Theodore P. Baker
Alain Bui
Sébastien Tixeuil (Eds.)

LNCS 5401

Principles of Distributed Systems

12th International Conference, OPODIS 2008
Luxor, Egypt, December 2008
Proceedings

Springer

Lecture Notes in Computer Science 5401

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

Theodore P. Baker   Alain Bui
Sébastien Tixeuil (Eds.)

Principles of Distributed Systems

12th International Conference, OPODIS 2008
Luxor, Egypt, December 15-18, 2008
Proceedings

Springer

Volume Editors

Theodore P. Baker
Florida State University
Department of Computer Science
207A Love Building, Tallahassee, FL 32306-4530, USA
E-mail: baker@cs.fsu.edu

Alain Bui
Université de Versailles-St-Quentin-en-Yvelines
Laboratoire PRiSM
45, avenue des Etats-Unis, 78035 Versailles Cedex, France
E-mail: alain.bui@prism.uvsq.fr

Sébastien Tixeuil
LIP6 & INRIA Grand Large
Université Pierre et Marie Curie - Paris 6
104 avenue du Président Kennedy, 75016 Paris, France
E-mail: Sebastien.Tixeuil@lip6.fr

Library of Congress Control Number: 2008940868

CR Subject Classification (1998): C.2.4, C.1.4, C.2.1, D.1.3, D.4.2, E.1, H.2.4

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

ISSN 0302-9743
ISBN-10 3-540-92220-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-92220-9 Springer Berlin Heidelberg New York

springer.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12582457 06/3180 5 4 3 2 1 0

Preface

This volume contains the 30 regular papers, the 11 short papers and the abstracts of two invited keynotes that were presented at the 12th International Conference on Principles of Distributed Systems (OPODIS) held during December 15–18, 2008 in Luxor, Egypt.

OPODIS is a yearly selective international forum for researchers and practitioners in design and development of distributed systems.

This year, we received 102 submissions from 28 countries. Each submission was carefully reviewed by three to six Program Committee members with the help of external reviewers, with 30 regular papers and 11 short papers being selected. The overall quality of submissions was excellent and there were many papers that had to be rejected because of organization constraints yet deserved to be published. The two invited keynotes dealt with hot topics in distributed systems: "The Next 700 BFT Protocols" by Rachid Guerraoui and "On Replication of Software Transactional Memories" by Luis Rodriguez.

On behalf of the Program Committee, we would like to thank all authors of submitted papers for their support. We also thank the members of the Steering Committee for their invaluable advice. We wish to express our appreciation to the Program Committee members and additional external reviewers for their tremendous effort and excellent reviews. We gratefully acknowledge the Organizing Committee members for their generous contribution to the success of the symposium. Special thanks go to Thibault Bernard for managing the conference publicity and technical organization. The paper submission and selection process was greatly eased by the EasyChair conference system (`http://www.easychair.org`). We wish to thank the EasyChair creators and maintainers for their commitment to the scientific community.

December 2008

Ted Baker
Sébastien Tixeuil
Alain Bui

Organization

OPODIS 2008 was organized by PRiSM (Université Versailles Saint-Quentin-en-Yvelines) and LIP6 (Université Pierre et Marie Curie).

General Chair

Alain Bui	University of Versailles St-Quentin-en-Yvelines, France

Program Co-chairs

Theodore P. Baker	Florida State University, USA
Sébastien Tixeuil	University of Pierre and Marie Curie, France

Program Committee

Bjorn Andersson	Polytechnic Institute of Porto, Portugal
James Anderson	University of North Carolina, USA
Alan Burns	University of York, UK
Andrea Clementi	University of Rome, Italy
Liliana Cucu	INPL Nancy, France
Shlomi Dolev	Ben-Gurion University, Israel
Khaled El Fakih	American University of Sharjah, UAE
Pascal Felber	University of Neuchatel, Switzerland
Paola Flocchini	University of Ottawa, Canada
Gerhard Fohler	University of Kaiserslautern, Germany
Felix Freiling	University of Mannheim, Germany
Mohamed Gouda	University of Texas, USA
Fabiola Greve	UFBA, Brazil
Isabelle Guerin-Lassous	University of Lyon 1, France
Ted Herman	University of Iowa, USA
Anne-Marie Kermarrec	INRIA, France
Rastislav Kralovic	Comenius University, Slovakia
Emmanuelle Lebhar	CNRS/University of Paris 7, France
Jane W.S. Liu	Academia Sinica Taipei, Taiwan
Steve Liu	Texas A&M University, USA
Toshimitsu Masuzawa	University of Osaka, Japan
Rolf H. Möhring	TU Berlin, Germany
Bernard Mans	Macquarie University, Australia
Maged Michael	IBM, USA
Mohamed Mosbah	University of Bordeaux 1, France

Marina Papatriantafilou	Chalmers University of Technology, Sweden
Boaz Patt-Shamir	Tel Aviv University, Israel
Raj Rajkumar	Carnegie Mellon University, USA
Sergio Rajsbaum	UNAM, Mexico
Andre Schiper	EPFL, Switzerland
Sam Toueg	University of Toronto, Canada
Eduardo Tovar	Polytechnic Institute of Porto, Portugal
Koichi Wada	Nogoya Institute of Technology, Japan

Organizing Committee

Thibault Bernard	University of Reims Champagne-Ardenne, France
Celine Butelle	EPHE, France

Publicity Chair

Thibault Bernard	University of Reims Champagne-Ardenne, France

Steering Committee

Alain Bui	University of Versailles St-Quentin-en-Yvelines, France
Marc Bui	EPHE, France
Hacene Fouchal	University of Antilles-Guyane, France
Roberto Gomez	ITESM-CEM, Mexico
Nicola Santoro	Carleton University, Canada
Philippas Tsigas	Chalmers University of Technology, Sweden

Referees

H.B. Acharya
Amitanand Aiyer
Mario Alves
James Anderson
Bjorn Andersson
Hagit Attiya
Rida Bazzi
Muli Ben-Yehuda
Alysson Bessani
Gaurav Bhatia
Konstantinos Bletsas
Bjoern Brandenburg
Alan Burns
John Calandrino
Pierre Castéran
Daniel Cederman
Keren Censor
Jérémie Chalopin
Claude Chaudet
Yong Hoon Choi
Andrea Clementi
Reuven Cohen
Alex Cornejo
Roberto Cortinas
Pilu Crescenzi
Liliana Cucu
Shantanu Das
Emiliano De Cristofaro
Gianluca De Marco
Carole Delporte
UmaMaheswari Devi
Shlomi Dolev
Pu Duan
Partha Dutta
Khaled El-fakih
Yuval Emek

Hugues Fauconnier
Pascal Felber
Paola Flocchini
Gerhard Fohler
Pierre Fraignaud
Felix Freiling
Zhang Fu
Shelby Funk
Emanuele G. Fusco
Giorgos Georgiadis
Seth Gilbert
Emmanuel Godard
Joel Goossens
Mohamed Gouda
Maria Gradinariu Potop-Butucaru
Vincent Gramoli
Fabiola Greve
Damas Gruska
Isabelle Guerin-Lassous
Phuong Ha Hoai
Ahmed Hadj Kacem
Elyes-Ben Hamida
Danny Hendler
Thomas Herault
Ted Herman
Daniel Hirschkoff
Akira Idoue
Nobuhiro Inuzuka
Taisuke Izumi
Tomoko Izumi
Katia Jaffres-Runser
Prasad Jayanti
Arshad Jhumka
Mohamed Jmaiel
Hirotsugu Kakugawa
Arvind Kandhalu
Yoshiaki Katayama
Branislav Katreniak
Anne-Marie Kermarrec
Ralf Klasing
Boris Koldehofe
Anis Koubaa
Darek Kowalski
Rastislav Kralovic
Evangelos Kranakis
Ioannis Krontiris
Petr Kuznetsov
Mikel Larrea
Erwan Le Merrer
Emmanuelle Lebhar
Hennadiy Leontyev
Xu Li
George Lima
Jane Liu
Steve Liu
Hong Lu
Victor Luchangco
Weiqin Ma
Bernard Mans
Soumaya Marzouk
Toshimitsu Masuzawa
Nicole Megow
Maged Michael
Luis Miguel Pinho
Rolf Möhring
Mohamed Mosbah
Heinrich Moser
Achour Mostefaoui
Junya Nakamura
Alfredo Navarra
Gen Nishikawa
Nicolas Nisse
Luis Nogueira
Koji Okamura
Fukuhito Ooshita
Marina Papatriantafilou
Dana Pardubska
Boaz Patt-Shamir
Andrzej Pelc
David Peleg
Nuno Pereira
Tomas Plachetka
Shashi Prabh
Giuseppe Prencipe
Shi Pu
Raj Rajkumar
Sergio Rajsbaum
Dror Rawitz
Tahiry Razafindralambo
Etienne Riviere
Gianluca Rossi
Anthony Rowe
Nicola Santoro
Gabriel Scalosub
Elad Schiller
Andre Schiper
Nicolas Schiper
Ramon Serna Oliver
Alexander Shvartsman
Riccardo Silvestri
Françoise Simonot-Lion
Alex Slivkins
Jason Smith
Kannan Srinathan
Sebastian Stiller
David Stotts
Weihua Sun
Høakan Sundell
Cheng-Chung Tan
Andreas Tielmann
Sam Toueg
Eduardo Tovar
Corentin Travers
Frederic Tronel
Rémi Vannier
Jan Vitek
Roman Vitenberg
Koichi Wada
Timo Warns
Andreas Wiese
Yu Wu
Zhaoyan Xu
Hirozumi Yamaguchi
Yukiko Yamauchi
Keiichi Yasumoto

Table of Contents

Invited Talks

Regular Papers

The Next 700 BFT Protocols
(Invited Talk)

Rachid Guerraoui

EPFL LPD, Bat INR 310, Station 14, 1015 Lausanne, Switzerland

Byzantine fault-tolerant state machine replication (BFT) has reached a reasonable level of maturity as an appealing, software-based technique, to building robust distributed services with commodity hardware. The current tendency however is to implement a new BFT protocol from scratch for each new application and network environment. This is notoriously difficult. Modern BFT protocols require each more than 20.000 lines of sophisticated C code and proving their correctness involves an entire PhD. Maintainning and testing each new protocol seems just impossible.

This talk will present a candidate abstraction, named ABSTRACT (Abortable State Machine Replication), to remedy this situation. A BFT protocol is viewed as a, possibly dynamic, composition of instances of ABSTRACT, each instance developed and analyzed independently. A new effective BFT protocol can be developped by adding less than 10% of code to an existing one. Correctness proofs become at human reach and even model checking techniques can be envisaged. To illustrate the ABSTRACT approach, we describe a new BFT protocol we name Aliph: the first of a hopefully long series of effective yet modular BFT protocols. The Aliph protocol has a peak throughput that outperforms those of all BFT protocols we know of by 300% and a best case latency that is less than 30% of that of state of the art BFT protocols.

This is joint work with Dr V. Quema (CNRS) and Dr M. Vukolic (IBM).

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, p. 1, 2008.

On Replication of Software Transactional Memories
(Invited Talk)

Luis Rodrigues

INESC-ID/IST

joint work with:
Paolo Romano and Nuno Carvalho
INESC-ID

Extended Abstract

Software Transactional Memory (STM) systems have garnered considerable interest of late due to the recent architectural trend that has led to the pervasive adoption of multi-core CPUs. STMs represent an attractive solution to spare programmers from the pitfalls of conventional explicit lock-based thread synchronization, leveraging on concurrency-control concepts used for decades by the database community to simplify the mainstream parallel programming [1].

As STM systems are beginning to penetrate into the realms of enterprise systems [2,3] and to be faced with the high availability and scalability requirements proper of production environments, it is rather natural to foresee the emergence of replication solutions specifically tailored to enhance the dependability and the performance of STM systems. Also, since STM and Database Management Systems (DBMS) share the key notion of transaction, it might appear that the state of the art database replication schemes e.g. [4,5,6,7] represent natural candidates to support STM replication as well.

In this talk, we will first contrast, from a replication oriented perspective, the workload characteristics of two standard benchmarks for STM and DBMS, namely TPC-W [8] and STBench7 [9]. This will allow us to uncover several pitfalls related to the adoption of conventional database replication techniques in the context of STM systems.

At the light of such analysis, we will then discuss promising research directions we are currently pursuing in order to develop high performance replication strategies able to fit the unique characteristics of the STM.

In particular, we will present one of our most recent results in this area which not only tackles some key issues characterizing STM replication, but actually represents a valuable tool for the replication of generic services: the Weak Mutual Exclusion (WME) abstraction. Unlike the classical Mutual Exclusion problem (ME), which regulates the concurrent access to a single and indivisible shared resource, the WME abstraction ensures mutual exclusion in the access to a shared resource that appears as single and indivisible only at a logical level, while instead being physically replicated for both fault-tolerance and scalability purposes.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 2–4, 2008.

Differently from ME, which is well known to be solvable only in the presence of very constraining synchrony assumptions [10] (essentially exclusively in synchronous systems), we will show that WME is solvable in an asynchronous system using an eventually perfect failure detector, $\diamondsuit P$, and prove that $\diamondsuit P$ is actually the weakest failure detector for solving the WME problem. These results imply, unlike ME, WME is solvable in partially synchronous systems, (i.e. systems in which the bounds on communication latency and relative process speed either exist but are unknown or are known but are only guaranteed to hold starting at some unknown time) which are widely recognized as a realistic model for large scale distributed systems [11,12].

However, this is not the only element contributing to the pragmatical relevance of the WME abstraction. In fact, the reliance on the WME abstraction, as a mean for regulating the concurrent access to a replicated resource, also provides the two following important practical benefits:

Robustness: pessimistic concurrency control is widely used in commercial off the shelf systems, e.g. DBMSs and operating systems, because of its robustness and predictability in presence of conflict intensive workloads. The WME abstraction lays a bridge between these proven contention management techniques and replica control schemes. Analogously to centralized lock based concurrency control, WME reveals particularly useful in the context of conflict-sensitive applications, such as STMs or interactive systems, where it may be preferable to bridle concurrency rather than incurring the costs of application level conflicts, such as transactions abort or re-submission of user inputs.

Performance: the WME abstraction ensures that users issue operations on the replicated shared resource in a sequential manner. Interestingly, it has been shown that, in such a scenario, it is possible to sensibly boost the performance of lower level abstractions [13,14], such as consensus or atomic broadcast, which are typically used as building blocks of modern replica control schemes and which often represent, like in typical STM workloads, the performance bottleneck of the whole system.

References

1. Adl-Tabatabai, A.R., Kozyrakis, C., Saha, B.: Unlocking concurrency. ACM Queue 4, 24–33 (2007)
2. Cachopo, J.: Development of Rich Domain Models with Atomic Actions. PhD thesis, Instituto Superior Técnico/Universidade Técnica de Lisboa (2007)
3. Carvalho, N., Cachopo, J., Rodrigues, L., Rito Silva, A.: Versioned transactional shared memory for the FénixEDU web application. In: Proc. of the Second Workshop on Dependable Distributed Data Management (in conjunction with Eurosys 2008), Glasgow, Scotland. ACM, New York (2008)
4. Agrawal, D., Alonso, G., Abbadi, A.E., Stanoi, I.: Exploiting atomic broadcast in replicated databases (extended abstract). In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 496–503. Springer, Heidelberg (1997)

5. Cecchet, E., Marguerite, J., Zwaenepole, W.: C-JDBC: flexible database clustering middleware. In: Proc. of the USENIX Annual Technical Conference, Berkeley, CA, USA, p. 26. USENIX Association (2004)
6. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable replication in database clusters. In: Proc. of the 14th International Conference on Distributed Computing, London, UK, pp. 315–329. Springer, Heidelberg (2000)
7. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14, 71–98 (2003)
8. Transaction Processing Performance Council: TPC Benchmark$^{\text{TM}}$ W, Standard Specification, Version 1.8. Transaction Processing Perfomance Council (2002)
9. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 315–324 (2007)
10. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. J. Parallel Distrib. Comput. 65, 492–505 (2005)
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35, 288–323 (1988)
12. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems 10, 642–657 (1999)
13. Brasileiro, F.V., Greve, F., Mostéfaoui, A., Raynal, M.: Consensus in one communication step. In: Proc. of the International Conference on Parallel Computing Technologies, pp. 42–50 (2001)
14. Lamport, L.: Fast paxos. Distributed Computing 9, 79–103 (2006)

Write Markers for Probabilistic Quorum Systems

Michael G. Merideth[1] and Michael K. Reiter[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] University of North Carolina, Chapel Hill, NC, USA

Abstract. Probabilistic quorum systems can tolerate a larger fraction of faults than can traditional (strict) quorum systems, while guaranteeing consistency with an arbitrarily high probability for a system with enough replicas. However, the masking and opaque types of probabilistic quorum systems are hampered in that their optimal load—a best-case measure of the work done by the busiest replica, and an indicator of scalability—is little better than that of strict quorum systems. In this paper we present a variant of probabilistic quorum systems that uses *write markers* in order to limit the extent to which Byzantine-faulty servers act together. Our masking and opaque probabilistic quorum systems have asymptotically better load than the bounds proven for previous masking and opaque quorum systems. Moreover, the new masking and opaque probabilistic quorum systems can tolerate an additional 24% and 17% of faulty replicas, respectively, compared with probabilistic quorum systems without write markers.

1 Introduction

Given a universe U of servers, a *quorum system* over U is a collection $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ such that each $Q_i \subseteq U$ and

$$|Q \cap Q'| > 0 \tag{1}$$

for all $Q, Q' \in \mathcal{Q}$. Each Q_i is called a *quorum*. The intersection property (1) makes quorums a useful primitive for coordinating actions in a distributed system. For example, if clients perform writes at a quorum of servers, then a client who reads from a quorum will observe the last written value. Because of their utility in such applications, quorums have a long history in distributed computing.

In systems that may suffer Byzantine faults [1], the intersection property (1) is typically not adequate as a mechanism to enable consistent data access. Because (1) requires only that the intersection of quorums be non-empty, it could be that two quorums intersect only in a single server, for example. In a system in which up to $b > 0$ servers might suffer Byzantine faults, this single server might be faulty and consequently, could fail to convey the last written value to a reader, for example.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 5–21, 2008.

For this reason, Malkhi and Reiter [2] proposed various ways of strengthening the intersection property (1) so as to enable quorums to be used in Byzantine environments. For example, an alternative to (1) is

$$|Q \cap Q' \setminus B| > |Q' \cap B| \tag{2}$$

for all $Q, Q' \in \mathcal{Q}$, where B is the (unknown) set of all (up to b) servers that are faulty. In other words, the intersection of any two quorums contains more non-faulty servers than the faulty ones in either quorum. As such, the responses from these non-faulty servers will outnumber those from faulty ones. These quorum systems are called *masking* systems.

Opaque quorum systems, have an even more stringent requirement as an alternative to (1):

$$|Q \cap Q' \setminus B| > |(Q' \cap B) \cup (Q' \setminus Q)| \tag{3}$$

for all $Q, Q' \in \mathcal{Q}$. In other words, the number of correct servers in the intersection of Q and Q' (i.e., $|Q \cap Q' \setminus B|$) exceeds the number of faulty servers in Q' (i.e., $|Q' \cap B|$) together with the number of servers in Q' but not Q. The rationale for this property can be seen by considering the servers in Q' but not Q as "outdated", in the sense that if Q was used to perform an update to the system, then those servers in $Q' \setminus Q$ are unaware of the update. As such, if the faulty servers in Q' behave as the outdated ones do, their behavior (i.e., their responses) will dominate that from the correct servers in the intersection ($Q \cap Q' \setminus B$) unless (3) holds.

The increasingly stringent properties of Byzantine quorum systems come with costs in terms of the smallest system sizes that can be supported while tolerating a number b of faults [2]. This implies that a system with a fixed number of servers can tolerate fewer faults when the property is more stringent as seen in Table 1, which refers to the quorums just discussed as *strict*. Table 1 also shows the negative impact on the ability of the system to disperse load amongst the replicas, as discussed next.

Naor and Wool [3] introduced the notion of an *access strategy* by which clients select quorums to access. An access strategy $p : \mathcal{Q} \rightarrow [0, 1]$ is simply a probability distribution on quorums, i.e., $\sum_{Q \in \mathcal{Q}} p(Q) = 1$. Intuitively, when a client accesses the system, it does so at a quorum selected randomly according to the distribution p.

The formalization of an access strategy is useful as a tool for discussing the load dispersing properties of quorums. The *load* [3] of a quorum system, $\mathcal{L}(\mathcal{Q})$, is the probability with which the busiest server is accessed in a client access, under the best possible access strategy p. As listed in Table 1, tight lower bounds have been proven for the load of each type of strict Byzantine quorum system. The load for opaque quorum systems is particularly unfortunate—systems that utilize opaque quorum systems cannot effectively disperse processing load across more servers (i.e., by increasing n) because the load is at least a constant. Such Byzantine quorum systems are used by many modern Byzantine-fault-tolerant protocols, e.g., [4,5,6,7,8,9] in order to tolerate the arbitrary failure of a subset of their replicas. As such, circumventing the bounds is an important topic.

One way to circumvent these bounds is with *probabilistic quorum systems.* Probabilistic quorum systems relax the quorum intersection properties, asking them to hold only with high probability. More specifically, they relax (2) or (3), for example, to hold only with probability $1 - \epsilon$ (for ϵ, a small constant), where probabilities are taken with respect to the selection of quorums according to an access strategy p [10,11]. This technique yields masking quorum constructions tolerating $b < 2.62/n$ and opaque quorum constructions tolerating $b < 3.15/n$ as seen in Table 1. These bounds hold in the sense that for any $\epsilon > 0$ there is an n_0 such that for all $n > n_0$, the required intersection property ((2) or (3) for masking and opaque quorum systems, respectively) holds with probability at least $1 - \epsilon$. Unfortunately, probabilistic quorum systems alone do not materially improve the load of Byzantine quorum systems.

