DISC at Its 20th Anniversary: Past, Present and Future

Michel Raynal[1], Sam Toueg[2], and Shmuel Zaks[3]

[1] IRISA, Campus de Beaulieu, 35042 Rennes, France
[2] Department of Computer Science, University of Toronto, Toronto, Canada
[3] Department of Computer Science, Technion, Haifa, Israel

S. Dolev (Ed.): DISC 2006, LNCS 4167, pp. 581–583, 2006.

DOI 10.1007/11864219_55

An error has been found in the above volume

The Invited Talks and the Anniversary Paper which are missing in the printed volume have been added to the online version.

The online version for this chapter can be found at
http://dx.doi.org/10.1007/11864219_55