In this paper, we present an additional modification, *write markers*, that improves on the bounds further. Intuitively, in each update access to a quorum of servers, a write marker is placed at the accessed servers in order to evidence the quorum used in that access. This write marker identifies the quorum used; as such, faulty servers not in this quorum cannot respond to subsequent quorum accesses as though they were.

As seen in Table 1, by using this method to constrain how faulty servers can collaborate, we show that probabilistic masking quorum systems with load $O(1/\sqrt{n})$ can be achieved, allowing the systems to disperse load independently of the value of b. Further, probabilistic opaque quorum systems with load $O(b/n)$ can be achieved, breaking the constant lower bound on load for opaque systems. Moreover, the resilience of probabilistic masking quorums can be improved an additional 24% to $b < n/2$, and the resilience of probabilistic opaque quorum systems can be improved an additional 17% to $b < n/2.62$.

Table 1. Improvements due to write markers (**Bold** entries are properties of particular constructions; others are lower bounds)

Non-Byzantine:	**load**		**faults**	
strict	$\Omega(1/\sqrt{n})$	[3]	$< n$	

Masking:	**load**		**faults**	
strict	$\Omega(\sqrt{b/n})$	[2]	$< n/4.00$	[12]
probabilistic	$\Omega(b/n)$	[10]	$< \mathbf{n/2.62}$	[11]
write markers	$\mathbf{O(1/\sqrt{n})}$	[here]	$< \mathbf{n/2.00}$	[here]

Opaque:	**load**		**faults**	
strict	$\geq 1/2$	[2]	$< n/5.00$	[2]
probabilistic	unproven		$< \mathbf{n/3.15}$	[11]
write markers	$\mathbf{O(b/n)}$	[here]	$< \mathbf{n/2.62}$	[here]

The probability of error in probabilistic quorums requires mechanisms to ensure that accesses are performed according to the required access strategy p if the clients cannot be trusted to do so. Therefore, we adapt one such mechanism, the access-restriction protocol of probabilistic opaque quorum systems [11], to accomodate write markers. Thus, as a side benefit, our implementation forces faulty clients to follow the access strategy. With this, we provide a protocol to implement write markers that tolerates Byzantine clients.

Our primary contributions are (i) the identification and analysis of the benefits of write markers; and (ii) a proposed implementation of write markers that handles the complexities of tolerating Byzantine clients. Our analysis yields the following results:

Masking Quorums: We show that the use of write markers allows probabilistic masking quorum systems to tolerate up to $b < n/2$ faults when quorums are of size $\Omega(\sqrt{n})$. Setting all quorums to size $\rho\sqrt{n}$ for some constant ρ, we achieve a load that is asymptotically optimal for any quorum system, i.e., $\rho\sqrt{n}/n = O(1/\sqrt{n})$ [3].

This represents an improvement in load and the number of faults that can be tolerated. Probabilistic masking quorums without write markers can tolerate up to $b < n/2.62$ faults [11] and achieve load no better than $\Omega(b/n)$ [10]. In addition, the maximum number of faults that can be tolerated is tied to the size of quorums [10]. Thus, without write markers, achieving optimal load requires tolerating fewer faults. Strict masking quorum systems can tolerate (only) up to $b < n/4$ faults [2] and can achieve load $\Omega(\sqrt{b/n})$ [12].

Opaque Quorums: We show that the use of write markers allows probabilistic opaque quorum systems to tolerate up to $b < n/2.62$ faults. We present a construction with load $O(b/n)$ when $b = \Omega(\sqrt{n})$, thereby breaking the constant lower bound of 1/2 on the load of strict opaque quorum systems [2]. Moreover, if $b = O(\sqrt{n})$, we can set all quorums to size $\rho\sqrt{n}$ for some constant ρ, in order to achieve a load that is asymptotically optimal for any quorum system, i.e., $\rho\sqrt{n}/n = O(1/\sqrt{n})$ [3].

This represents an improvement in load and the number of faults that can be tolerated. Probabilistic opaque quorum systems without write markers can tolerate (only) up to $b < n/3.15$ faults [11]. Strict opaque quorum systems can tolerate (only) up to $b < n/5$ faults [2]; these quorum systems can do no better than constant load even if $b = 0$ [2].

2 Definitions and System Model

We assume a system with a set U of servers, $|U| = n$, and an arbitrary but bounded number of clients. Clients and servers can fail arbitrarily (i.e., Byzantine faults [1]). We assume that up to b servers can fail, and denote the set of faulty servers by B, where $B \subseteq U$. Any number of clients can fail. Failures are permanent. Clients and servers that do not fail are said to be *non-faulty*. We allow that faulty clients and servers may collude, and so we assume that faulty clients and servers all know the membership of B (although non-faulty clients and servers do not). However, for our implementation of write markers, as is typical for many Byzantine-fault-tolerant protocols (c.f., [4,5,6,9]), we assume that faulty clients and servers are computationally bound such that they cannot subvert standard cryptographic primitives such as digital signatures.

Communication. Write markers require no communication assumptions beyond those of the probabilistic quorums for which they are used. For completeness, we summarize the model of [11], which is common to prior works in probabilistic [10] and signed [13] quorum systems: we assume that each non-faulty client can successfully communicate with each non-faulty server with high probability, and hence with all non-faulty servers with roughly equal probability. This assumption is in place to ensure that the network does not significantly bias a non-faulty client's interactions with servers either toward faulty servers or toward different non-faulty servers than those with which another non-faulty client can interact. Put another way, we treat a server that can be reliably reached by none or only some non-faulty clients as a member of B.

Access set; access strategy; operation. We abstractly describe client operations as either *writes* that alter the state of the service or *reads* that do not. Informally, a non-faulty client performs a write to update the state of the service such that its value (or a later one) will be observed with high probability by any subsequent operation; a write thus successfully performed is called "established" (we define established more precisely below). A non-faulty client performs a read to obtain the value of the latest established write, where "latest" refers to the value of the most recent write preceding this read in a linearization [14] of the execution.

In the introduction, we discussed *access strategies* as probability distributions on quorums used for operations. For the remainder of the paper, we follow [11] in strictly generalizing the notion of access strategy to apply instead to *access sets* from which quorums are chosen. An access set is a set of servers from which the client selects a quorum. If the client is non-faulty, we assume that this selection is done uniformly at random. We adopt the access strategy that all access sets are chosen uniformly at random (even by faulty clients). In Section 4, we adapt a protocol to support write markers from one in [11] that approximately ensures this access strategy. Our analysis allows that access sets may be larger than quorums, though if access sets and quorums are of the same size, then our protocol effectively forces even faulty clients to select quorums uniformly at random as discussed in the introduction. In our analysis, all access sets used for reads and writes are of constant size a_{rd} and a_{wt} respectively. All quorums used for reads and writes are of constant size q_{rd} and q_{wt} respectively.

Candidate; conflicting; error probability; established; participant; qualified; vote. Each write yields a corresponding *candidate* at some number of servers. A candidate is an abstraction used in part to ensure that two distinct write operations are distinguishable from each other, even if the corresponding data values are the same. A candidate is *established* once it is accepted by all of the non-faulty servers in some write quorum of size q_{wt} within the write access set of size a_{wt}. In opaque quorum systems, property (3) anticipates that different non-faulty servers each may hold a different candidate due to concurrent writes. A candidate that is characterized by the property that a non-faulty server would accept either it or a given established candidate, but not both, is called a *conflicting* candidate. Two candidates may conflict because, e.g., they both bear the same timestamp. In either masking or opaque quorum systems, a faulty server may try to forge a conflicting candidate. No non-faulty server accepts two candidates that conflict with each other.

A server can try to *vote* for some candidate (e.g., by responding to a read operation) if the server is a *participant* in voting (i.e., if the server is a member of the client's read access set). However, a server becomes *qualified* to vote for a particular candidate only if the server is a member of the client's write access set selected for the write operation for which it votes. Non-faulty clients wait for responses from a read quorum of size q_{rd} contained in the read access set of size a_{rd}. An *error* is said to occur in a read operation when a non-faulty client fails to observe the latest value or a faulty client obtains sufficiently many votes for a conflicting value.[1] The *error probability* is the probability of this occurring.

Behavior of faulty clients. We assume that faulty clients seek to maximize the error probability by following specific strategies [11]. This is a conservative assumption; a client cannot increase—but may decrease—the probability of error by failing to follow these strategies. At a high level, the strategies are as follows: a faulty client, which may be completely restricted in its choices: (i) when establishing a candidate, writes the candidate to as few non-faulty servers as possible to minimize the probability that it is observed by a non-faulty client; and (ii) writes a conflicting candidate to as many servers as will accept it (i.e., faulty servers plus, in the case of an opaque quorum system, any non-faulty server that has not accepted the established candidate) in order to maximize the probability that it is observed.

3 Analysis of Write Markers

Intuitively, when a client submits a write, the candidate is associated with a write marker. We require that the following three properties are guaranteed by an implementation of write markers:

W1. Every candidate has a write marker that identifies the access set chosen for the write;
W2. A verifiable write marker implies that the access set was selected uniformly at random (i.e., according to the access strategy);
W3. Every non-faulty client can verify a write marker.

When considering a candidate, non-faulty clients and servers verify the candidate's write marker. Because of this verification, no non-faulty node will accept a vote for a candidate unless the issuing server is qualified to vote for the candidate. Since each write access set is chosen uniformly at random (W2), the faulty servers that can vote for a candidate, i.e., the faulty qualified servers, are therefore a random subset of the faulty servers.

[1] Faulty clients may be able to affect the system with such votes in some protocols [11].

Thus, write markers remove the advantage enjoyed by faulty servers in strict and traditional-probabilistic masking and opaque quorum systems, where any faulty participant can vote for any candidate—and therefore can collude to have a conflicting, potentially fabricated candidate chosen instead of an established candidate. This aspect of write markers is summarized in Table 2, which shows the impact of write markers in terms of the abilities of faulty and non-faulty servers to vote for a given candidate.

3.1 Consistency Constraints

Probabilistic quorum systems must satisfy constraints similar to those of strict quorum systems (e.g., (2), (3)), but only with probability $1 - \epsilon$. As with strict quorum systems, the purpose of these constraints is to guarantee that operations can be observed consistently in subsequent operations by receiving enough votes.

Table 2. Ability of a server to vote for a given candidate: $\bullet$ (traditional quorums); $\star$ (write markers)

Type of server	**Vote**
Non-faulty qualified participant	$\bullet$ $\star$
Faulty qualified participant	$\bullet$ $\star$
Non-faulty non-qualified participant	
Faulty non-qualified participant	$\bullet$

First, the constraints must ensure in expectation that a non-faulty client can observe the latest established candidate if such a candidate exists. Let Q_{rd} represent a read quorum chosen uniformly at random, i.e., a random variable, from a read access set itself chosen uniformly at random. (Think of this quorum as one used by a non-faulty client.) Let Q_{wt} represent a write quorum chosen by a potentially faulty client; Q_{wt} must be chosen from A_{wt}, an access set chosen uniformly at random. (Think of Q_{wt} as a quorum used for an established candidate.) Then the threshold r number of votes necessary to observe a value must be less than the expected number of non-faulty qualified participants, which is

$$\mathbb{E}\left[\left|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B\right|\right]. \tag{4}$$

The use of write markers has no impact here on (4) because $(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B$ contains no faulty servers. However, write markers do enable us to set r smaller, as the following shows.

Second, the constraints must ensure that a conflicting candidate (which is in conflict with an established candidate as described in Section 2) is, in expectation, not observed by any client (non-faulty or faulty). In general, it is important for all clients to observe only established candidates so as to enable higher-level protocols (e.g., [4]) that employ repair phases that may affect the state of the system within a read [11]. Let $\mathsf{A}'_{\mathrm{rd}}$ and $\mathsf{A}'_{\mathrm{wt}}$ represent read and write access sets, respectively, chosen uniformly at random. (Think of $\mathsf{A}'_{\mathrm{wt}}$ as the access set used by a faulty client for a conflicting candidate, and of $\mathsf{A}'_{\mathrm{rd}}$ as the access set used by a faulty client for a read operation. How faulty clients can be forced to choose uniformly at random is described in Section 4.) We consider the cases for masking and opaque quorums separately:

Probabilistic Masking Quorums. In a masking quorum system, (2) dictates that only faulty servers may vote for a conflicting candidate. Using write markers, we require that the faulty qualified participants alone cannot produce sufficient votes for a candidate to be observed in expectation. Taking (4) into consideration, we require:

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B|\right]. \tag{5}$$

Contrast this with (2) and with the consistency requirement for traditional probabilistic masking quorum systems [10] (adapted to consider access sets), which requires that the faulty participants (qualified or not) cannot produce sufficient votes for a candidate to be observed in expectation:

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|\mathsf{A}'_{\mathrm{rd}} \cap B|\right]. \tag{6}$$

Intuitively, the intersection between access sets can be smaller with write markers because the right-hand side of (5) is less than the right-hand side of (6) if $a_{wt} < n$.

Probabilistic Opaque Quorums. With write markers, we have the benefit, described above for probabilistic masking quorums, in terms of the number of faulty participants that can vote for a candidate in expectation. However, as shown in (3), opaque quorum systems must additionally consider the maximum number of non-faulty qualified participants that vote for the same conflicting candidate in expectation. As such, instead of (5), we have:

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B|\right] + \mathbb{E}\left[|\left((\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \setminus B\right) \setminus \mathsf{Q}_{\mathrm{wt}}|\right]. \tag{7}$$

Contrast this with the consistency requirement for traditional probabilistic opaque quorums [11]:

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|\mathsf{A}'_{\mathrm{rd}} \cap B|\right] + \mathbb{E}\left[|\left((\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \setminus B\right) \setminus \mathsf{Q}_{\mathrm{wt}}|\right]. \tag{8}$$

Again, intuitively, the intersection between access sets can be smaller with write markers because the right-hand side of (7) is less than the right-hand side of (8) if $a_{wt} < n$.

3.2 Implied Bounds

In this subsection, we are concerned with quorum systems for which we can achieve error probability (as defined in Section 2) no greater than a given ϵ for any n sufficiently large. For such quorum systems, there is an upper bound on b in terms of n, akin to the bound for strict quorum systems.

Intuitively, the maximum value of b is limited by the relevant constraint (i.e., either (5) or (7)). Of primary interest are Theorem 1 and its corollaries, which demonstrate the benefits of write markers for probabilistic masking quorum systems, and Theorem 2 and its corollaries, which demonstrate the benefits of write markers for probabilistic opaque quorum systems. They utilize Lemmas 1 and 2, which together present basic requirements for the types of quorum systems with which we are concerned. Due to space constraints, proofs of the lemmas and theorems appear only in a companion technical report [15].

Define $\mathsf{MinCorrect}$ to be a random variable for the number of non-faulty servers with the established candidate, i.e., $\mathsf{MinCorrect} = |(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|$ as indicated in (4).

Lemma 1. *Let $n - b = \Omega(n)$. For all $c > 0$ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$ and $q_{rd}q_{wt} - n = \Omega(1)$, it is the case that $\mathbb{E}[\mathsf{MinCorrect}] > c$ for all n sufficiently large.*

Let r be the threshold, discussed in Section 3.1, for the number of votes necessary to observe a candidate. Define $\mathsf{MaxConflicting}$ to be a random variable for the maximum number of servers that vote for a conflicting candidate. For example: due to (5), in masking quorums with write markers, $\mathsf{MaxConflicting} = |(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B|$; and due to (7), in opaque quorums with write markers, $\mathsf{MaxConflicting} = |(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B| + |\left((\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \setminus B\right) \setminus \mathsf{Q}_{\mathrm{wt}}|$.

Lemma 2. *Let the following hold,*[2]

$$\mathbb{E}[\mathsf{MinCorrect}] - \mathbb{E}[\mathsf{MaxConflicting}] > 0,$$
$$\mathbb{E}[\mathsf{MinCorrect}] - \mathbb{E}[\mathsf{MaxConflicting}] = \omega(\sqrt{\mathbb{E}[\mathsf{MinCorrect}]}).$$

Then it is possible to set r such that,

$$\textit{error probability} \rightarrow 0 \quad \textit{as } \mathbb{E}[\mathsf{MinCorrect}] \rightarrow \infty.$$

Here and below, a suitable setting of r is one between $\mathbb{E}[\mathsf{MinCorrect}]$ and $\mathbb{E}[\mathsf{MaxConflicting}]$, inclusive. The remainder of the section is focused on determining, for each type of probabilistic quorum system, the upper bound on b and bounds on the load that Lemmas 1 and 2 imply.

Theorem 1. *For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$, $q_{rd}q_{wt} - n = \Omega(1)$, and*

$$b < \frac{q_{rd}q_{wt}n}{q_{rd}a_{wt} + a_{rd}a_{wt}},$$

any such probabilistic masking quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large.

Corollary 1. *Let $a_{rd} = q_{rd}$ and $a_{wt} = q_{wt}$. For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$, $q_{rd}q_{wt} - n = \Omega(1)$, and*

$$b < n/2,$$

any such probabilistic masking quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large.

[2] ω is the little-oh analog of Ω, i.e., $f(n) = \omega(g(n))$ if $f(n)/g(n) \rightarrow \infty$ as $n \rightarrow \infty$.

In other words, with write markers, the size of quorums does not impact the maximum fraction of faults that can be tolerated when quorums are selected uniformly at random (i.e., when $a_{rd} = q_{rd}$ and $a_{wt} = q_{wt}$).

Corollary 2. *Let $a_{rd} = q_{rd}$, $a_{wt} = q_{wt}$, and $b < n/2$. For all ϵ there is a constant $\rho > 1$ such that if $q_{rd} = q_{wt} = \rho\sqrt{n}$, any such probabilistic masking quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large, and has load*

$$\rho\sqrt{n}/n = O(1/\sqrt{n}).$$

Theorem 2. *For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$, $q_{rd}q_{wt} - n = \Omega(1)$, and*

$$b < \frac{n(a_{rd}a_{wt}^2 + a_{rd}q_{wt}n + q_{rd}q_{wt}n - 2a_{rd}a_{wt}n)}{a_{wt}(a_{rd}a_{wt} + q_{rd}n)},$$

any such probabilistic opaque quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large.

Corollary 3. *Let $a_{rd} = q_{rd}$ and $a_{wt} = q_{wt}$. For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$, $q_{rd}q_{wt} - n = \Omega(1)$, and*

$$b < \frac{q_{wt}n}{q_{wt} + n},$$

any such probabilistic opaque quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large.

Comparing Corollary 3 with Corollary 1, we see that in the opaque quorum case q_{wt} cannot be set independently of b.

Corollary 4. *Let $a_{rd} = q_{rd}$, $a_{wt} = q_{wt}$, and $b < (q_{wt}n)/(q_{wt} + n)$. For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$ and $q_{rd}q_{wt} - n = \Omega(1)$, any such probabilistic opaque quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large, and has load*

$$\Omega(b/n).$$

Corollary 5. *Let $b = \Omega(\sqrt{n})$. For all ϵ there is a constant $d > 1$ such that for all a_{rd}, a_{wt}, q_{rd}, q_{wt} where $a_{rd} = a_{wt} = q_{rd} = q_{wt} = lb$ for a value l such that $c' \geq l > n/(n - b)$ for some constant c', $(lb)^2 > dn$ and $(lb)^2 - n = \Omega(1)$, any such probabilistic opaque quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large, and has load*

$$O(b/n).$$

Corollary 6. *Let $a_{rd} = q_{rd}$ and $a_{wt} = q_{wt} = n - b$. For all ϵ there is a constant $d > 1$ such that for all q_{rd}, q_{wt} where $q_{rd}q_{wt} > dn$, $q_{rd}q_{wt} - n = \Omega(1)$, and*

$$b < n/2.62,$$

any such probabilistic opaque quorum system employing write markers achieves error probability no greater than ϵ given a suitable setting of r for all n sufficiently large.

4 Implementation

Our implementation of write markers provides the behavior assumed in Section 3, even with Byzantine clients. Specifically, it ensures properties W1–W3. (Though, technically, it ensures W2 only approximately in the case of opaque quorum systems, in which, as we explain below, a faulty server might be able to create a conflicting candidate using a write marker for a stale, i.e., out-of-date, access set—but to no advantage.)

Because clients may be faulty, we cannot rely on, e.g., digital signatures issued by them to implement write markers. Instead, we adapt mechanisms of our access-restriction protocol for probabilistic opaque quorum systems [11]. The access-restriction protocol is designed to ensure that all clients follow the access strategy. It already enables non-faulty *servers* to verify this before accepting a write. And, since it is the only way of which we are aware for a probabilistic quorum system to tolerate Byzantine clients when write markers are of benefit (i.e., when the sizes of write access sets are restricted), its mechanisms are appropriate.

The relevant parts of the preexisting protocol work as follows [11]. From a preconfigured number of servers, a client obtains a *verifiable recent value* (VRV), the value of which is unpredictable to clients and b or fewer servers prior to its creation. This VRV is used to generate a pseudorandom sequence of access sets. Since a VRV can be verified using only public information, both it and the sequence of access sets it induces can be verified by clients and servers. Non-faulty clients simply choose the next unused access set for each operation.[3] However, a faulty client is motivated to maximize the probability of error. If the use of the next access set in the sequence does not maximize the probability of error given the current state of the system (i.e., the candidates accepted by the servers), such a client may try to skip ahead some number of access sets. Alternatively, such a client might try to wait to use the next access set until the state of the system changes. If allowed to follow either strategy, such a client would circumvent the access strategy because its choice of access set would not be independent from the state of the system.

Three mechanisms are used together to coerce a faulty client to follow the access strategy. First, the client must perform exponentially increasing work in expectation in order to use later access sets. As such, a client requires exponentially

[3] Non-faulty clients should choose a new access set for each operation to ensure independence from the decisions of faulty clients [11].

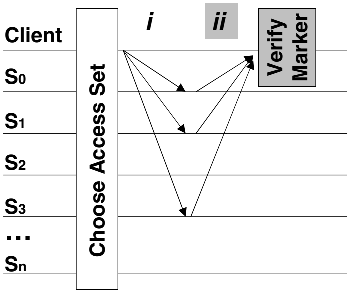


Fig. 1. Read operation with write markers: messages and stages of verification of access set (Changes in gray)

increasing time in expectation in order to choose a later access set. This is implemented by requiring that the client solve a client puzzle [16] of the appropriate difficulty. The solution to the puzzle is, in expectation, difficult to find but easy to verify. Second, the VRV and sequence of access sets become invalid as the non-faulty servers accept additional candidates, or as the system otherwise progresses (e.g., as time passes). Non-faulty servers verify that an access set is still valid, i.e., not stale, before accepting it. Thus, system progress forces the client to start its work anew, and, as such, makes the work solving the puzzle for any unused access set wasted. Finally, during the time that the client is working, the established candidate propagates in the background to the non-faulty servers that are non-qualified (c.f., [17]). This decreases the window of vulnerability in which a given access set in the sequence is useful for a conflicting write by making non-qualified servers aware that (i) there is an established candidate (so that they will not accept a conflicting candidate) and (ii) that the state of the system has progressed (so that they will invalidate the current VRV if appropriate).

The impact of these three mechanisms is that a non-faulty *server* can be confident that the choice of write access set adheres (at least approximately) to the access strategy upon having verified that the access set is valid, current, and is accompanied by an appropriate puzzle solution.

For write markers, we extend the protocol so that, as seen in Figure 1, *clients* can also perform verification. This requires that information about the puzzle solution and access set (including the VRV used to generate it) be returned by the servers to clients. (As seen in Figure 2 and explained below, this information varies across masking and opaque quorum systems.) In the preexisting access-restriction protocol, this information is verified and discarded by each server. For write markers, this information is instead stored by each server in the verification stage as a write marker. It is sent along with the data value as part of the candidate to the client during any read operation. If the server is non-faulty—a fact of which a non-faulty client cannot be certain—the access set used for the operation was indeed chosen according to the access strategy because the server performed verification before accepting the candidate. However, because the server may be faulty, the client performs verification as well; it verifies the write marker and that the server is a member of the access set. This allows us to guarantee points W1–W3. As such, faulty non-qualified servers are unable to vote for the candidates for which qualified servers can vote.

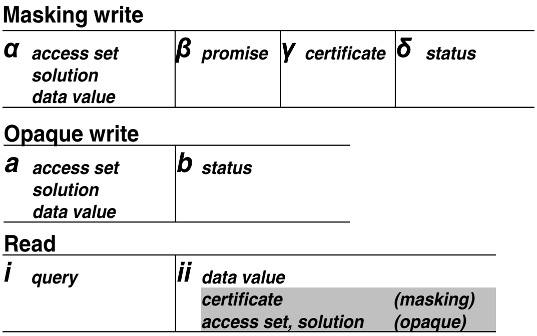


Fig. 2. Message types (Write marker emphasized with gray)

Figures 1, 2, 3, and 4 illustrate relevant pieces of the preexisting protocol and our modifications for write markers in the context of read and write operations in probabilistic masking and opaque quorum systems. The figures highlight that the additions to the protocol for write markers involve saving the write markers and returning them to clients so that clients can also verify them.

The differences in the structure of the write marker for probabilistic opaque and masking quorum systems mentioned above results in subtly different guarantees. The remainder of the section discusses these details.

4.1 Probabilistic Opaque Quorums

As seen in Figure 2 (message *ii*), a write marker for a probabilistic opaque quorum system consists of the write-access-set identifier (including the VRV) and the solution to the puzzle that unlocks the use of this access set. Unlike a non-faulty server that verifies the access set at the time of use, a non-faulty client cannot verify that an access set was not already stale when the access set was accepted by a faulty server. Initially, this may appear problematic because it is clear that, given sufficient time, a faulty client will eventually be able to solve the puzzle for its preferred access set to use for a conflicting write—this access set may contain all of the servers in B. In addition, the faulty client can delay the use of this access set because non-faulty clients will be unable to verify whether it was already stale when it was used.

Fortunately, because non-faulty servers will not accept a stale candidate (i.e., a candidate accompanied by a stale access set), the fact that a stale access set may be accepted by a faulty server does not impact the benefit of write markers for opaque quorum systems. In general, consistency requires (7), i.e.,

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B|\right] + \mathbb{E}\left[|\left((\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \setminus B\right) \setminus \mathsf{Q}_{\mathrm{wt}}|\right].$$

However, only faulty servers will accept a stale candidate. Therefore, if the candidate was stale when written to A'_{wt}, no non-faulty server would have accepted it. Thus, in this case, the consistency constraint is equivalent to,

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|(\mathsf{A}'_{\mathrm{rd}} \cap \mathsf{A}'_{\mathrm{wt}}) \cap B|\right].$$

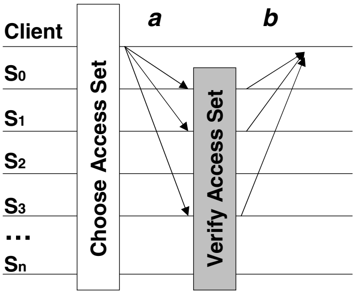


Fig. 3. Write operation in opaque quorum systems: messages and stages of verification of write marker (Changes in gray)

However, this is (6), the constraint on probabilistic masking quorum systems without write markers. In effect, a faulty client must either: (i) use a recent access set that is therefore chosen approximately uniformly at random, and be limited by (7); or (ii), use a stale access set and be limited by (6). If quorums are the sizes of access sets, both inequalities have the same upper bound on b (see [15]); otherwise, a faulty client is disadvantaged by using a stale access set because a system that satisfies (6) can tolerate more faults than one that satisfies (7), and is therefore less likely to result in error (see [15]). Even if the access set contains all of the faulty servers, i.e., $B \subset A'_{wt}$, then this becomes,

$$\mathbb{E}\left[|(\mathsf{Q}_{\mathrm{rd}} \cap \mathsf{Q}_{\mathrm{wt}}) \setminus B|\right] > \mathbb{E}\left[|\mathsf{A}'_{\mathrm{rd}} \cap B|\right].$$

4.2 Probabilistic Masking Quorums

Protocols for masking quorum systems involve an additional round of communication (an echo phase, c.f., [8] or broadcast phase, c.f., [18]) during write operations in order to tolerate Byzantine or concurrent clients. This round prevents non-faulty servers from accepting conflicting data values, as assumed by (2). In order to write a data value, a client must first obtain a *write certificate* (a quorum of replies that together attest that the non-faulty servers will accept no conflicting data value). In contrast to optimistic protocols that use opaque quorum systems, these protocols are pessimistic.

This additional round allows us to prevent clients from using stale access sets. Specifically, in the request to authorize a data value (message α in Figure 2 and Figure 4), the client sends the access set identifier (including the VRV), the solution to the puzzle enabling use of this access set, and the data value. We require that the certificate come from servers in the access set that is chosen for the write operation. Each server verifies

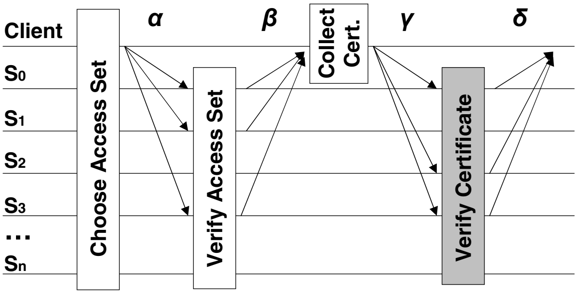


Fig. 4. Write operation in masking quorum systems: messages and stages of verification of write marker (Changes in gray)

the VRV and that the puzzle solution enables use of the indicated access set before returning authorization (message β in Figure 2 and Figure 4). The non-faulty servers that contribute to the certificate all implicitly agree that the access set is not stale, for otherwise they would not agree to the write. This certificate (sent to each server in message γ in Figure 2 and Figure 4) is stored along with the data value as a write marker. Thus, unlike in probabilistic opaque quorum systems, a verifiable write marker in a probabilistic masking quorum system implies that a stale access set was not used. The reading client verifies the certificate (returned in message ii in Figure 1 and Figure 2) before accepting a vote for a candidate. Because a writing client will be unable to obtain a certificate for a stale access set, votes for such a candidate will be rejected by reading clients. Therefore, the analysis in Section 3 applies without additional complications.

5 Additional Related Work

Probabilistic quorum systems were explored in the context of dynamic systems with non-uniform access strategies by Abraham and Malkhi [19]. Recently, probabilistic quorum systems have been used in the context of security for wireless sensor networks [20] as well as storage for mobile ad hoc networks [21]. Lee and Welch make use of probabilistic quorum systems in randomized algorithms for distributed read-write registers [22] and shared queue data structures [23].

Signed quorum systems presented by Yu [13] also weaken the requirements of strict quorum systems but use different techniques. However, signed quorum systems have not been analyzed in the context of Byzantine faults, and so they are not presently affected by write markers.

Another implementation of write markers was introduced by Alvisi et al. [24] for purposes different than ours. We achieve the goals of (i) improving the load, and (ii) increasing the maximum fraction of faults that the system can tolerate by using write markers to prevent some faulty servers from colluding. In contrast to this, Alvisi et al. use write markers in order to increase accuracy in estimating the number of faults present in Byzantine quorum systems, and for identifying faulty servers that consistently return incorrect results. Because the implementation of Alvisi et al. does not prevent faulty servers from lying about the write quorums of which they are members, it cannot be used directly for our purposes. In addition, our implementation is designed to tolerate Byzantine clients, unlike theirs.

6 Conclusion

We have presented write markers, a way to improve the load of masking and opaque quorum systems asymptotically. Moreover, our new masking and opaque probabilistic quorum systems with write markers can tolerate an additional 24% and 17% of faulty replicas, respectively, compared with the proven bounds of probabilistic quorum systems without write markers. Write markers achieve this by limiting the extent to which Byzantine-faulty servers may cooperate to provide incorrect values to clients. We have presented a proposed implementation of write markers that is designed to be effective even while tolerating Byzantine-faulty clients and servers.

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4, 382–401 (1982)
2. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11, 203–213 (1998)
3. Naor, M., Wool, A.: The load, capacity, and availability of quorum systems. SIAM Journal on Computing 27, 423–447 (1998)
4. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. In: Symposium on Operating Systems Principles (2005)
5. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Symposium on Operating Systems Design and Implementation (1999)
6. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient Byzantine-tolerant erasure-coded storage. In: International Conference on Dependable Systems and Networks (2004)
7. Kong, L., Manohar, D., Subbiah, A., Sun, M., Ahamad, M., Blough, D.: Agile store: Experience with quorum-based data replication techniques for adaptive Byzantine fault tolerance. In: IEEE Symposium on Reliable Distributed Systems, pp. 143–154 (2005)
8. Malkhi, D., Reiter, M.K.: An architecture for survivable coordination in large distributed systems. IEEE Transactions on Knowledge and Data Engineering 12, 187–202 (2000)
9. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing 3, 202–215 (2006)
10. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Information and Computation 170, 184–206 (2001)
11. Merideth, M.G., Reiter, M.K.: Probabilistic opaque quorum systems. In: International Symposium on Distributed Computing (2007)
12. Malkhi, D., Reiter, M.K., Wool, A.: The load and availability of Byzantine quorum systems. SIAM Journal of Computing 29, 1889–1906 (2000)
13. Yu, H.: Signed quorum systems. Distributed Computing 18, 307–323 (2006)
14. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12, 463–492 (1990)
15. Merideth, M.G., Reiter, M.K.: Write markers for probabilistic quorum systems. Technical Report CMU-CS-07-165R, Computer Science Department, Carnegie Mellon University (2008)
16. Juels, A., Brainard, J.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In: Network and Distributed Systems Security Symposium, pp. 151–165 (1999)
17. Malkhi, D., Mansour, Y., Reiter, M.K.: Diffusion without false rumors: On propagating updates in a Byzantine environment. Theoretical Computer Science 299, 289–306 (2003)
18. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal Byzantine storage. In: International Symposium on Distributed Computing (2002)

19. Abraham, I., Malkhi, D.: Probabilistic quorums for dynamic systems. Distributed Computing 18, 113–124 (2005)
20. Du, W., Deng, J., Han, Y.S., Varshney, P.K., Katz, J., Khalili, A.: A pairwise key predistribution scheme for wireless sensor networks. ACM Transactions on Information and System Security 8, 228–258 (2005)
21. Luo, J., Hubaux, J.P., Eugster, P.T.: Pan: providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In: International symposium on mobile ad hoc networking and computing, pp. 1–12 (2003)
22. Lee, H., Welch, J.L.: Applications of probabilistic quorums to iterative algorithms. In: International Conference on Distributed Computing Systems, pp. 21–30 (2001)
23. Lee, H., Welch, J.L.: Randomized shared queues applied to distributed optimization algorithms. In: International Symposium on Algorithms and Computation (2001)
24. Alvisi, L., Malkhi, D., Pierce, E., Reiter, M.K.: Fault detection for Byzantine quorum systems. IEEE Transactions on Parallel and Distributed Systems 12, 996–1007 (2001)

Byzantine Consensus with Unknown Participants

Eduardo A. P. Alchieri[1], Alysson Neves Bessani[2],
Joni da Silva Fraga[1], and Fabíola Greve[3]

[1] Department of Automation and Systems
Federal University of Santa Catarina (UFSC)
Florianópolis, SC - Brazil
alchieri@das.ufsc.br,fraga@das.ufsc.br
[2] Large-Scale Informatics Systems Laboratory
Faculty of Sciences, University of Lisbon
Lisbon, Portugal
bessani@di.fc.ul.pt
[3] Department of Computer Science
Federal University of Bahia (UFBA)
Bahia, BA - Brazil
fabiola@dcc.ufba.br

Abstract. Consensus is a fundamental building block used to solve many practical problems that appear on reliable distributed systems. In spite of the fact that consensus is being widely studied in the context of classical networks, few studies have been conducted in order to solve it in the context of dynamic and self-organizing systems characterized by unknown networks. While in a classical network the set of participants is static and known, in a scenario of unknown networks, the set and number of participants are previously unknown. This work goes one step further and studies the problem of *Byzantine Fault-Tolerant Consensus with Unknown Participants*, namely BFT-CUP. This new problem aims at solving consensus in unknown networks with the additional requirement that participants in the system can behave maliciously. This paper presents a solution for BFT-CUP that does not require digital signatures. The algorithms are shown to be optimal in terms of synchrony and knowledge connectivity among participants in the system.

Keywords: Consensus, Byzantine fault tolerance, Self-organizing systems.

1 Introduction

The *consensus* problem [1,2,3,4,5], and more generally the *agreement* problems, form the basis of almost all solutions related to the development of reliable distributed systems. Through these protocols, participants are able to coordinate their actions in order to maintain state consistency and ensure system progress. This problem has been extensively studied in classical networks, where the set of processes involved in a particular computation is static and known by all participants in the system. Nonetheless, even in these environments, the consensus problem has no deterministic solution in presence of one single process crash, when entities behave asynchronously [2].

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 22–40, 2008.

In self-organizing systems, such as wireless mobile ad-hoc networks, sensor networks and, in a different context, unstructured peer to peer networks (P2P), solving consensus is even more difficult. In these environments, an initial knowledge about participants in the system is a strong assumption to be adopted and the number of participants and their knowledge cannot be previously determined. These environments define indeed a new model of distributed systems which has essential differences regarding the classical one. Thus, it brings new challenges to the specification and resolution of fundamental problems. In the case of consensus, the majority of existing protocols are not suitable for the new dynamic model because their computation model consists of a set of initially known nodes. The only notably exceptions are the works of Cavin *et al.* [6,7] and Greve *et al.* [8].

Cavin *et al.* [6,7] defined a new problem named FT-CUP (*fault-tolerant consensus with unknown participants*) which keeps the consensus definition but assumes that nodes are *not* aware of Π, the set of processes in the system. They identified necessary and sufficient conditions in order to solve FT-CUP concerning knowledge about the system composition and synchrony requirements regarding the failure detection. They concluded that in order to solve FT-CUP in a scenario with the weakest knowledge connectivity, the strongest synchrony conditions are necessary, which are represented by failures detectors of the class $\mathscr{P}$ [4].

Greve and Tixeuil [8] show that there is in fact a trade-off between knowledge connectivity and synchrony for consensus in fault-prone unknown networks. They provide an alternative solution for FT-CUP which requires minimal synchrony assumptions; indeed, the same assumptions already identified to solve consensus in a classical environment, which are represented by failure detectors of the class $\Diamond\mathscr{S}$ [4]. The approach followed on the design of their FT-CUP protocol is modular: Initially, algorithms identify a set of participants in the network that share the same view of the system. Subsequently, any classical consensus – like for example, those initially designed for traditional networks – can be reused and executed by these participants.

Our work extends these results and study the problem of *Byzantine Fault-Tolerant Consensus with Unknown Participants* (BFT-CUP). This new problem aims at solving CUP in unknown networks with the additional requirement that participants in the system can behave maliciously [1]. The main contribution of the paper is then the identification of necessary and sufficient conditions in order to solve BFT-CUP. More specifically, an algorithm for solving BFT-CUP is presented for a scenario which does not require the use of digital signatures (a major source of performance overhead on Byzantine fault-tolerant protocols [9]). Finally, we show that this algorithm is optimal in terms of synchrony and knowledge connectivity requirements, establishing then the necessary and sufficient conditions for BFT-CUP solvability in this context.

The paper is organized in the following way. Section 2 presents our system model and the concept of participant detectors, among other preliminary definitions used in this paper. Section 3 describes a basic dissemination protocol used for process communication. BFT-CUP protocols and respective necessary and sufficient proofs are described in Section 4. Section 5 presents some comments about our protocol. Section 6 presents our final remarks.

2 Preliminaries

2.1 System Model

We consider a distributed system composed by a finite set Π of n processes (also called participants or nodes) drawn from a larger universe U. In a *known network*, Π and n is known to every participanting process, while in an *unknown network*, a process $i \in \Pi$ may only be aware of a subset $\Pi_i \subseteq \Pi$.

Processes are subject to *Byzantine failures* [1], i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Processes that do not follow their algorithm in some way are said to be *faulty*. A process that is not faulty is said to be *correct*. Despite the fact that a process does not know all participants of the system, it does know the expected maximum number of process that may fail, denoted by f. Moreover, we assume that all processes have a unique id, and that it is infeasible for a faulty process to obtain additional ids to be able to launch a *sybil attack* [10] against the system.

Processes communicate by sending and receiving messages through *authenticated and reliable point to point channels* established between known processes[1]. Authenticity of messages disseminated to a not yet known node is verified through message channel redundancy, as explained in Section 3. A process i may only send a message directly to another process j if $j \in \Pi_i$, i.e., if i knows j. Of course, if i sends a message to j such that $i \notin \Pi_j$, upon receipt of the message, j may add i to Π_j, i.e., j now knows i and become able to send messages to it. We assume the existence of an underlying routing layer resilient to Byzantine failures [11,12,13], in such a way that if $j \in \Pi_i$ and there is sufficient network connectivity, then i can send a message reliably to j. For example, [12] presents a secure multipath routing protocol that guarantees a proper communication between two processes provided that there is at least one path between these processes that is not compromised, i.e., none of its processes or channels are faulty.

There are no assumptions on the relative speed of processes or on message transfer delays, i.e., the system is asynchronous. However, the protocol presented in this paper uses an underlying classical Byzantine consensus that could be implemented over an eventually synchronous system [14] (e.g., Byzantine Paxos [9]) or over a completely asynchronous system (e.g., using a randomized consensus protocol [5,15,16]). Thus, our protocol requires the same level of synchrony required by the underlying classical Byzantine consensus protocol.

2.2 Participant Detectors

To solve any nontrivial distributed problem, processes must somehow get a partial knowledge about the others if some cooperation is expected. The *participant detector* oracle, namely PD, was proposed to handle this subset of known processes [6]. It can be seen as a distributed oracle that provides hints about the participating processes in the computation. Let *i.PD* be defined as the participant detector of a process i. When

[1] Without authenticated channels it is not possible to tolerate process misbehavior in an asynchronous system since a single faulty process can play the roles of all other processes to some (victim) process.

queried by i, $i.PD$ returns a subset of processes in Π with whom i can collaborate. Let $i.PD(t)$ be the query of i at time t. The information provided by $i.PD$ can evolve between queries, but must satisfy the following two properties:

- *Information Inclusion*: The information returned by the participant detectors is non-decreasing over time, i.e., $\forall i \in \Pi, \forall t' \geq t : i.PD(t) \subseteq i.PD(t')$;
- *Information Accuracy*: The participant detectors do not make mistakes, i.e., $\forall i \in \Pi, \forall t : i.PD(t) \subseteq \Pi$.

Participant detectors provide an initial context about participants present in the system by which it is possible to expand the knowledge about Π. Thus, the participant detector abstraction enriches the system with a knowledge connectivity graph. This graph is directed since the knowledge provided by participant detectors is not necessarily bidirectional [6].

Definition 1. ***Knowledge Connectivity Graph****: Let $G_{di} = (V, \xi)$ be the directed graph representing the knowledge relation determined by the PD oracle. Then, $V = \Pi$ and $(i, j) \in \xi$ iff $j \in i.PD$, i.e., i knows j.*

Definition 2. ***Undirected Knowledge Connectivity Graph****: Let $G = (V, \xi)$ be the undirected graph representing the knowlegde relation determined by the PD oracle. Then, $V = \Pi$ and $(i, j) \in \xi$ iff $j \in i.PD$ or $i \in j.PD$, i.e., i knows j or j knows i.*

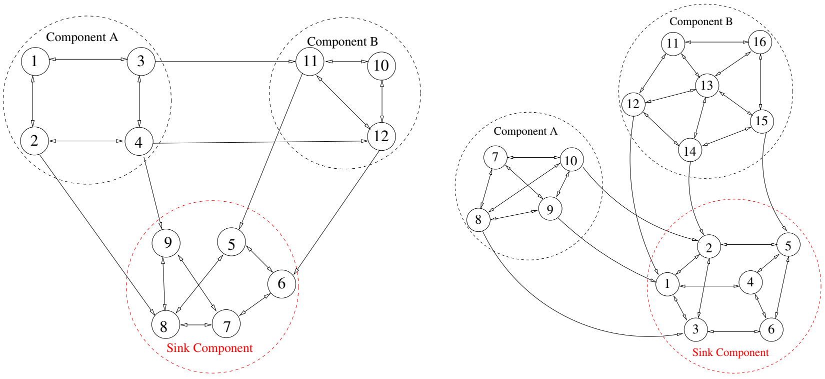
Based on the properties of the knowledge connectivity graph, some classes of participant detectors have been proposed to solve CUP [6] and FT-CUP [7,8]. Before defining how a participant detector encapsulates the knowledge of a system, let us define some graph notations. We say that a component G_c of G_{di} is *k-strongly connected* if for any pair (v_i,v_j) of nodes in G_c, v_i can reach v_j through k node-disjoint paths. A component G_s of G_{di} is a *sink component* when there is no path from a node in G_s to other nodes of G_{di}, except nodes in G_s itself. In this paper we use the weakest participant detector defined to solve FT-CUP, which is called k-OSR [8].

Definition 3. ***k-One Sink Reducibility (k-OSR) PD****: The knowledge connectivity graph G_{di}, which represents the knowledge induced by PD, satisfies the following conditions:*

1. *the undirected knowledge connectivity graph G obtained from G_{di} is connected;*
2. *the directed acyclic graph obtained by reducing G_{di} to its k-strongly connected components has exactly one sink;*
3. *consider any two k-strongly connected components G_1 and G_2, if there is a path from G_1 to G_2, then there are k node-disjoint paths from G_1 to G_2.*

To better illustrate Definition 3, Figure 1 presents two graphs G_{di} induced by a k-OSR participant detector. Figures 1(a) and 1(b) show knowledge relations induced by participant detectors of the class 2-OSR and 3-OSR, respectively. For example, in Figure 1(a), the value returned by $1.PD$ is the subset $\{2, 3\} \subset \Pi$.

In our algorithms, we assume that for each process i, its participant detector $i.PD$ is queried exactly once at the beginning of the protocol execution. This can be implemented by caching the result of the first query to $i.PD$ and returning that value in



(a) 2-OSR (b) 3-OSR

Fig. 1. Knowledge Connectivity Graphs Induced by k-OSR Participant Detectors

subsequent calls. This ensures that the partial view about the initial composition of the system is consistent for all nodes in the system, what defines a common knowledge connectivity graph G_{di}. Also, in this work we say that some participant p is *neighbor* of another participant i iff $p \in i.PD$.

2.3 The Consensus Problem

In a distributed system, the consensus problem consists of ensuring that all correct processes eventually decide the same value, previously proposed by some processes in the system. Thus, each process i *proposes* a value v_i and all correct processes *decide* on some unique value v among the proposed values. Formally, consensus is defined by the following properties [4]:

- *Validity*: if a correct process decides v, then v was proposed by some process;
- *Agreement*: no two correct processes decide differently;
- *Termination*: every correct process eventually decides some value[2];
- *Integrity*: every correct process decides at most once.

The *Byzantine Fault-Tolerant Consensus with Unknown Participants*, namely BFT-CUP, proposes to solve consensus in unknown networks with the additional requirement that a bounded number of participants in the system can behave maliciously.

3 Reachable Reliable Broadcast

This section introduces a new primitive, namely *reachable reliable broadcast*, used by processes of the system to communicate. It is invoked by two basic operations:

- ***reachable_send(m,p)*** – through which the participant p sends the message m to all reachable participants from p. A participant q is reachable from another participant

[2] If a randomized protocol such as [5,15,17] is used as an underlying Byzantine consensus, the termination is ensured only with probability 1.

p if there is enough connectivity from p to q (see below). In this case, q is a receiver of messages disseminated by p.

- ***reachable_deliver(m,p)*** – invoked by the receiver to deliver a message m disseminated by the participant p.

This primitive should satisfy the following four properties:

- *Validity*: If a correct participant p disseminates a message m, then m is eventually delivered by a correct participant reachable from p or there is no correct participant reachable from p;
- *Agreement*: If a correct participant delivers some message m, disseminated by a correct participant p, then all correct participants reachable from p eventually deliver m;
- *Integrity*: For any message m, every correct participant p delivers m only if m was previously disseminated by some participant p', in this case p is reachable from p'.

Notice that these properties establish a communication primitive with specification similar to the usual reliable broadcast [4,5,15]. Nonetheless, the proposed primitive ensures the delivery to all correct processes reachable in the system.

Implementation. The main idea of our implementation is that participants execute a flood of their messages to all reachable processes, which, in turn, will deliver these messages as soon as its authenticity has been proved. Assuming a k-OSR PD, a participant q is reachable from a participant p if there is enough connectivity in the knowlegde graph, i.e., if there are at least $2f+1$ node-disjoint paths from p to q ($k \geq 2f+1$). This connectivity is necessary to ensure that all reachable processes will be able to receive and authenticate messages.

In our implementation, formally described in Algorithm 1, a process i disseminates a message m through the system by executing the procedure *reachable_send*. In this procedure (line 6), i sends m to its neighbors (i.e., processes in $i.PD$) and when m is received at some process p, p forwards m to its neighbors and so on, until that m arrives at all reachable participants (line 17). Moreover, p stores m together with the route traversed by m in a buffer (line 11). Also, p delivers m if it has received m through $f+1$ node-disjoint paths (lines 13-14), i.e., the authenticity of m has been verified. Afterward, since m has been delivered, p removes it from the buffer of received messages (line 15). The function $computeRoutes(m.message, i.received\_msgs)$ computes the number of node-disjoint paths through which $m.message$ has been received at participant i.

An important feature of this dissemination is that each message has the accumulated route according with the path traversed from the sender to some destination. A participant will process a received message only if the participant that is sending (or forwarding) this message appears at the end of the accumulated route (line 8). This solution is based on the approach used in [18] and it enforces that each participant appends itself at the end of the routing information in order to send or forward a message. Nonetheless, a malicious participant is able to modify the accumulated route (removing or adding participants) and modify or block the message being propagated. Notice, however, that the connectivity of the knowledge graph ($k \geq 2f+1$) ensures that messages will be received at all reachable participants. Moreover, since a process delivers a message only

Algoritm 1. Dissemination algorithm executed at participant *i*.

```
constant:
$f$ : $int$                                          // upper bound on the number of failures

variables:
$i.received\_msgs$ : set of $\langle message, route \rangle$ tuples    // set of received messages

message:
REACHABLE_FLOODING:                                  // struct of this message
    $message$ :value to flood                        // value to be disseminated
    $route$ : ordered list of nodes                  // path traversed by message

** Initiator Only **
procedure: reachable_send($message, sender$)                   // sender = i
$\forall j \in i.PD$, send REACHABLE_FLOODING($message, sender$) to $j$;

** All Nodes **
INIT:
$i.received\_msgs \leftarrow \emptyset$;

upon receipt of REACHABLE_FLOODING($m.message, m.route$) from $j$
if $getLastElement(m.route) = j \wedge i \notin m.route$ then
    $append(m.route, i)$;
    $initiator \leftarrow getFirstElement(m.route)$;
    $i.received\_msgs \leftarrow i.received\_msgs \cup \{\langle m.message, m.route \rangle\}$;
    $routes \leftarrow computeRoutes(m.message, i.received\_msgs)$;
    if $routes \geq f+1$ then
        trigger reachable_deliver($m.message, initiator$);
        $i.received\_msgs \leftarrow i.received\_msgs \setminus \{\langle m.message, * \rangle\}$;
    end if
    $\forall z \in i.PD \setminus \{j\}$, send REACHABLE_FLOODING($m.message, m.route$) to $z$;
end if
```

after it has been received through $f+1$ node disjoint paths, it is able to verify its authenticity. These measures prevent the delivery of forged messages (generated by malicious participants), because the authenticity of them cannot be verified by correct processes.

An "undesirable" property of the proposed solution is that the same message, sent by some participant, could be delivered more than once by its receivers. This property does not affect the use of this protocol in our consensus protocol (Section 4). Thus, we do not deal with this limitation of the algorithm. However, it can be easily solved by using buffers to store delivered messages that must have unique identifiers.

Additionaly, each message' receiver, disseminated by some participant p, is able to send back a reply to p using some routing protocol resilient to Byzantine failures [11,12,13]. Our BFT-CUP protocol (Section 4) uses this algorithm to disseminate messages.

Sketch of Proof. The correctness of this protocol is based on the proof of the properties defined for the reachable reliable broadcast.

Validity: By assumption, the connectivity of the system is $k \geq 2f+1$. Thus, according to Definition 3, there are at least $2f+1$ node-disjoint paths from the sender of a message m to the receivers (nodes that are reachable from the sender). Moreover, as validity is established over messages sent by correct participants (correct sender), there are at least $f+1$ node-disjoint paths formed only by correct participants, through which it is guaranteed that the same message m will reach the correct receivers. In this case, the predicate of line 8 will be true at least $f+1$ times and the authenticity of m can be verified through redundancy. This is done by the execution of lines 9–12, which are responsible to maintain information regarding the different routes from which m has been received. Whenever the message authenticity is proved, i.e., m has been received by at least $f+1$ different routes (line 13), the delivery of m is authorized by the invocation of *reachable_deliver* (line 14).

Agreement: As the agreement is established over messages sent by correct participants, this proof is identical to the validity proof.

Integrity: A message is delivered only after its reception through $f+1$ node-disjoint paths (lines 13-14), what guarantees that this message is authentic, i.e., this message was really sent by its sender (*sender*). Thus, a malicious participant j is not able to forge that message m was sent by a participant i because the autenticity of m will not be proven. That is, a receiver r will not be able to find $f+1$ node-disjoint paths from i to r through which m has been received. Even with a collusion of up to f malicious participants, r will obtain at most f node-disjoint paths through which m was received "from i" (each of these f paths could contain one malicious participant). □

4 BFT-CUP: Byzantine Consensus with Unknown Participants

This section presents our solution for BFT-CUP. Our protocol is based on the dissemination algorithm presented in Section 3, which, together with the underlying routing layer resilient to Byzantine failures, hides all details related to participants communication. Thereafter, as in [8], the consensus protocol with unknown participants is divided into three phases. In the first phase – called *participants discovery* (Section 4.1) – each participant increases its knowledge about other processes in the system, discovering the maximum possible number of participants that are present in some computation. The second phase – called *sink component determination* (Section 4.2) – defines which participants belong to the sink component of the knowlegde graph induced by a k-OSR PD. Thus, each participant will be able to determine whether it belongs to the sink component or not. In the last phase (Section 4.3), members of the sink component *execute a classical Byzantine fault tolerant consensus* and disseminate the decision value to other participants in the system. The number of participants in the sink component, namely n_{sink}, should be enough in order to e xecute a classical Byzantine fault-tolerant consensus. Usually $n_{sink} \geq 3f+1$, to run, for example, Byzantine Paxos [9,19].

4.1 Participants Discovery

The first step to solve consensus in a system with unknown participants is to provide processes with the maximum possible knowledge about the system. Notice that, through its local participant detector, a process is able to get an initial knowledge about the system that is not enough to solve BFT-CUP. Then, a process expands this knowledge by executing the DISCOVERY protocol, presented in Algorithm 2. The main idea is that each participant i broadcasts a message requesting information about neighbors of each reachable participant, making a sort of breadth-first search in the knowledge graph. At the end of the algorithm, i obtains the maximal set of reachable participants, which represents the participants known by i (a partial view of the system).

The algorithm uses three sets:

1. $i.known$ – set containing identifiers of all processes known by i;
2. $i.msg\_pend$ – this set contains identifiers of processes that should send a message to i, i.e., for each $j \in i.msg\_pend$, i should receive a message from j;
3. $i.nei\_pend$ – this set contains identifiers of processes that i knows, but does not know all of their neighbors (i is still waiting for information about them), i.e., for each $\langle j, j.neighbor\rangle \in i.nei\_pend$, i knows j but does not know all neighbors of j.

In the initialization phase of the algorithm for participant i, the set $i.known$ is updated to itself plus its neighbors, returned by $i.PD$, and the set $i.msg\_pend$ to its neighbors (line 7). Moreover, a message requesting information about neighbors is disseminated to all participants reachable from i (line 8). When a participant p delivers this message, p sends back to i a reply indicating its neighbors (line 9).

Upon receipt of a reply at participant i, the set of known participants is updated, along with the set of pending neighbors[3] and the set of pending messages (lines 10 - 12). The next step is to verify whether i has acquired knowledge about any new participant (line 13 - 16). Thus, i gets to know other participant j if at least $f+1$ other processes known by i reported to i that j is their neighbor (line 13). After this verification, the set of pending neighbors is updated (lines 17 - 21), according to the new participants discovered.

To determine if there is still some participant to be discovered, i uses the sets $i.msg\_pend$ and $i.nei\_pend$, which store the pendencies related to the replies received by i. Then, the algorithm ends when there remain at most f pendencies (lines 22 - 24). The intuition behind this condition is that if there are at most f pendencies at process i, then i already has discovered all processes reachable from it because $k \geq 2f+1$. Thus, the algorithm ends by returning the set of participants discovered by i (line 23), which contains all participants (correct or faulty) reachable from it. Algorithm 2 satisfies some properties that are stated by Lemma 1.

Lemma 1. *Consider G_{di} a knowlegde graph induced by a k-OSR PD. Let $f < \frac{k}{2} < n$ be the number of nodes that may fail. Algorithm DISCOVERY executed by each correct participant p satisfies the following properties:*

- *Termination: p terminates the execution of the algorithm and returns a set of known processes;*
- *Accuracy: the algorithm returns the maximal set of processes reachable from p in G_{di}.*

[3] If i reaches p, i also reaches all neigbours of p and should receive a reply to its initial dissemination (line 8) from all of them.

Algorithm 2. Algorithm DISCOVERY executed at participant *i*.

```
constant:
 1. $f$ : $int$                                  // upper bound on the number of failures

variables:
 2. $i.known$ : set of nodes                                        // set of known nodes
 3. $i.nei\_pend$ : set of $\langle node, node.neighbor\rangle$ tuples
                                               // $i$ does not know all neighbors of $node$
 4. $i.msg\_pend$ : set of nodes           // nodes that $i$ is waiting for messages (replies)

message:
 5. SET_NEIGHBOR:                               // struct of the message SET_NEIGHBOR
 6.     $neighbor$ : set of nodes      // neighbors of the node that is sending the message

** All Nodes **
INIT:
 7. $i.known \leftarrow \{i\} \cup i.PD$; $i.nei\_pend \leftarrow \emptyset$; $i.msg\_pend \leftarrow i.PD$;
 8. $reachable\_send$(GET_NEIGHBOR, $i$);

upon execution of $reachable\_deliver$(GET_NEIGHBOR, $sender$)
 9. send SET_NEIGHBOR($i.PD$) to $sender$;

upon receipt of SET_NEIGHBOR($m.neighbor$) from $sender$
10. $i.known \leftarrow i.known \cup \{sender\}$;
11. $i.nei\_pend \leftarrow i.nei\_pend \cup \{\langle sender, m.neighbor\rangle\}$;
12. $i.msg\_pend \leftarrow i.msg\_pend \setminus \{sender\}$;
13. if $(\exists j : \#_{\langle *,\langle j\rangle\rangle} i.nei\_pend > f) \wedge (j \notin i.known)$ then
14.     $i.known \leftarrow i.known \cup \{j\}$;
15.     $i.msg\_pend \leftarrow i.msg\_pend \cup \{j\}$;
16. end if
17. for all $\langle j, j.neighbor\rangle \in i.nei\_pend$ do
18.     if $(\forall z \in j.neighbor : z \in i.known)$ then
19.         $i.nei\_pend \leftarrow i.nei\_pend \setminus \{\langle j, j.neighbor\rangle\}$;
20.     end if
21. end for
22. if $(|i.nei\_pend| + |i.msg\_pend|) \leq f$ then
23.     return $i.known$;
24. end if
```

Sketch of Proof. *Termination:* In the worst case, the algorithm ends when p receives replies from at least all correct reachable participants (line 22). By dissemination protocol properties, even in the presence of $f < \frac{k}{2}$ failures, all messages disseminated by p is delivered by its correct receivers (processes reachable from p). Thus, each correct participant reachable from p receives a request (line 8) and sends back a reply (line 9) that is received by p (lines 10 - 24). Then, as Π is finite, it is guaranteed that p receives replies from at least all correct reachable participants and ends the algorithm by returning a set of known processes.

Accuracy: The algorithm only ends when there remain at most f pendencies, which may be divided between processes that supply information about neighbors that do not exist in the system (*i.nei_pend*) and processes from which p is still waiting for their messages/replies (*i.msg_pend*). Moreover, each participant z (being z reachable from p) is neighbor of at least $2f+1$ other participants, because $f < \frac{k}{2} < n$. Now, we have to consider two cases:

- If z is malicious and does not send back a reply to p (line 9), then p computes messages (replies) from at least $f+1$ correct neighbors of z, discovering z (lines 13 - 16).
- If z is correct, in the worst case, the message from z to p is delayed and f neighbors of z are malicious and do not inform p that z is in the system. However, as $f < \frac{k}{2}$, there remain $f+1$ correct neighbors of z in the system that inform p about the presence of z in the system.

As the algorithm only ends when there remain at most f pendencies, in both cases it is guaranteed that p only ends after discovering z, even if it firstly computes messages from the f malicious processes. □

4.2 Sink Component Determination

The objective of this phase is to define which participants belong to the sink component of the knowlegde graph induced by a k-OSR PD. More specifically, through Algorithm 3 (SINK), each participant is able to determine whether or not it is member of the sink component. The idea behind this algorithm is that after the execution of the procedure DISCOVERY, members in the sink component obtain the same partial view of the system, whereas in the other components, nodes have strictly more knowledge than in the sink, i.e., each node knows at least members of the component to which it belongs and members of the sink (see Definition 3).

In the initialization phase of the algorithm for participant i, i executes the DISCOVERY procedure in order to obtain its partial view of the system (line 8) and sends this view to all reachable/known participant (line 10). When these messages are delivered by some participant j, j sends back an *ack* response to i if it has the same knowledge of i (i.e., j belongs to the same component of i). Otherwise, j sends back a *nack* response (lines 11-15).

Upon receipt of a reply (lines 16-27), i updates the set of processes that have already answered (line 16). Moreover, if the reply received is a *nack*, the set of processes that belong to other components (*i.nacked*) is updated (line 18) and if the number of processes that do not belong to the same component of i is greater than f (line 19), i concludes that it does not belong to the sink component (lines 20-21). This condition holds because the system has at least $3f+1$ processes in the sink, known by all participants, that have strictly less knowledge about Π than processes not in the sink (Lemma 1). On the other hand, if i has received replies from all known processes, excluding f possible faulty (line 24), and the number of processes that belong to other components is not greater than f, i concludes that it belongs to the sink component (lines 25-26). This condition holds because processes in the sink receive messages only from members of this component. Moreover, in both cases, a collusion of f malicious participants cannot lead a process to decide incorrectly. Lemma 2 states the properties satisfied by Algorithm 3.

Algorithm 3. Algorithm SINK executed at participant *i*.

```
constant:
  $f$ : int                                  // upper bound on the number of failures

variables:
  $i.known$ : set of nodes                   // set of known nodes
  $i.responded$ : set of nodes               // set of nodes that has sent a reply to $i$
  $i.nacked$ : set of nodes                  // set of processes not in the same component of $i$
  $i.in\_the\_sink$ : boolean                // is $i$ in the sink?

message:
  RESPONSE:                                  // struct of the message RESPONSE
      $ack/nack$ : boolean

** All Nodes **
INIT:
  $i.known \leftarrow$ DISCOVERY();
  $i.responded \leftarrow \{i\}$; $i.nacked \leftarrow \emptyset$;
  $reachable\_send(i.known, i)$;

upon execution of $reachable\_deliver(sender.known, sender)$
  if $i.known = sender.known$ then
      send RESPONSE($ack$) to $sender$;
  else
      send RESPONSE($nack$) to $sender$;
  end if

upon receipt of RESPONSE($m$) from $sender$
  $i.responded \leftarrow i.responded \cup \{sender\}$
  if $m.nack$ then
      $i.nacked \leftarrow i.nacked \cup \{sender\}$;
      if $|i.nacked| \geq f + 1$ then
         $i.in\_the\_sink \leftarrow false$;
         return $\langle i.in\_the\_sink, i.known \rangle$;
      end if
  end if
  if $|i.responded| \geq |i.known| - f$ then
      $i.in\_the\_sink \leftarrow true$;
      return $\langle i.in\_the\_sink, i.known \rangle$;
  end if
```

Lemma 2. *Consider a k-OSR PD. Let $f < \frac{k}{2} < n$ be the number of nodes that may fail. Algorithm SINK, executed by each correct participant p of the system that has at least $3f + 1$ nodes in the sink component, satisfies the following properties:*

- *Termination: p terminates the execution by deciding whether it belongs (true) or not (false) to the sink;*
- *Accuracy: p is in the unique k-strongly connected sink component iff algorithm SINK returns true.*

Sketch of Proof. *Termination:* For each participant p, the algorithm returns in two cases: (i) when it receives $f+1$ replies from processes that belong to other components (processes not in the sink – line 19) or (ii) when it receives replies from at least all correct known processes (processes in the sink – line 24). By properties of the dissemination protocol, even in the presence of $f < \frac{k}{2}$ failures, all messages disseminated by p are delivered by its receivers (processes reachable from p). Thus, each correct participant known by p (reachable from p) receives the request (line 10) and sends back a reply (lines 11-15) that is received by p (lines 16-27). Then, it is guaranteed that either (i) or (ii) always occur.

Accuracy: By Lemma 1, after execution of the DISCOVERY algorithm, each correct participant discovers the maximal set of participants reachable from it. Then, by Lemma 1 and by k-OSR PD properties, it is guaranteed that all correct processes that belong to the same component obtain the same partial view of the system. Thus, as members in the sink component receive replies only from members of this component, it is guaranteed that these participants end correctly (line 26). Moreover, as the sink has at least $3f+1$ nodes, members in other components know at least $2f+1$ correct members in the sink (Lemma 1). Then, before making a wrong decision, these members must compute at least $f+1$ replies from correct members in the sink (that have strictly less knowledge about Π, due to Lemma 1), what makes it possible for correct members not in the sink to end correctly (line 21). □

4.3 Achieving Consensus

This is the last phase of the protocol for solving BFT-CUP. Here, the main idea is to make members of the sink component execute a classical Byzantine consensus and send the decision value to other participants of the system. The optimal resilience of these algorithms to solve a classical consensus is $3f+1$ [3,9]. Thus, it is necessary at least $3f+1$ participants in the sink component.

The Algorithm 4 (CONSENSUS) presents this protocol. In the initialization, each participant executes the SINK procedure (line 11) in order to get its partial view of the system and decide whether or not it belongs to the sink component. Depending on whether or not the node belongs to the sink, two distinct behaviors are possible:

1. Nodes in the sink execute a classical consensus (line 13) and send the decision value to other participants (lines 18 and 20-24). By construction, all correct nodes in the sink component share the same partial view of the system (exactly the members in the sink – Lemma 1). Thus, these nodes know at least $2f+1$ correct members that belong to the sink component, what makes possible to reach the properties of the classical Byzantine consensus (Section 2.3);
2. Other nodes (in the remaining components) do not participate to the classical consensus. These nodes ask for the decison value to all known nodes, i.e., all reachable nodes, what includes all nodes in the sink (line 15). Each node decides for a value v only after it has received v from at least $f+1$ other participants, ensuring that v is gathered from at least one correct participant (lines 25-31). Theorem 1 shows that Algorithm 4 solves the BFT-CUP problem as defined in Section 2.3 with the stated participant detector and connectivity requirements.

Algorithm 4. Algorithm CONSENSUS executed at participant i.

```
constant:
$f$ : $int$                                          // upper bound on the number of failures

input:
$i.initial$ : $value$                                // proposal value ($input$)

variables:
$i.in\_the\_sink$ : $boolean$                        // is $i$ in the sink?
$i.known$ : set of nodes                             // partial view of $i$
$i.decision$ : $value$                               // decision value
$i.asked$ : set of nodes                             // nodes that have required the decision value
$i.values$ : set of $\langle node, value\rangle$ tuples   // reported decisions

message:
SET_DECISION:                                        // struct of the message SET_DECISION
    $decision$ : $value$                             // the decided value

** All Nodes **
INIT: {Main Decision Task}
$i.decision \leftarrow \perp$; $i.values \leftarrow \varnothing$; $i.asked \leftarrow \varnothing$;
$(i.in\_the\_sink, i.known) \leftarrow$ SINK();
if $i.in\_the\_sink$ then
    $Consensus.propose(i.initial)$;                  // underlying Byzantine consensus with all
    $p \in i.known$
else
    $reachable\_send$(GET_DECISION, $i$);
end if

** Node In Sink **
upon $Consensus.decide(v)$
$i.decision \leftarrow v$;
$\forall j \in i.asked$, send SET_DECISION($i.decision$) to $j$;
return $i.decision$;

upon execution of $reachable\_deliver$(GET_DECISION, $sender$)
if $i.decision = \perp$ then
    $i.asked \leftarrow i.asked \cup \{sender\}$;
else
    send SET_DECISION($i.decision$) to $sender$;
end if

** Node Not In Sink **
upon receipt of SET_DECISION($m.decision$) from $sender$
if $i.decision = \perp$ then
    $i.values \leftarrow i.values \cup \{\langle sender, m.decision\rangle\}$;
    if $\#_{\langle *, m.decision\rangle}$ $i.values \geq f + 1$ then
        $i.decision \leftarrow m.decision$;
        return $i.decision$;
    end if
end if
```

Theorem 1. *Consider a classical Byzantine consensus protocol. Algorithm CONSENSUS solves BFT-CUP, in spite of* $f < \frac{k}{2} < n$ *failures, if k-OSR PD is used and assuming at least* $3f+1$ *participants in the sink.*

Sketch of Proof. In this proof we have to consider two cases:
Processes in the sink: All correct participants in the sink component determine that they belong to the sink (Lemma 2) (line 12) and start the execution of an underlying classical Byzantine consensus algorithm (line 13). Then, as the sink has at least $2f+1$ correct nodes, it is guaranteed that all properties of the classical consensus will be met, i.e., *validity*, *integrity*, *agreement* and *termination*. Thus, nodes in the sink obtain the decision value (line 17), send this value to other participants (line 18) and return the decided value to the application (line 19), ensuring *termination*. Whenever a process in the sink receives a request for decision from other processes (lines 20–24), it will send the value if it has already decided (line 23); otherwise, it will store the sender's identity in order to send the decision value later (line 18) after the consensus has been achieved.

Processes not in the sink: Processes not in the sink request the decision value to all participants in the sink (line 15). Notice that if there is enough connectivity ($k \geq 2f+1$), nodes in the sink are reachable from any node of the system. Moreover, by properties of the reachable reliable broadcast, all correct participant in the sink will receive requests sent by correct participants not in the sink, even in the presence of $f < \frac{k}{2}$ failures (lines 20–24). Thus, as there are at least $2f+1$ correct participants in the sink able to send back replies for these requests (lines 18, 23), it is guaranteed that nodes not in the sink will receive at least $f+1$ messages with the same decision value (lines 25-31) and the predicate of line 27 will be true, allowing the process to *terminate* and return the decided value (line 28). Moreover, a collusion of up to f malicious participants cannot lead a process to decide for incorrect values (line 27), guaranteeing thus *agreement*. *Integrity* is ensured through the verification of predicate on line 25, by which each correct participant decides only once. Notice that *validity* is ensured through the underlying classical Byzantine consensus protocol, i.e., the decided value is a value proposed by nodes in the sink. This proves that k-OSR PD is sufficient to solve BFT-CUP. □

4.4 Necessity of *k*-OSR Participant Detector to Solve BFT-CUP

Using a k-OSR PD, our protocol requires a degree of connectivity $k \geq 2f+1$ to solve BFT-CUP. Theorem 2 states that a participant detector of this class and this connectivity degree are necessary to solve BFT-CUP.

Theorem 2. *A participant detector PD* $\in$ *k-OSR is necessary to solve BFT-CUP, in spite of* $f < \frac{k}{2} < n$ *failures.*

Sketch of Proof. This proof is based on the same arguments to prove the necessity of OSR (One Sink Reducibility) for solving CUP [6]. Assume by contradiction that there is an algorithm which solves BFT-CUP with a *PD* $\notin$ *k-OSR*. Let G_{di} be the knowledge graph induced by *PD*, then two scenarios are possible: (*i.*) there are less than k node-disjoint paths connecting a participant p in G_{di}; or (*ii.*) the directed acyclic graph obtained by reduction of G_{di} to its k-strongly connected components has at least two sinks. There are two possible scenarios to be considered.

In the first scenario, let at most $2f$ node-disjoint paths connect p in G_{di}. Then, the simple crash failure of f neighbors of p makes it impossible for a participant i (being p reachable from i) to discover p, because only f processes are able to inform i about the presence of p in the system. In fact, i is not able to determine if p really exists, i.e., it is not guaranteed that i has received this information from a correct process. Then, the partial view obtained by i will be inconsistent, what makes it impossible to solve BFT-CUP. Thus, we reach a contradiction.

In the second scenario, let G_1 and G_2 be two of the sink components and consider that participants in G_1 have proposition value v and participants in G_2 value w, with $v \neq w$. By *Termination* property of consensus, processes in G_1 and G_2 must eventually decide. Let us assume that the first process in G_1 that decides, say p, does so at time t_1, and the first process in G_2 that decides, say q, does so at time t_2. Delay all messages sent to G_1 and G_2 such that they are received after $max\{t_1, t_2\}$. Since the processes in a sink component are unaware of the existence of other participants, p decides v and q decides w, violating the *Agreement* property of consensus and reaching thus a contradiction. □

5 Discussion

This section presents some comments about the protocol presented in this paper.

5.1 Digital Signatures

It is worth to notice that the lower bound required to solve BFT-CUP in terms of connectivity and resiliency is $k \geq 2f+1$, and it holds even if digital signatures are used. By using digital signatures, it is possible to exchange messages among participants, since there is at least one path formed only by correct processes ($k \geq f+1$). However, even with digital signatures, a connectivity of $k \geq 2f+1$ is still required in order to discover the participants properly (first phase of the protocol). In fact, if $k < 2f+1$, a malicious participant can lead a correct participant p not to discover every node reachable from it, what makes it impossible to use this protocol to solve BFT-CUP (the partial view of p will be inconsistent).

For example, Figure 2 presents a knowledge connectivity graph induced by a 2-OSR PD ($k = 2$) in which the system does not support any fault (to support $f = 1$, $k \geq 3$). Now, consider that process 2 is malicious and that process 1 is starting the DISCOVERY phase. Then, process 2 could inform to process 1 that it only knows process 3. At this point, process 1 will break the search because it is only waiting for a message from process 3, i.e., number of pending messages less or equal to f. Thus, process 1 obtains the wrong partial view $\{1,2,3\}$ of the system.

5.2 Protocol Limitations

The model used in this study, as well as in all solutions for FT-CUP [7,8], supports mobility of nodes, but it is not strong enough to tolerate arbitrary churn (arrivals and

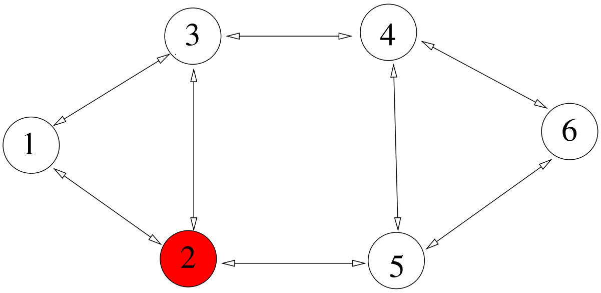


Fig. 2. 2-OSR with Process 2 Faulty

departures of processes) during protocol executions. This happens because, after the relations of knowledge have been established (first phase of the protocol), new participants will be considered only in future executions of consensus.

In current algorithms, process departures can be considered as failures. Nonetheless, this is not the optimal approach, since our protocols tolerate Byzantine faults and the behaviour of a departing process resembles a simple crash failure. An alternative approach consists in specifying an additional parameter d to indicate the number of supported departures, separating departures from malicious faults. In this way, the degree of connectivity in the knowledge graph should be $k \geq 2f + d + 1$ to support up to f malicious faults and up to d departures. Moreover, even with departures, the sink component should remains with enough participants to execute a classical consensus, i.e., $n_{sink} \geq 3f + 2d + 1$, following the same reasoning as [19].

5.3 Other Participant Detectors

Although k-OSR PD is the weakest participant detector defined to solve FT-CUP, there are other (stronger) participant detectors able to solve BFT-CUP [6,8]:

- FCO (Full Connectivity PD): the knowledge connectivity graph $G_{di} = (V, \xi)$ induced by the PD oracle is such that for all $p, q \in \Pi$, we have $(p, q) \in \xi$.
- k-SCO (k-Strong Connectivity PD): the knowledge connectivity graph $G_{di} = (V, \xi)$ induced by the PD oracle is k-strongly connected.

Notice that a characteristic common to all participant detectors able to solve BFT-CUP (except for the FCO PD that is fully connected) is the degree of connectivity k, which makes possible the proper work of the protocol even in the presence of failures. Using these participant detectors (FCO or k-SCO) the partial view obtained by each process in the system contains exactly all processes in the system (first phase of the protocol). Thereafter, the consensus problem is trivially solved using a classical Byzantine consensus protocol, since all processes have the same (complete) view of the system.

6 Final Remarks

Most of the studies about consensus found in the literature consider a static known set of participants in the system (e.g., [1,3,4,5,17,19]). Recently, some works which

Table 1. Comparing solutions for the consensus with unknown participants problem

Approach	failure model	participant detector	k	participants in the sink	connectivity between components	synchrony model
CUP [6]	without failures	OSR	–	1	OSR	asynchronous
FT-CUP [7]	crash	OSR	–	1	OSR + safe crash pattern	asynchronous + $\mathscr{P}$
FT-CUP [8]	crash	k-OSR	$f+1$	$2f+1$	k node-disjoint paths	asynchronous + $\Diamond\mathscr{S}$
BFT-CUP (this paper)	Byzantine	k-OSR	$2f+1$	$3f+1$	k node-disjoint paths	same of the underlying consensus protocol

deal with a partial knowledge about the system composition have been proposed. The works of [6,7,8] are worth noticing. They propose solutions and study conditions in order to solve consensus whenever the set of participants is unknown and the system is asynchronous. The work presented herein extends these previous results and presents an algorithm for solving FT-CUP in a system prone to Byzantine failures. It shows that to solve Byzantine FT-CUP in an environment with little synchrony requirements, it is necessary to enrich the system with a greater degree of knowledge connectivity among its participants. The main result of the work is to show that it is possible to solve Byzantine FT-CUP with the same class of participant detectors (k-OSR) and the same synchrony requirements ($\Diamond\mathscr{S}$) necessary to solve FT-CUP in a system prone to crash failures [8]. As a side effect, a Byzantine fault-tolerant dissemination primitive, namely *reachable reliable broadcast*, has been defined and implemented and can be used in other protocols for unknown networks.

Table 1 summarizes and presents a comparison with the known results regarding the consensus solvability with unknown participants.

Acknowledgements

Eduardo Alchieri is supported by a CAPES/Brazil grant. Joni Fraga and Fabíola Greve are supported by CNPq/Brazil grants. This work was partially supported by the EC, through projects IST-2004-27513 (CRUTIAL), by the FCT, through the Multiannual (LaSIGE) and the CMU-Portugal Programmes, and by CAPES/GRICES (project TISD).

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programing Languages and Systems 4(3), 382–401 (1982)
2. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
3. Toueg, S.: Randomized Byzantine Agreements. In: Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, pp. 163–178 (1984)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)

5. Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. The Computer Journal 49(1) (2006)
6. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
7. Cavin, D., Sasson, Y., Schiper, A.: Reaching agreement with unknown participants in mobile self-organized networks in spite of process crashes. Technical Report IC/2005/026, EPFL - LSR (2005)
8. Greve, F.G.P., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: Proceedings of the International Conference on Dependable Systems and Networks - DSN, pp. 82–91 (2007)
9. Castro, M., Liskov, B.: Practical Byzantine fault-tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
10. Douceur, J.: The sybil attack. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (2002)
11. Awerbuch, B., Holmer, D., Nita-Rotaru, C., Rubens, H.: An on-demand secure routing protocol resilient to byzantine failures. In: Proceedings of the 1st ACM workshop on Wireless security - WiSE, pp. 21–30. ACM, New York (2002)
12. Kotzanikolaou, P., Mavropodi, R., Douligeris, C.: Secure multipath routing for mobile ad hoc networks. In: Wireless On-demand Network Systems and Services - WONS, pp. 89–96 (2005)
13. Papadimitratos, P., Haas, Z.: Secure routing for mobile ad hoc networks. In: Proceedings of SCS Communication Networks and Distributed Systems Modeling and Simulation Conference - CNDS (2002)
14. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of ACM 35(2), 288–322 (1988)
15. Bracha, G.: An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In: Proceedings of the 3rd ACM symposium on Principles of Distributed Computing, pp. 154–162 (1984)
16. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In: Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, pp. 27–30 (1983)
17. Friedman, R., Mostefaoui, A., Raynal, M.: Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. IEEE Transactions on Dependable and Secure Computing 2(1), 46–56 (2005)
18. Dolev, D.: The Byzantine generals strike again. Journal of Algorithms (3), 14–30 (1982)
19. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing 3(3), 202–215 (2006)

With Finite Memory
Consensus Is Easier Than Reliable Broadcast

Carole Delporte-Gallet[1], Stéphane Devismes[2], Hugues Fauconnier[1],
Franck Petit[3,⋆], and Sam Toueg[4]

[1] LIAFA, Université D. Diderot, Paris, France
{cd,hf}@liafa.jussieu.fr
[2] VERIMAG, Université Joseph Fourier, Grenoble, France
stephane.devismes@imag.fr
[3] INRIA/LIP Laboratory,Univ. of Lyon/ENS Lyon, Lyon, France
franck.petit@ens-lyon.fr
[4] Department of Computer Science, University of Toronto, Toronto, Canada
sam@cs.toronto.edu

Abstract. We consider asynchronous distributed systems with message losses and process crashes. We study the impact of finite process memory on the solution to *consensus*, *repeated consensus* and *reliable broadcast*. With finite process memory, we show that in some sense consensus is easier to solve than reliable broadcast, and that reliable broadcast is as difficult to solve as repeated consensus: More precisely, with finite memory, consensus can be solved with failure detector $\mathcal{S}$, and $\mathcal{P}^-$ (a variant of the perfect failure detector which is stronger than $\mathcal{S}$) is necessary and sufficient to solve reliable broadcast and repeated consensus.

1 Introduction

Designing fault-tolerant protocols for asynchronous systems is highly desirable but also highly complex. Some classical agreement problems such as *consensus* and *reliable broadcast* are well-known tools for solving more sophisticated tasks in faulty environments (e.g., [1,2]). Roughly speaking, with consensus processes must reach a common decision on their inputs, and with reliable broadcast processes must deliver the same set of messages.

It is well known that consensus cannot be solved in asynchronous systems with failures [3], and several mechanisms were introduced to circumvent this impossibility result: *randomization* [4], *partial synchrony* [5,6] and *(unreliable) failure detectors* [7].

Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about the process crashes. Each process can access a local failure detector module that monitors the processes of the system and maintains a list of processes that are suspected of having crashed.

[⋆] This work was initiated while Franck Petit was with MIS Lab., Université of Picardie, France. Research partially supported by Région Picardie, Proj. APREDY.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 41–57, 2008.

Several classes of failure detectors have been introduced, *e.g.*, $\mathcal{P}$, $\mathcal{S}$, Ω, etc. Failure detectors classes can be compared by reduction algorithms, so for any given problem P, a natural question is "*What is the weakest failure detector (class) that can solve P ?*". This question has been extensively studied for several problems in systems *with infinite process memory* (*e.g.*, uniform and non-uniform versions of consensus [8,9,10], non-blocking atomic commit [11], uniform reliable broadcast [12,13], implementing an atomic register in a message-passing system [11], mutual exclusion [14], boosting obstruction-freedom [15], set consensus [16,17], etc.). This question, however, has not been as extensively studied in the context of systems *with finite process memory.*

In this paper, we consider systems where processes have finite memory, processes can crash and links can lose messages (more precisely, links are fair lossy and FIFO[1]). Such environments can be found in many systems, for example in sensor networks, sensors are typically equipped with small memories, they can crash when their batteries run out, and they can experience message losses if they use wireless communication.

In such systems, we consider (the uniform versions of) reliable broadcast, consensus and repeated consensus. Our contribution is threefold: First, we establish that the weakest failure detector for reliable broadcast is $\mathcal{P}^-$ — a failure detector that is almost as powerful than the perfect failure detector $\mathcal{P}$. Next, we show that consensus can be solved using failure detector $\mathcal{S}$. Finally, we prove that $\mathcal{P}^-$ is the weakest failure detector for repeated consensus. Since $\mathcal{S}$ is strictly weaker than $\mathcal{P}^-$, in some precise sense these results imply that, in the systems that we consider here, consensus is easier to solve than reliable broadcast, and reliable broadcast is as difficult to solve as repeated consensus.

The above results are somewhat surprising because, when processes have infinite memory, reliable broadcast is easier to solve than consensus[2], and repeated consensus is not more difficult to solve than consensus.

Roadmap. The rest of the paper is organized as follows: In the next section, we present the model considered in this paper. In Section 4, we show that in case of process memory limitation and possibility of crashes, $\mathcal{P}^-$ is necessary and sufficient to solve reliable broadcast. In Section 5, we show that consensus can be solved using a failure detector of type $\mathcal{S}$ in our systems. In Section 6, we show that $\mathcal{P}^-$ is necessary and sufficient to solve repeated consensus in this context.

For space considerations, all the proofs are omitted, see the technical report for details ([20], `http://hal.archives-ouvertes.fr/hal-00325470/fr/`).

[1] The FIFO assumption is necessary because, from the results in [18], if lossy links are not FIFO, reliable broadcast requires unbounded message headers.

[2] With infinite memory and fair lossy links, (uniform) reliable broadcast can be solved using Θ [19], and Θ is strictly weaker than (Σ, Ω) which is necessary to solve consensus.

2 Model

Distributed System. A system consists of a set $\Pi = \{p_1, ..., p_n\}$ of processes. We consider *asynchronous* distributed systems where each process can communicate with each other through *directed links*.[3] By asynchronous, we mean that there is no bound on message delay, clock drift, or process execution rate.

A process has a local memory, a local sequential and deterministic algorithm, and input/output capabilities. In this paper we consider systems of processes having either a *finite* or an *infinite* memory. In the sequel, we denote such systems by $\Phi^{\mathcal{F}}$ and $\Phi^{\mathcal{I}}$, respectively.

We consider links with unbounded capacities. We assume that the messages sent from p to q are *distinguishable*, *i.e.*, if necessary, the messages can be numbered with a non-negative integer. These numbers are used for notational purpose only, and are unknown to the processes. Every link satisfies the *integrity*, *i.e.*, if a message m from p is received by q, m is received by q at most once, and only if p previously sent m to q. Links are also *unreliable* and *fair*. Unreliable means that the messages can be lost. Fairness means that for each message m, if process p sends infinitely often m to process q and if q tries to receive infinitely often a message from p, then q receives infinitely often m from p. Each link are *FIFO*, *i.e.*, the messages are received in the same order as they were sent.

To simplify the presentation, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

Failures and Failure Patterns. Every process can fail by permanently *crashing*, in which case it definitively stops to execute its local algorithm. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once crashed, a process never recoves, *i.e.*, $\forall t : F(t) \subseteq F(t{+}1)$. We define $crashed(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $correct(F) = \Pi \backslash crashed(F)$. If $p \in crashed(F)$ we say that p *crashes in* F (or simply *crashed* when it is clear in the context) and if $p \in correct(F)$ we say that p is *correct in* F (or simply *correct* when it is clear in the context). An environment is a set of failure patterns. We do not restrict here the number of crash and we consider as environment $\mathcal{E}$ the set of all failure patterns.

Failure Detectors. A failure detector [7] is a local module that outputs a set of processes that are currently suspected of having crashed. A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}. $H(p,t)$ is the value of the failure detector module of process p at time t. If $q \in H(p,t)$, we say that p *suspects* q *at time* t *in* H. We omit references to H when it is obvious from the context.

Formally, *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories $\mathcal{D}(F)$.

A failure detector can be defined in terms of two *abstract properties*: *Completeness* and *Accuracy* [7] . Let us recall one type of *completeness* and two types of *accuracy.*

[3] We assume that each process knows the set of processes that are in the system; some papers related to failure detectors do not make this assumption e.g. [21,22,23].

Definition 1 (Strong Completeness). *Eventually every process that crashes is permanently suspected by* every correct process. *Formally, $\mathcal{D}$ satisfies strong completeness if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \forall q \in \mathit{correct}(F), \forall t' \geq t : p \in H(q, t')$

Definition 2 (Strong Accuracy). *No process is suspected before it crashes. Formally, $\mathcal{D}$ satisfies strong accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in \Pi \setminus F(t) : p \notin H(q, t)$

Definition 3 (Weak Accuracy). *A correct process is never suspected. Formally, $\mathcal{D}$ satisfies weak accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t {\in} \mathcal{T}, \exists p \in \mathit{correct}(F), \forall q \in \Pi : p \notin H(q, t)$

We introduce a last type of accuracy:

Definition 4 (Almost Strong Accuracy). *No correct process is suspected. Formally, $\mathcal{D}$ satisfies almost strong accuracy if:* $\forall F \in \mathcal{E}, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p \in \mathit{correct}(F), \forall q \in \Pi : p \notin H(q, t)$

This definition was the definition of strong accuracy in [24].

For all these aformentioned properties, we can assume, without loss of generality, that when a process is suspected it remains suspected forever.

We now recall the definition of the perfect and the strong failure detectors [7] and we introduce our almost perfect failure detector:

Definition 5 (Perfect). *A failure detector is said to be* perfect *if it satisfies the strong completeness and the strong accuracy properties. This failure detector is denoted by $\mathcal{P}$.*

Definition 6 (Almost Perfect). *A failure detector is said to be* almost perfect *if it satisfies the strong completeness and the almost strong accuracy properties. This failure detector is denoted by $\mathcal{P}^-$.*

Note that $\mathcal{P}-$ was given as the definition of the perfect failure detector in the very first paper on unreliable failure detector in [24]. In fact, failure detector in $\mathcal{P}^-$ can suspect faulty processes before they crash and be *unrealistic* according to the definition in [25].

Definition 7 (Strong). *A failure detector is said to be* strong *if it satisfies the strong completeness and the weak accuracy properties. This failure detector is denoted by $\mathcal{S}$.*

Algorithms, Runs, and Specification. A distributed algorithm is a collection of n sequential and deterministic algorithms, one for each process in Π. Computations of distributed algorithm $\mathcal{A}$ proceed in *atomic steps.*

In a step, a process p executes each of the following actions at most once: p try to receive a message from another process, p queries its failure detector module, p modifies its (local) state. and p sends a message to another process.

A *run* of Algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ is a tuple $\langle F, H_{\mathcal{D}}, \gamma_{init}, E, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is an history of failure detector $\mathcal{D}$ for the failure pattern F, γ_{init} is an *initial configuration* of $\mathcal{A}$, E is an infinite sequence of steps of A, and T is a list of increasing time values indicating when each step in E occurred. A run must satisfy certain well-formedness and fairness properties. In particular:

1. E is applicable to γ_{init}.
2. A process cannot take steps after it crashes.
3. When a process takes a step and queries its failure detector module, it gets the current value output by its local failure detector module.
4. Every correct process takes an infinite number of local steps in E.
5. Any message sent is eventually received or lost.

A *problem* P is defined by a set of properties that runs must satisfy. An algorithm A *solves a problem* P using a failure detector $\mathcal{D}$ if and only if all the runs of A using $\mathcal{D}$ satisfy the properties required by P.

A failure detector $\mathcal{D}$ is said to be *weaker* than another failure detector $\mathcal{D}'$ (denote $\mathcal{D} \leq \mathcal{D}'$) if there is an algorithm that uses only $\mathcal{D}'$ to emulate the output of $\mathcal{D}$ for every failure pattern. If $\mathcal{D}$ is weaker than $\mathcal{D}'$ but $\mathcal{D}'$ is not weaker than $\mathcal{D}$ we say that $\mathcal{D}$ is *strictly weaker* than $\mathcal{D}'$ (denote $\mathcal{D} < \mathcal{D}'$).

From [7] and our definition of $\mathcal{P}^-$, we get:

Proposition 1

$$\mathcal{S} < \mathcal{P}^- < \mathcal{P}$$

The *weakest* [8] failure detector $\mathcal{D}$ to solve a given problem is a failure detector $\mathcal{D}$ that is sufficient to solve the problem and that is also necessary to solve the problem, i.e. $\mathcal{D}$ is weaker than any failure detector that solves the problem.

Notations. In the sequel, v_p denotes the value of the variable v at process p. Finally, a datum in a message can be replaced by "$-$" when this value has no impact on the reasonning.

3 Problem Specifications

Reliable Broadcast. The reliable broadcast [26] is defined with two primitives: `BROADCAST`(m) and `DELIVER`(m). Informally, any reliable broadcast algorithm guarantees that after a process p invokes `BROADCAST`(m), every correct process eventually executes `DELIVER`(m). In the formal definition below, we denote by *sender*(m) the process that invokes `BROADCAST`(m).

Specification 1 (Reliable Broadcast). *A run R satisfies the specification Reliable Broadcast if and only if the following three requirements are satisfied in R:*

- Validity: *If a correct process invokes* `BROADCAST`(m)*, then it eventually executes* `DELIVER`(m)*.*

- (Uniform) Agreement: *If a process executes* DELIVER(m)*, then all other correct processes eventually execute* DELIVER(m)*.*
- Integrity: *For every message* m*, every process executes* DELIVER(m) *at most once, and only if sender*(m) *previously invokes* BROADCAST(m)*.*

Consensus. In the consensus problem, all correct processes *propose* a value and must reach a unanimous and irrevocable *decision* on some value that is chosen between the proposed values. We define the consensus problem in terms of two primitives, PROPOSE(v) and DECIDE(u). When a process executes PROPOSE(v), we say that it *proposes* v; similarly, when a process executes DECIDE(u), we say that it *decides* u.

Specification 2 (Consensus). *A run R satisfies the specification Consensus if and only if the following three requirements are satisfied in R:*

- (Uniform) Agreement: *No two processes* decide *differently.*
- Termination: *Every correct process eventually* decides *some value.*
- Validity: *If a process* decides v*, then* v *was* proposed *by some process.*

Repeated Consensus. We now define repeated consensus. Each correct process has as input an infinite sequence of proposed values, and outputs an infinite sequence of decision values such that:

1. Two correct processes have the same output. (The output of a faulty process is a prefix of this output.)
2. The i^{th} value of the output is the i^{th} value of the input of some process.

We define the repeated consensus in terms of two primitives, R-PROPOSE(v) and R-DECIDE(u). When a process executes the i^{th} R-PROPOSE(v), v is the i^{th} value of its input (we say that it *proposes* v for the i^{th} consensus); similarly, when a process executes the i^{th} R-DECIDE(u) u is the i^{th} value of its output (we say that it *decides* v for the i^{th} consensus).

Specification 3 (Repeated Consensus). *A run R satisfies the specification Repeated Consensus if and only if the following three requirements are satisfied in R:*

- Agreement: *If* u *and* v *are the outputs of two processes, then* u *is a prefix of* v *or* v *is a prefix of* u*.*
- Termination: *Every correct process has an infinite output.*
- Validity: *If the* i^{th} *value of the output of a process is* v*, then* v *is the* i^{th} *value of the input of some process.*

4 Reliable Broadcast in $\Phi^{\mathcal{F}}$

In this section, we show that $\mathcal{P}^{-}$ is the weakest failure detector to solve the reliable broadcast in $\Phi^{\mathcal{F}}$.

$\mathcal{P}^-$ is Necessary. To show that $\mathcal{P}^-$ is necessary to solve the reliable broadcast the following lemma is central to the proof:

Lemma 1. *Let $\mathcal{A}$ be an algorithm solving Reliable Broadcast in $\Phi^{\mathcal{F}}$ with a failure detector $\mathcal{D}$. There exists an integer k such that for every process p and every correct process q, for every run R of $\mathcal{A}$ where process p* `BROADCAST`*s and* `DELIVER`*s k messages, at least one message from q has been received by some process.*

Assume now that there exists an algorithm $\mathcal{A}$ that implements the *reliable broadcast* in $\Phi^{\mathcal{F}}$ using the failure detector $\mathcal{D}$. To show our result we have to give an algorithm that uses only $\mathcal{D}$ to emulate the output of $\mathcal{P}^-$ for every failure pattern.

Actually, we give an algorithm $\mathcal{A}_{(p,q)}$ (Figure 1) where a given process p monitors a given process q. This algorithm uses one instance of $\mathcal{A}$ with $\mathcal{D}$. Note that all processes except q participate to this algorithm following the code of $\mathcal{A}$. In this algorithm *Output_q* is equal to either $\{q\}$ (q is faulty) or $\emptyset$ (q is correct).

The algorithm $\mathcal{A}_{(p,q)}$ works as follows: p tries to `BROADCAST` k messages, all processes execute the code of the algorithm $\mathcal{A}$ using $\mathcal{D}$ except q that does nothing. If p `DELIVER`s k messages, it sets *Output_q* to q and never changes the values of *Output_q*. By lemma 1, if q is correct p can't `DELIVER` k messages and so it never sets *Output_q* to $\{q\}$. If q is faulty and p is correct: as $\mathcal{A}$ solve reliable broadcast, p has to deliver `DELIVER` k messages and so p sets *Output_q* to $\{q\}$.[4]

To emulate $\mathcal{P}^-$, each process p uses algorithm $\mathcal{A}_{(p,q)}$ for every process q. As $\mathcal{D}$ is a failure detector it can be used for each instance. The output of $\mathcal{P}^-$ at p (variable *Output*) is then the union of *Output_q* for every process q.

```
        /* CODE FOR PROCESS $p$ */
begin
    $Output\_q \leftarrow \emptyset$
    for $i = 1$ to $k$ do
        BROADCAST($m$)        /* using $\mathcal{A}$ with $\mathcal{D}$ */
        wait for DELIVER($m$)
    end for
    $Output\_q \leftarrow \{q\}$
end
        /* CODE FOR PROCESS $q$ */
begin
end
        /* CODE FOR EVERY PROCESS $\Pi - \{p, q\}$ */
begin
    execute the code of $\mathcal{A}$ with $\mathcal{D}$ for these messages
end
```

Fig. 1. $\mathcal{A}_{(p,q)}$

Theorem 1. *$\mathcal{P}^-$ is necessary to solve Reliable Broadcast in $\Phi^{\mathcal{F}}$.*

$\mathcal{P}^-$ is Sufficient. In Algorithm $\mathcal{B}$ (Figure 2), every process uses a failure detector module of type $\mathcal{P}^-$ and a finite memory. Theorem 2 shows that Algorithm $\mathcal{B}$ solves the reliable broadcast in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{P}^-$ is sufficient to solve the reliable broadcast in $\Phi^{\mathcal{F}}$ (Corollary 1).

[4] If q is faulty and p is faulty, the property of failure detector is trivially ensured.

```
    /* CODE FOR EVERY PROCESS $q$ */
variables:
    Flag$[1 \dots n][1 \dots n] \in \{0,1\}^{n^2}$; $\forall (i,j) \in \Pi^2$, Flag$[i][j]$ is initialized to 0
    FD: failure detector of type $\mathcal{P}^-$
    Mes$[1 \dots n]$: array of data messages; $\forall i \in \Pi$, Mes$[i]$ is initialized to $\perp$
function:
    MesToBrd(): returns a message or $\perp$
begin
  repeat forever
    if Mes$[p] = \perp$ then
      Mes$[p] \leftarrow$ MesToBrd()
      if Mes$[p] \neq \perp$ then
        Flag$[p][p] \leftarrow$ (Flag$[p][p] + 1$) mod 2
      end if
    end if
    for all $i \in \Pi \setminus$ FD do
      for all $j \in \Pi \setminus ($FD$ \cup \{p,i\})$,Flag$[i][p] \neq$Flag$[i][j]$ do
        if (receive$\langle i$-ACK$, F\rangle$ from $j$) $\wedge$ ($F$=Flag$[i][p]$) then
          Flag$[i][j] \leftarrow F$
        else
          send$\langle i$-BRD, Mes$[i]$, Flag$[i][p]\rangle$ to $j$
        end if
      end for
      if (Mes$[i] \neq \perp$) $\wedge$ ($\forall q \in \Pi \setminus$ FD, Flag$[i][i]$ = Flag$[i][q]$) then
        DELIVER(Mes$[i]$); Mes$[i] \leftarrow \perp$
      end if
    end for
    for all $i \in \Pi \setminus$ FD $\setminus \{p\}$ do
      for all $j \in \Pi \setminus ($FD $\cup \{p\})$ do
        if (receive$\langle i$-BRD$, m, F\rangle$ from $j$) then
          if ($\forall q \in \Pi \setminus$ FD, Flag$[i][q]$ = Flag$[i][i]$) $\wedge$ ($F \neq$ Flag$[i][p]$) then
            Mes$[i] \leftarrow m$; Flag$[i][p] \leftarrow F$
          end if
          if $i = j$ then
            Flag$[i][i] \leftarrow F$
          end if
          if ($i \neq j$) $\vee$ ($\forall q \in \Pi \setminus$ FD, Flag$[i][q]$ = Flag$[i][i]$) then
            send$\langle i$-ACK, Flag$[i][p]\rangle$ to $j$
          end if
        end if
      end for
    end for
   end repeat
 end
```

Fig. 2. Algorithm $\mathcal{B}$

In Algorithm $\mathcal{B}$, each process p executes broadcasts sequentially: p starts a new broadcast only after the termination of the previous one. To that goal, any process p initializes Mes$[p]$ to $\perp$. Then, p periodically checks if an external application invokes BROADCAST($-$). In this case, MesToBrd() returns the message to broadcast, say m. When this event occurs, Mes$[p]$ is set to m and the broadcast procedure starts. Mes$[p]$ is set to $\perp$ at the end of the broadcast, p checks again, and so on.

Algorithm $\mathcal{B}$ has to deal with two types of faults: process crashes and message loss.

- *Dealing with process crashes.* Every process uses a failure detector of type $\mathcal{P}^-$ to detect the process crashes. Note that, as mentionned in Section 2, we assume that when a process is suspected by some process it remains suspected forever.

Assume that a process p broadcasts the message m: p sends a broadcast message (p-BRD) with the datum m to any process it believes to be correct.

In Algorithm $\mathcal{B}$, p executes DELIVER(m) only after all other processes it does not suspect receive m. To that goal, we use acknowledgment mechanisms. When p received an acknowledgment for m (p-ACK) from every other process it does not suspect, p executes DELIVER(m) and the broadcast of m terminates (*i.e.*, Mes[p] is set to $\perp$).

To ensure the *agreement* property, we must guarantee that if p crashes but another process q already executes DELIVER(m), then any correct process eventually executes DELIVER(m). To that goal, any process can execute DELIVER(m) only after all other processes it does not suspect except p receive m. Once again, we use acknowledgment mechanisms to that end: q also broadcasts m to every other process it does not suspect except p (this induces that a process can now receive m from a process different of p) until receiving an acknowledgment for m from all these processes and the broadcast message from p if q does not suspect it.

To prevent m to be still broadcasted when p broadcasts the next message, we synchronize the system as follows: any process acknowledges m to p only after it received an acknowledgment for m from every other process it does not suspect except p. By contrast, if a process i receives a message broadcasted by p (p-BRD) from the process $j \neq p$, i directly acknowledges the message to j.

- *Dealing with message loss.* The broadcast messages have to be periodically retransmitted until they are acknowledged. To that goal, any process q stores the last broadcasted message from p into its variable $\texttt{Mes}_q[p]$ (initialized to $\perp$). However, some copies of previously received messages can be now in transit at any time in the network. So, each process must be able to distinguish if a message it receives is copy of a previously received message or a new one, say *valid*. To circumvent this problem, we use the traditional alternating-bit mechanism [27,28]: a flag value (0 or 1) is stored into any message and a two-dimentionnal array, noted Flag$[1 \ldots n][1 \ldots n]$, allows us to distinguish if the messages are *valid* or not. Initially, any process sets Flag$[i][j]$ to 0 for all pairs of processes (i,j). In the code of process p, the value $\texttt{Flag}_p[p][p]$ is used to mark every p-BRD messages sent by p. In the code of every process $q \neq p$, $\texttt{Flag}_q[p][q]$ is equal to the flag value of the last valid p-BRD message q receives (not necessarily from p). For all $q' \neq q$, $\texttt{Flag}_q[p][q']$ is equal to the flag value of the last valid p-BRD message q receives from q'.

  At the beginning of any broadcast at p, p increments $\texttt{Flag}_p[p][p]$ modulus 2. The broadcast terminates at p when for every other process q that p does not suspect, $\texttt{Flag}_p[p][q] = \texttt{Flag}_p[p][p]$, $\texttt{Flag}_p[p][q]$ being set to $\texttt{Flag}_p[p][p]$ only when p received a valid acknowledgement from q, *i.e.*, an acknowledgment marked with the value $\texttt{Flag}_p[p][p]$.

  Upon receiving a p-BRD message marked with the value F, a process $q \neq p$ detects that it is a new valid message broadcasted by p (but not necessarily sent by p) if for every non-suspected process j, ($\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) and ($F \neq \texttt{Flag}_q[p][q]$). In this case, p sets $\texttt{Mes}_q[p]$ to m and sets $\texttt{Flag}_q[p][q]$ to F. From this point on, q periodically sends $\langle p\text{-}\texttt{BRD}, \texttt{Mes}_q[p], \texttt{Flag}_q[p][q]\rangle$ to any other process it does not suspect except p until receiving a valid acknowledgment (*i.e.*, an acknowledgment marked with the value $\texttt{Flag}_q[p][q]$) from all these processes. For any non-suspected process j different from p and q, $\texttt{Flag}_q[p][j]$ is set to $\texttt{Flag}_q[p][q]$ when q received an acknowledgment marked with the value $\texttt{Flag}_q[p][q]$ from j. Finally, $\texttt{Flag}_q[p][p]$ is set to $\texttt{Flag}_q[p][q]$ when q received the broadcast message from p (marked with the value $\texttt{Flag}_q[p][q]$). Hence, q can execute $\texttt{DELIVER}(\texttt{Mes}[p])$ when ($\texttt{Mes}[p] \neq \perp$) and ($\forall j \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) because (1) it receives a valid broadcast message from p if p was not suspected and it has the guarantee that any non-suspected process different of p receives m in a valid message. To ensure that q executes $\texttt{DELIVER}(\texttt{Mes}[p])$ at most one, q just has to set $\texttt{Mes}[p]$ to $\perp$ after.

It is important to note that q acknowledges the valid p-$\texttt{BRD}$ messages it receives from p only when the predicate ($\forall j \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_q[p][j] = \texttt{Flag}_q[p][p]$) holds. However, to guarantee the liveness, q acknowledges any p-$\texttt{BRD}$ message that it receives from any other process. Every p-$\texttt{ACK}$ messages sent by q is marked with the value $\texttt{Flag}_q[p][q]$.

Finally, p stops its current broadcast when the following condition holds: ($\texttt{Mes}_p[p] \neq \perp$) $\wedge$ ($\forall q \in \Pi \setminus \texttt{FD}$, $\texttt{Flag}_p[p][p] = \texttt{Flag}_p[p][q]$), *i.e.*, any non-suspected process has acknowledged $\texttt{Mes}_p[p]$. In this case, p sets $\texttt{Mes}[p]$ to $\perp$.

Theorem 2. *Algorithm $\mathcal{B}$ is a Reliable Broadcast algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{P}^-$.*

Corollary 1. *$\mathcal{P}^-$ is sufficient for solving Reliable Broadcast in $\Phi^{\mathcal{F}}$.*

5 Consensus in $\Phi^{\mathcal{F}}$

In this section, we show that we can solve consensus in system $\Phi^{\mathcal{F}}$ with a failure detector that is strictly weaker than the failure detector necessary to solve reliable broadcast and repeated consensus. We solve consensus with the strong failure detector $\mathcal{S}$. $\mathcal{S}$ is not the weakest failure detector to solve consensus whatever the number of crash but it is strictly weaker than $\mathcal{P}^-$ and so enough to show our results.

We customize the algorithm of Chandra and Toueg [7] that works in an asynchronous message-passing system with reliable links and augmented with a strong failure detector ($\mathcal{S}$), to our model.

In this algorithm, called $\mathcal{CS}$ in the following (Figure 3), the processes execute n asynchronous rounds. First, processes execute $n-1$ asynchronous rounds (r denotes the current round number) during which they broadcast and relay their proposed values. Each process p waits until it receives a round r message from every other non-suspected process (*n.b.* as mentionned in Section 2, we assume that when a process is suspected it remains suspected forever) before proceeding to round $r+1$. Then, processes execute a last asynchronous round during which they eliminate some proposed values. Again each process p waits until it receives a round n message from every other process it does not suspected. By the end of these n rounds, correct processes *agree* on a vector based on the proposed values of all processes. The i^{th} element of this vector either contains the proposed value of process i or $\perp$. This vector contains the proposed value of *at least one process*: a process that is never suspected by all processes. Correct processes decide the first nontrivial component of this vector.

To customize this algorithm to our model, we have to ensure the progress of each process: While a process has not ended the asynchronous round r it must be able to retransmit all the messages[5] that it has previously sent in order to allow others processes to progress despite the link failures. As we used a strong failure detector and unreliable but fair links, it is possible that one process has decided and the other ones still wait messages of asynchronous rounds. When a process has terminated the n asynchronous rounds, it uses a special `Decide` message to allow others processes to progress.

In the algorithm, we first use a function *consensus*. This function takes the proposed value in parameter and returns the decision value and uses a failure detector. Then, at processes request, we propagate the `Decide` message.

Theorem 3 shows that Algorithm $\mathcal{CS}$ solves the consensus in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{S}$ is sufficient to solve te consensus problem in $\Phi^{\mathcal{F}}$ (Corollary 2).

Theorem 3. *Algorithm $\mathcal{CS}$ is a Consensus algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{S}$.*

Corollary 2. *$\mathcal{S}$ is sufficient for solving Consensus in $\Phi^{\mathcal{F}}$.*

6 Repeated Consensus in $\Phi^{\mathcal{F}}$

We show in this section that $\mathcal{P}^-$ is the weakest failure detector to solve the reliable consensus problem in $\Phi^{\mathcal{F}}$.

$\mathcal{P}^-$ is Necessary. The proof is similar to the one in Section 4, and here the following lemma is central to the proof:

Lemma 2. *Let $\mathcal{A}$ be an algorithm solving Repeated Consensus in $\Phi^{\mathcal{F}}$ with a failure detector $\mathcal{D}$. There exists an integer k such that for every process p and every correct process q for every run R of $\mathcal{A}$ where process p* `R-PROPOSE`*s and* `R-DECIDE`*s k times, at least one message from q has been received by some process.*

Assume that there exists an algorithm $\mathcal{A}$ that implements *Repeated Consensus* in $\Phi^{\mathcal{F}}$ using the failure detector $\mathcal{D}$. To show our result we have to give an algorithm that uses only $\mathcal{D}$ to emulate the output of $\mathcal{P}^-$ for every failure pattern.

In fact we give an algorithm $\mathcal{A}_q$ (Figure 4) where processes monitor a given process q. This algorithm uses one instance of $\mathcal{A}$ with $\mathcal{D}$. Note that all processes except q participate to this algorithm following the code of $\mathcal{A}$. In this algorithm *Output_q* is equal to either $\{q\}$ (q is crashed) or $\emptyset$ (q is correct).

[5] Notice that they are in finite number.

```
    /* CODE FOR PROCESS $p$ */
function consensus($v$) with the failure detector fd
    variables:
        Flag$[1 \dots n] \in \{true, false\}^n$; $\forall i \in \Pi$, Flag$[i]$ is initialized to false
        V$[1 \dots n]$: array of propositions; $\forall i \in \Pi$, V$[i]$ is initialized to $\perp$
        Mes$[1 \dots n]$: array of arrays of propositions; $\forall i \in \Pi$, Mes$[i]$ is initialized to $\perp$
        $r$: integer; $r$ is initialized to 1
    begin
        V$[p] \leftarrow v$ the proposed values
        Mes$[1] \leftarrow$ V
        while $(r \leq n)$ do
            send$\langle$R-$r$, Mes$[r]\rangle$ to every process, except $p$
            for all $i \in \Pi \setminus ($fd$ \cup \{p\})$, Flag$[i] = false$ do
                if (receive$\langle$R-$r$, $W\rangle$ from $i$) then
                    Flag$[i] \leftarrow true$
                    if $r < n$ then
                        if V$[i] = \perp$ then
                            V$[i] \leftarrow W[i]$; Mes$[r+1][i] \leftarrow W[i]$
                        end if
                    else
                        if V$[i] \neq \perp$ then
                            V$[i] \leftarrow W[i]$
                        end if
                    end if
                end if
            end for
            for all $i \in \Pi \setminus \{p\}$ do
                if (receive$\langle$R-$x$, $W\rangle$ from $i$), $x < r$ then
                    send$\langle$R-$x$, Mes$[x]\rangle$ to $i$
                end if
            end for
            if $\forall q \in \Pi \setminus ($fd$ \cup \{p\})$, Flag$[q] = true$ then
                if $r < n$ then
                    for all $i \in \Pi$ do
                        Flag$[i] \leftarrow false$
                    end for
                end if
                if $r = n - 1$ then
                    Mes$[n] \leftarrow$ V
                end if
                $r \leftarrow r + 1$
            end if
            for all $i \in \Pi \setminus \{p\}$ do
                if (receive$\langle$Decide, $d\rangle$ from $i$) then
                    $return(d)$
                end if
            end for
        end while
        $d \leftarrow$ the first component of V different from $\perp$; $return(d)$
    end
end function
procedure PROPOSE($v$)
    variables: $u$: integer; FD: failure detector of type $\mathcal{S}$
    begin
        $u \leftarrow$consensus($v$) with FD
        DECIDE($u$)
        repeat forever
            for all $j \in \Pi \setminus \{p\}, x \in \{1, \dots, n\}$ do
                if (receive$\langle$R-$x$, $W\rangle$ from $j$) then
                    send$\langle$Decide, $u\rangle$ to $j$
                end if
            end for
        end repeat
    end
end procedure
```

Fig. 3. Algorithm $\mathcal{CS}$, *Consensus* with $\mathcal{S}$

The algorithm $\mathcal{A}_q$ works as follows: processes try to `R-DECIDE` k times, all processes execute the code of the algorithm $\mathcal{A}$ using $\mathcal{D}$ except q that does nothing. If p `R-DECIDE` k messages, it sets *Output_q* to q and never changes the values of *Output_q*.

By lemma 2, if q is correct p cannot decides k times and so it never sets *Output_q* to q. If q is faulty and p is correct[6]: as $\mathcal{A}$ solve *Repeated Consensus*, p has to `R-DECIDE` k times and so p sets *Output_q* to $\{q\}$.

To emulate $\mathcal{P}^-$, each process p uses Algorithm $\mathcal{A}_q$ for every process q. As $\mathcal{D}$ is a failure detector it can be used for each instance. The output of $\mathcal{P}^-$ at p (variable *Output*) is then the union of *Output_q* for every process q.

```
    /* Code for process $p$ of $\Pi \setminus q$ */
begin
    $Output\_q \leftarrow \emptyset$
    for $i = 1$ to $k$ do
        R-PROPOSED($v$)        /* using $\mathcal{A}$ with $\mathcal{D}$ */
        wait for R-DECIDE($v$)
    end for
    $Output\_q \leftarrow \{q\}$
end
    /* Code for process $q$ */
begin
end
```

Fig. 4. $\mathcal{A}_q$

Theorem 4. *$\mathcal{P}^-$ is necessary to solve Repeated Consensus problem in $\Phi^{\mathcal{F}}$.*

$\mathcal{P}^-$ is Sufficient. In this section, we show that $\mathcal{P}^-$ is sufficient to solve the repeated consensus in $\Phi^{\mathcal{F}}$. To that goal, we consider an algorithm called Algorithm $\mathcal{RCP}$ (Figures 5 and 6). In this algorithm, any process uses a failure detector module of type $\mathcal{P}^-$ (again we assume that since a process is suspected by some process it is suspected forever) and a finite memory. Theorem 5 shows that Algorithm $\mathcal{RCP}$ solves the repeated consensus in $\Phi^{\mathcal{F}}$ and directly implies that $\mathcal{P}^-$ is sufficient to solve the repeated consensus in $\Phi^{\mathcal{F}}$ (Corollary 3).

We assume that each correct processes has an infinite sequence of input and when it terminates `R-PROPOSED`(v) where v is the i^{th} value of its input, it executes `R-PROPOSED`(w) where w is the $(i+1)^{th}$ value of its input.

When a process executes `R-PROPOSED`(v), it first executes a consensus in which it proposes v. The decision of this consensus is then outputted. Then, processes have to avoid that the messages of two consecutive consensus are mixed up. We construct a synchronization barrier. Without message loss, and with a perfect failure detector, it is sufficient that each process waits a `Decide` message from every process trusted by its failure detector module. By FIFO property, no message $\langle$`R-`$x, -\rangle$ sent before this `Decide` message can be received in the next consensus.

[6] If q is faulty and p is faulty, the property of failure detector is trivially ensured.

```
    /* CODE FOR PROCESS $p$ */
function consensus($v$) with the failure detector fd
    variables:
        Flag$[1 \ldots n] \in \{true, false\}^n$; $\forall i \in \Pi$, Flag$[i]$ is initialized to false
        V$[1 \ldots n]$: array of propositions; $\forall i \in \Pi$, V$[i]$ is initialized to $\bot$
        Mes$[1 \ldots n]$: array of arrays of propositions; $\forall i \in \Pi$, Mes$[i]$ is initialized to $\bot$
        $r$: integer; $r$ is initialized to 1
    begin
        V$[p] \leftarrow v$ the proposed values; Mes$[1] \leftarrow$ V
        while ($r \leq n$) do
            send$\langle$R-$r$, Mes$[r]\rangle$ to every process, except $\{p\} \cup$ fd
            for all $i \in \Pi \setminus ($fd$ \cup \{p\})$, Flag$[i] = false$ do
                if (receive$\langle$R-$r$, $W\rangle$ from $i$) then
                    Flag$[i] \leftarrow true$
                    if $r < n$ then
                        if V$[i] = \bot$ then
                            V$[i] \leftarrow W[i]$; Mes$[r+1][i] \leftarrow W[i]$
                        end if
                    else
                        if V$[i] \neq \bot$ then
                            V$[i] \leftarrow W[i]$
                        end if
                    end if
                end if
            end for
            if $\forall q \in \Pi \setminus ($fd$ \cup \{p\})$, Flag$[q] = true$ then
                if $r < n$ then
                    for all $i \in \Pi$ do
                        Flag$[i] \leftarrow false$
                    end for
                end if
                if $r = n - 1$ then
                    Mes$[n] \leftarrow$ V
                end if
                $r \leftarrow r + 1$
            end if
            for all $i \in \Pi \setminus ($fd$ \cup \{p\})$ do
                if (receive$\langle$Decide, $d\rangle$ from $i$) then
                    $return(d)$
                end if
            end for
        end while
        $d \leftarrow$ the first component of V different from $\bot$; $return(d)$
    end
end function
```

Fig. 5. Algorithm $\mathcal{RCP}$, *Repeated Consensus* with $\mathcal{P}^-$. Part 1: function consensus().

To deal with message loss, the synchronization barrier is obtained by two asynchronous rounds: In the first asynchronous rounds, each process sends a `Decide` message and waits to receive a `Decide` message or a `Start` message from every other process it does not suspect. In the second one, each process sends a `Decide` message and waits to receive a `Start` message or a $\langle$R-$x, -\rangle$ message. Actually, due to message loss it is possible that a process goes to its second round despite some process have not received its `Decide` message, but it cannot finish the second round before every correct processes have finished the first one.

As a faulty process can be suspected before it crashes (due to the quality of $\mathcal{P}^-$), it is possible that a faulty process will not be waited by other processes although it is still alive. To avoid that this process disturbs the round, since a process p suspects a process q, p stops to wait messages from q and to send messages to q.

```
    /* CODE FOR PROCESS $p$ */
variables:
    FD: failure detector of type $\mathcal{P}^-$
procedure R-PROPOSED($v$)
    variables:
        FlagR[$1 \dots n$] $\in \{true, false\}^n$; $\forall i \in \Pi$, FlagR[$i$] is initialized to false
        $stop$: boolean; $stop$ is initialized to false
        $u$: integer;
    begin
        $u \leftarrow$consensus($v$) with FD
        R-DECIDE($u$)
        repeat
            send⟨Decide, $u$⟩ to every process, except $\{p\} \cup$ FD
            for all $i \in \Pi \setminus$ (FD $\cup \{p\}$), FlagR[$i$] $= false$ do
                if (receive⟨Decide, $u$⟩ from $i$) $\vee$ (receive⟨Start⟩ from $i$) then
                    FlagR[$i$] $\leftarrow true$
                end if
            end for
            if $\forall q \in \Pi \setminus$ (FD $\cup \{p\}$), FlagR[$q$] $= true$ then
                $stop \leftarrow true$
            end if
        until $stop$
        for all $i \in \Pi$ do
            FlagR[$i$] $\leftarrow false$
        end for
        $stop \leftarrow false$
        repeat
            send⟨Start⟩ to every process, except $\{p\} \cup$ FD
            for all $i \in \Pi \setminus$ (FD $\cup \{p\}$), FlagR[$i$] $= false$ do
                if (receive⟨Start⟩ from $i$) $\vee$ (receive⟨R-1, $W$⟩ from $j$) then
                    FlagR[$i$] $\leftarrow true$
                end if
            end for
            if $\forall q \in \Pi \setminus$ (FD $\cup \{p\}$), FlagR[$q$] $= true$ then
                $stop \leftarrow true$
            end if
        until $stop$
    end
end procedure
```

Fig. 6. Algorithm $\mathcal{RCP}$, *Repeated Consensus* with $\mathcal{P}^-$. Part 2.

Note also that if the consensus function is executed with $\mathcal{P}^-$, then there is no need to send ⟨R-x, $-$⟩ in round $r > x$ again. We have rewritten the consensus function to take account of these facts, but the behaviour remains the same.

Theorem 5. *Algorithm $\mathcal{RCP}$ (Figure 5 and 6) is a Repeated Consensus algorithm in $\Phi^{\mathcal{F}}$ with $\mathcal{P}^-$.*

Corollary 3. *$\mathcal{P}^-$ is sufficient for solving Repeated Consensus in $\Phi^{\mathcal{F}}$.*

Contrary to these results in system $\Phi^{\mathcal{F}}$, in system $\Phi^{\mathcal{I}}$, we have the same weakest failure detector to solve the consensus problem and the repeated consensus problem:

Proposition 2. *In system $\Phi^{\mathcal{I}}$, if there is an algorithm $\mathcal{A}$ with failure detector $\mathcal{D}$ solving Consensus, then there exists an algorithm solving Repeated Consensus with $\mathcal{D}$.*

References

1. Guerraoui, R., Schiper, A.: The generic consensus service. IEEE Transactions on Software Engineering 27(1), 29–41 (2001)
2. Gafni, E., Lamport, L.: Disk paxos. Distributed Computing 16(1), 1–20 (2003)
3. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
4. Chor, B., Coan, B.A.: A simple and efficient randomized byzantine agreement algorithm. IEEE Trans. Software Eng. 11(6), 531–539 (1985)
5. Dolev, D., Dwork, C., Stockmeyer, L.J.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987)
6. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. Journal of the ACM 35(2), 288–323 (1988)
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
8. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Shared memory vs message passing. Technical report, LPD-REPORT-2003-001 (2003)
10. Eisler, J., Hadzilacos, V., Toueg, S.: The weakest failure detector to solve nonuniform consensus. Distributed Computing 19(4), 335–359 (2007)
11. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In: Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC 2004), pp. 338–346 (2004)
12. Aguilera, M.K., Toueg, S., Deianov, B.: Revisiting the weakest failure detector for uniform reliable broadcast. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 13–33. Springer, Heidelberg (1999)
13. Halpern, J.Y., Ricciardi, A.: A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In: Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 73–82 (1999)
14. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing 65(4), 492–505 (2005)
15. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 399–412. Springer, Heidelberg (2006)
16. Raynal, M., Travers, C.: In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 3–19. Springer, Heidelberg (2006)
17. Zielinski, P.: Anti-omega: the weakest failure detector for set agreement. Technical Report UCAM-CL-TR-694, Computer Laboratory, University of Cambridge, Cambridge, UK (July 2007)
18. Lynch, N.A., Mansour, Y., Fekete, A.: Data link layer: Two impossibility results. In: Symposium on Principles of Distributed Computing, pp. 149–170 (1988)
19. Bazzi, R.A., Neiger, G.: Simulating crash failures with many faulty processors (extended abstract). In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 166–184. Springer, Heidelberg (1992)
20. Delporte-Gallet, C., Devismes, S., Fauconnier, H., Petit, F., Toueg, S.: With finite memory consensus is easier than reliable broadcast. Technical Report hal-00325470, HAL (October 2008)

21. Cavin, D., Sasson, Y., Schiper, A.: Consensus with unknown participants or fundamental self-organization. In: Nikolaidis, I., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2004. LNCS, vol. 3158, pp. 135–148. Springer, Heidelberg (2004)
22. Greve, F., Tixeuil, S.: Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In: DSN, pp. 82–91. IEEE Computer Society, Los Alamitos (2007)
23. Fernández, A., Jiménez, E., Raynal, M.: Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: DSN, pp. 166–178. IEEE Computer Society, Los Alamitos (2006)
24. Chandra, T.D., Toueg, S.: Unreliable failure detectors for asynchronous systems (preliminary version). In: 10th Annual ACM Symposium on Principles of Distributed Computing (PODC 1991), pp. 325–340 (1991)
25. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A realistic look at failure detectors. In: DSN, pp. 345–353. IEEE Computer Society, Los Alamitos (2002)
26. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Department of Computer Science, Cornell University (1994)
27. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over halfduplex links. Journal of the ACM 12, 260–261 (1969)
28. Stenning, V.: A data transfer protocol. Computer Networks 1, 99–110 (1976)

Group Renaming

Yehuda Afek[1], Iftah Gamzu[1,⋆], Irit Levy[1], Michael Merritt[2], and Gadi Taubenfeld[3]

[1] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel
{afek,iftgam,levyirit}@tau.ac.il
[2] AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA
mischu@research.att.com
[3] The Interdisciplinary Center, P.O. Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract. We study the *group renaming* task, which is a natural generalization of the *renaming* task. An instance of this task consists of n processors, partitioned into m groups, each of at most g processors. Each processor knows the name of its group, which is in $\{1, \dots, M\}$. The task of each processor is to choose a new name for its group such that processors from different groups choose different new names from $\{1, \dots, \ell\}$, where $\ell < M$. We consider two variants of the problem: a *tight* variant, in which processors of the same group must choose the same new group name, and a *loose* variant, in which processors from the same group may choose different names. Our findings can be briefly summarized as follows:

1. We present an algorithm that solves the tight variant of the problem with $\ell = 2m - 1$ in a system consisting of g-consensus objects and atomic read/write registers. In addition, we prove that it is impossible to solve this problem in a system having only $(g - 1)$-consensus objects and atomic read/write registers.
2. We devise an algorithm for the loose variant of the problem that only uses atomic read/write registers, and has $\ell = 3n - \sqrt{n} - 1$. The algorithm also guarantees that the number of different new group names chosen by processors from the same group is at most $\min\{g, 2m, 2\sqrt{n}\}$. Furthermore, we consider the special case when the groups are uniform in size and show that our algorithm is self-adjusting to have $\ell = m(m + 1)/2$, when $m < \sqrt{n}$, and $\ell = 3n/2 + m - \sqrt{n}/2 - 1$, otherwise.

1 Introduction

1.1 The Group Renaming Problem

We investigate the *group renaming* task which generalizes the well known *renaming* task [3]. In the original renaming task, each processor starts with a unique identifier taken from a large domain, and the goal of each processor is to select a new unique identifier from a smaller range. Such an identifier can be used, for example,

⋆ Supported by the Binational Science Foundation, by the Israel Science Foundation, and by the European Commission under the Integrated Project QAP funded by the IST directorate as Contract Number 015848.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 58–72, 2008.

to mark a memory slot in which the processor may publish information in its possession. In the *group renaming* task, groups of processors may hold some information which they would like to publish, preferably using a common memory slot for each group. An additional motivation for studying the group version of the problem is to further our understanding about the inherent difficulties in solving tasks with respect to groups [10].

More formally, an instance of the *group renaming* task consists of n processors partitioned into m groups, each of which consists of at most g processors. Each processor has a *group name* taken from some large name space $[M] = \{1, \ldots, M\}$, representing the group that the processor affiliates with. In addition, every processor has a unique *identifier* taken from $[N]$. The objective of each processor is to choose a new group name from $[\ell]$, where $\ell < M$. The collection of new group names selected by the processors must satisfy the *uniqueness* property meaning that any two processors from different groups choose distinct new group names. We consider two variants of the problem:

- a *tight* variant, in which in addition to satisfying the uniqueness property, processors of the same group must choose the same new group name (this requirement is called the *consistency* property), and
- a *loose* variant, in which processors from the same group may choose different names, rather than a single one, as long as no two processors from different groups choose the same new name.

1.2 Summary of Results

We present a wait-free algorithm that solves the tight variant of the problem with $\ell = 2m - 1$ in a system equipped with g-consensus objects and atomic read/write registers. This algorithm extends the upper bound result of Attiya et al. [3] for $g = 1$. On the lower bound side, we show that there is no wait-free implementation of tight group renaming in a system equipped with $(g - 1)$-consensus objects and atomic read/write registers. In particular, this result implies that there is no wait-free implementation of tight group renaming using only atomic read/write registers for $g \geq 2$.

We then restrict our attention to shared memory systems which support only atomic read/write reagisters and study the loose variant. We develop a self-adjusting algorithm, namely, an algorithm that achieves distinctive performance guarantees conditioned on the number of groups and processors. On worst case, this algorithm has $\ell = 3n - \sqrt{n} - 1$, while guaranteeing that the number of different new group names chosen by processors from the same group is at most $\min\{g, 2m, 2\sqrt{n}\}$. It seems worthy to note that the algorithm is built around a filtering technique that overcomes scenarios in which both the size of the maximal group and the number of groups are large, i.e., $g = \Omega(n)$ and $m = \Omega(n)$. Essentially, such scenario arises when there are $\Omega(n)$ groups containing only few members and few groups containing $\Omega(n)$ members.

We also consider the special case when the groups are uniform in size, and refine the analysis of our loose group renaming algorithm. Notably, we demonstrate that $\ell = m(m + 1)/2$, when $m < \sqrt{n}$, and $\ell = 3n/2 + m - \sqrt{n}/2 - 1$, otherwise. This last result settles, to some extent, an open question posed by Gafni [10].

1.3 Related Work

Group solvability was first introduced and investigated in [10]. The renaming problem was first solved for message-passing systems [3], and then for shared memory systems using atomic registers [6]. Both these papers present one-shot algorithms (i.e., solutions that can be used only once). In [8] the first long-lived renaming algorithm was presented: The ℓ-assignment algorithm presented in [8] can be used as an optimal long-lived $(2n-1)$-renaming algorithm with exponential step complexity. Several of the many papers on renaming using atomic registers are [1,2,4,11,14,15]. Other references are mentioned later in the paper.

2 Model and Definitions

Our model of computation consists of an asynchronous collection of n processors communicating via shared objects. Each object has a *type* which defines the set of operations that the object supports. Each object also has *sequential specification* that specifies how the object behaves when these operations are applied sequentially. Asynchrony means that there is no assumptions on the relative speeds of the processors.

Various systems differ in the level of *atomicity* that is supported. Atomic (or indivisible) operations are defined as operations whose execution is not interfered with by other concurrent activities. This definition of atomicity is too restrictive, and it is safe to relax it by assuming that processors can try to access the object at the same time, however, although operations of concurrent processors may overlap, each operation should appear to take effect instantaneously. In particular, operations that do not overlap should take effect in there "real-time" order. This type of correctness requirement is called *linearizability* [13].

We will always assume that the system supports *atomic registers*, which are shared objects that support atomic reads and writes operations. In addition, the system may also support forms of atomicity which are stronger than atomic reads and writes. One specific atomic object that will play an important role in our investigation is the consensus object. A consensus object o supports one operation: *o.propose*(v), satisfying:

1. *Agreement.* In any run, the *o.propose*$()$ operation returns the same value, called the *consensus value*, to every processor that invokes it.
2. *Validity.* In any run, if the consensus value is v, then some processor invoked *o.propose*(v).

Throughout the paper, we will use the notation g-consensus to denote a consensus object for g processors.

An object is *wait-free* if it guarantees that *every* processor is always able to complete its pending operation in a finite number of its own steps regardless of the execution speed of other processors (does not admit starvation). Similarly, an implementation or an algorithm is wait-free, if every processor makes a decision within a finite number of its own steps. We will focus only on wait-free objects, implementations or algorithms. Next, we define two notions for measuring the relative computational power of shared objects.

- The *consensus number* of an object of type o, is the largest n for which it is possible to implement an n-consensus object in a wait-free manner, using any number of objects of type o and any number of atomic registers. If no largest n exists, the consensus number of o is infinite.
- The *consensus hierarchy* (also called the wait-free hierarchy) is an infinite hierarchy of objects such that the objects at level i of the hierarchy are exactly those objects with consensus number i.

It has been shown in [12], that in the consensus hierarchy, for any positive i, in a system with i processors: (1) no object at level less than i together with atomic registers can implement any object at level i; and (2) each object at level i together with atomic registers can implement any object at level i or at a lower level, in a system with i processors. Classifying objects by their consensus numbers is a powerful technique for understanding the relative power of shared objects.

Finally, for simplicity, when refereing to the group renaming problem, we will assume that m, the number of groups, is greater or equal to 2.

3 Tight Group Renaming

3.1 An Upper Bound

In what follows, we present a wait-free algorithm that solves tight group renaming using g-consensus objects and atomic registers. Essentially, we prove the following theorem.

Theorem 1. *For any $g \geq 1$, there is a wait-free implementation of tight group renaming with $\ell = 2m - 1$ in a system consisting of g-consensus objects and atomic registers.*

Corollary 1. *The consensus number of tight group renaming is at most g.*

Our implementation, i.e., Algorithm 1, is inspired by the renaming algorithm of Attiya et al. [3], which achieves an optimal new names space size of $2n - 1$. In this renaming algorithm, each processor iteratively picks some name and suggests it as its new name until an agreement on the collection of new names is reached. The communication between the processors is done using an atomic snapshot object. Our algorithm deviates from this scheme by adding an agreement step between processors of the same group, implemented using g-consensus objects. Intuitively, this agreement step ensures that all the processors of any group will follow the decisions made by the "fastest" processor in the group. Consequently, the selection of the new group names can be determined between the representatives of the groups, i.e., the "fastest" processors. This enables us to obtain the claimed new names space size of $2m - 1$. It is worthy to note that the "fastest" processor of some group may change over time, and hence our agreement step implements a "follow the (current) group leader" strategy. We believe that this concept may be of independent interest. Note that the group name of processor i is designated by GID_i, and the overall number of iterations executed is marked by I.

We now turn to establish Theorem 1. Essentially, this is achieved by demonstrating that Algorithm 1 maintains the consistency and uniqueness properties (Lemmas 2 and

Algorithm 1. Tight group renaming algorithm: code for processor $i \in [N]$.

```
In shared memory:
    $\mathsf{SS}[1,\ldots,N]$ array of swmr registers, initially $\perp$.
    $\mathsf{HIS}[1,\ldots,N][1,\ldots,I][1,\ldots,N]$ array of swmr registers, initially $\perp$.
    $\mathsf{CON}[1,\ldots,M][1,\ldots,I]$ array of $g$-consensus objects.
 1: $p \leftarrow 1$
 2: $k \leftarrow 1$
 3: while true do
 4:     $\mathsf{SS}[i] \leftarrow \langle \mathsf{GID}_i, p, k \rangle$
 5:     $\mathsf{HIS}[i][k][1,\ldots,N] \leftarrow$ Snapshot$(\mathsf{SS})$
        ▷ Agree on $w$, the winner of group $GID_i$ in iteration $k$, and import its snapshot:
 6:     $w \leftarrow \mathsf{CON}[\mathsf{GID}_i][k]$.Compete$(i)$
 7:     $(\langle \mathsf{GID}_1, p_1, k_1 \rangle, \ldots, \langle \mathsf{GID}_N, p_N, k_N \rangle) \leftarrow \mathsf{HIS}[w][k][1,\ldots,N]$
        ▷ Check if $p_w$, the proposal of $w$, can be chosen as the new name of group $GID_i$:
 8:     $P = \{p_j : j \in [N]$ has $\mathsf{GID}_j \neq \mathsf{GID}_w$ and $k_j = \max_{q \in [N]}\{k_q : \mathsf{GID}_q = \mathsf{GID}_j\}\}$
 9:     if $p_w \in P$ then
10:         $r \leftarrow$ the rank of $\mathsf{GID}_w$ in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
11:         $p \leftarrow$ the $r$-th integer not in $P$
12:     else return $p_w$
13:     end if
14:     $k \leftarrow k + 1$
15: end while
```

3), that it has $\ell = 2m - 1$ (Lemma 4), and that it terminates after a finite number of steps (Lemma 5). Let us denote the value of p written to the snapshot array (see line 4) in some iteration as the *proposal value* of the underlying processor in that iteration.

Lemma 1. *The proposal values of processors from the same group is identical in any iteration.*

Proof. Consider some group. One can easily verify that the processors of that group, and in fact all the processors, have an identical proposal value of 1 in the first iteration. Thus, let us consider some iteration $k > 1$ and prove that all these processors have an identical proposal value. Essentially, this is done by claiming that all the processors update their value of p in the preceding iteration in an identical manner. For this purpose, notice that all the processors compete for the same g-consensus object in that iteration, and then import the same snapshot of the processor that won this consensus (see lines 6–7). Consequently, they execute the code in lines 8–13 in an identical manner. In particular, this guarantees that the update of p in line 11 is done exactly alike. □

Lemma 2. *All the processors of the same group choose an identical new group name.*

Proof. The proof of this lemma follows the same line of argumentation presented in the proof of Lemma 1. Again, the key observation is that in each iteration, all the processors of some group compete for the same g-consensus object, and then import the same snapshot. Since the decisions made by the processors in lines 8–13 are solely based on this snapshot, it follows that they are identical. In particular, this ensures that once a processor chooses a new group name, all the other processors will follow its lead and choose the same name. □

Lemma 3. *No two processors of different groups choose the same new group name.*

Proof. Recall that we know, by Lemma 2, that all the processors of the same group choose an identical new group name. Hence, it is sufficient that we prove that no two groups select the same new name. Assume by way of contradiction that this is not the case, namely, there are two distinct groups $\mathcal{G}$ and $\mathcal{G}'$ that select the same new group name p^*. Let k and k' be the iteration numbers in which the decisions on the new names of $\mathcal{G}$ and $\mathcal{G}'$ are done, and let $w \in \mathcal{G}$ and $w' \in \mathcal{G}'$ be the corresponding processors that won the g-consensus objects in that iterations. Now, consider the snapshot $(\langle \mathsf{GID}_1, p_1, k_1\rangle, \ldots, \langle \mathsf{GID}_N, p_N, k_N\rangle)$, taken by w in its k-th iteration. One can easily validate that $p_w = p^*$ since w writes its proposed value before taking a snapshot. Similarly, it is clear that $p'_{w'} = p^*$ in the snapshot $(\langle \mathsf{GID}'_1, p'_1, k'_1\rangle, \ldots, \langle \mathsf{GID}'_N, p'_N, k'_N\rangle)$, taken by w' in its k'-th iteration. By the linearizability property of the atomic snapshot object and without loss of generality, we may assume that snapshot of w was taken before the snapshot of w'. Consequently, w' must have captured the proposal value of w in its snapshot, i.e., $p'_w = p^*$. This implies that p^* appeared in the set P of w'. However, this violates the fact that w' reached the decision step in line 12, a contradiction. □

Lemma 4. *All the new group names are from the range $[\ell]$, where $\ell = 2m - 1$.*

Proof. In what follows, we prove that the proposal value of any processor in any iteration is in the range $[\ell]$. Clearly, this proves the lemma as the chosen name of any group is a proposal value of some processor. Consider some processor. It is clear that its first iteration proposal value is in the range $[\ell]$. Thus, let us consider some iteration $k > 1$ and prove that its proposal value is at most $2m-1$. Essentially, this is done by bounding the value of p calculated in line 11 of the preceding iteration. For this purpose, we first claim that the set P consists of at most $m - 1$ values. Notice that P holds the proposal values of processors from at most $m - 1$ groups. Furthermore, observe that for each of those groups, it holds the proposal values of processors having the same maximal iteration counter. This implies, in conjunction with Lemma 1, that for each of those groups, the proposal values of the corresponding processors are identical. Consequently, P consists of at most $m - 1$ distinct values. Now, one can easily verify that the rank of every group calculated in line 10 is at most m. Therefore, the new value of p is no more than $2m - 1$. □

Lemma 5. *Any processor either takes finite number of steps or chooses a new group name.*

Proof. The proof of this theorem is a natural generalization of the termination proof of the renaming algorithm (see, e.g., [5, Sec. 16.3]). Thus, we defer it to the final version of the paper. □

3.2 An Impossibility Result

In Appendix A.1, we provide an FLP-style proof of the following theorem.

Theorem 2. *For any $g \geq 2$, it is impossible to wait-free implement tight group renaming in a system having $(g-1)$-consensus objects and atomic registers.*

In particular, Theorem 2 implies that there is no wait-free implementation of tight group renaming, even when $g = 2$, using only atomic registers.

4 Loose Group Renaming

In this section, we restrict our attention to shared memory systems which support only atomic registers. By Theorem 2, we know that it is impossible to solve the tight group renaming problem unless we relax our goal. Accordingly, we consider a variant in which processors from the same group may choose a different new group name, as long as the uniqueness property is maintained. The objective in this case is to minimize both the *inner scope* size, which is the upper bound on the number of new group names selected by processors from the same group, and the *outer scope* size, which is the new group names range size. We use the notation, (α, β)-group renaming algorithm to designate an algorithm yielding an inner scope of α and an outer scope of β.

4.1 The Non-uniform Case

In the following we consider the task when group sizes are not uniform. We present a group renaming algorithm having a worst case inner scope size of $\min\{g, 2m, 2\sqrt{n}\}$ and a worst case outer scope size of $3n - \sqrt{n} - 1$. The algorithm is self-adjusting with respect to the input properties. Namely, it achieves better performance guarantees conditioned on the number of groups and processors. It seems worthy to emphasize that the performance guarantees of our algorithm are not only based on g and m, but also on $\sqrt{n}$, which is crucial in several cases.

The algorithm is built upon a consolidation of two algorithms, denoted as Algorithm 2 and Algorithm 3. Both algorithms are adaptations of previously known renaming methods for groups (see, e.g., [10]). Algorithm 2, which efficiently handles small-sized groups, is a $(g, n+m-1)$-group renaming algorithm, while Algorithm 3, which efficiently attends to small number of groups, is a $(\min\{m, g\}, m(m+1)/2)$-group renaming algorithm.

Theorem 3. *Algorithm 2 is a wait-free $(g, n+m-1)$-group renaming algorithm.*

Proof. The algorithm is very similar to the original renaming algorithm of Attiya et. al. [3]. While there processors select a new name by computing the rank of their original large id among the ids of participating processors, here processors consider the rank of their original *group name* among the already published (participating) original group names. One can prove that Algorithm 2 maintains the uniqueness property and terminates after finite number of steps by applying nearly identical arguments to those used in the analogous proofs of the underlying renaming method (see, e.g., [5, Sec. 16.3]). Therefore, we only focus on analyzing the size of the resulting new name-spaces. The inner scope size of the algorithm is trivially g since there are at most g processors in any group. We turn to bound the outer scope size. This is done by demonstrating that the proposal value p_i of any processor i in any iteration is at most $n + m - 1$. Clearly, p_i satisfies this requirement in the first iteration as its value is 1. Hence, let us consider some other iteration and bound its proposal value. This is accomplished by bounding the value of p_i calculated in line 7 of the preceding iteration. For this purpose, notice that the rank of every group calculated in line 6 is at most m. Furthermore, there are at most $n - 1$ values proposed by other processors. Thus, the new value of p_i is at most $n + m - 1$. □

Algorithm 2. code for processor $i \in [N]$.

In shared memory: $\mathsf{SS}[1, \ldots, N]$ array of swmr registers, initially $\perp$.

```
1:  $p_i \leftarrow 1$
2:  while true do
3:      $\mathsf{SS}[i] \leftarrow \langle \mathsf{GID}_i, p_i \rangle$
4:      $(\langle \mathsf{GID}_1, p_1 \rangle, \ldots, \langle \mathsf{GID}_N, p_N \rangle) \leftarrow$ Snapshot($\mathsf{SS}$)
5:      if $p_i = p_j$ for some $j \in [N]$ having $\mathsf{GID}_j \neq \mathsf{GID}_i$ then
6:          $r \leftarrow$ the rank of $\mathsf{GID}_i$ in $\{\mathsf{GID}_k \neq \perp : k \in [N]\}$
7:          $p_i \leftarrow$ the $r$-th integer not in $\{p_k \neq \perp : k \in [N] \setminus \{i\}\}$
8:      else return $p_i$
9:      end if
10: end while
```

Theorem 4. *Algorithm 3 is a wait-free* $(\min\{m, g\}, m(m + 1)/2)$*-group renaming algorithm.*

Proof. In this algorithm each processor records its participation by publishing its id and its group original name. Each processor then takes a snapshot of the memory and returns as its new group name the size of the snapshot it had obtained, concatenated with its group id rank among the group ids recorded in the snapshot. One can prove that Algorithm 3 supports the uniqueness property by applying nearly identical arguments to those used in the corresponding proof of the underlying renaming method (see, e.g., [7, Sec. 6]). Moreover, it is clear that the algorithm terminates after finite number of steps. Thus, we only focus on analyzing the performance properties of the algorithm. We begin with the inner scope size. Particularly, we prove a bound of m, noting that a bound of g is trivial since there are at most g processors in any group. Consider the case that two processors of the same group obtain the same number of observable groups $\tilde{m}$ in line 3. We argue that they also choose the same new group name. For this purpose, notice that the set of GIDs that reside in SS may only grow during any execution sequence. Hence, if two processors have an identical $\tilde{m}$ then their snapshot holds the same set of GIDs. Consequently, if those processors are of the same group then their group rank calculated in line 4 is also the same, and therefore the new names they select are identical. This implies that the number of new group names selected by processors from the same group is bound by the maximal value of $\tilde{m}$, which is clearly never greater than m. We continue by bounding the outer scope size. As already noted, $\tilde{m} \leq m$, and the rank of every group is at most m. Thus, the maximal group name is no more than $m(m - 1)/2 + m$. □

Algorithm 3. code for processor $i \in [N]$.

```
In shared memory: $\mathsf{SS}[1,\ldots,N]$ array of swmr registers, initially $\perp$.
$\mathsf{SS}[i] \leftarrow \mathsf{GID}_i$
$(\mathsf{GID}_1,\ldots,\mathsf{GID}_N) \leftarrow$ Snapshot($\mathsf{SS}$)
$\tilde{m} \leftarrow$ the number of distinct GIDs in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
$r \leftarrow$ the rank of $\mathsf{GID}_i$ in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
return $\tilde{m}(\tilde{m}-1)/2 + r$
```

We are now ready to present our self-adjusting loose group renaming algorithm. The algorithm has its roots in the natural approach that applies the best response with respect to the instance under consideration. For example, it is easy to see that Algorithm 3 outperforms Algorithm 2 with respect to the inner scope size, for any instance. In addition, one can verify that when $m < \sqrt{n}$, Algorithm 3 has an outer scope size of at most $n/2 - \sqrt{n}/2$, whereas Algorithm 2 has an outer scope size of at least n. Hence, given an instance having $m < \sqrt{n}$, the best response would be to execute Algorithm 3. Unfortunately, a straight-forward application of this approach has several difficulties.

One immediate difficulty concerns the implementation since none of the processors have prior knowledge of the real values of m or g. Our algorithm bypasses this difficulty by maintaining an estimation of these parameters using an atomic snapshot object. Another difficulty concerns with performance issues. Specifically, both algorithms have poor inner scope size guarantees for instances which simultaneously satisfy $g = \Omega(n)$ and $m = \Omega(n)$. One concrete example having $g = n/2$ and $m = n/2 + 1$ consists of a single group having $n/2$ members and $n/2$ singleton groups. In this case, both algorithms have an inner scope size guarantee of $n/2$. We overcome this difficulty by sensibly combining the algorithms, therefore yielding an inner scope size guarantee of $2\sqrt{n}$ for these "hard" cases. The key observation utilized in this context is that if there are many groups then most of them must be small. Consequently, by filtering out the small-sized groups, we are left with a small number of large groups that we can handle efficiently. Note that Algorithm 4 employs Algorithm 3 as sub-procedure in two cases (see lines 6 and 12). It is assumed that the shared memory space used by each application of the algorithm is distinct.

Theorem 5. *Algorithm 4 is a group renaming algorithm having a worst case inner scope size of* $\min\{g, 2m, 2\sqrt{n}\}$ *and a worst case outer scope size of* $3n - \sqrt{n} - 1$.

Proof. We begin by establishing the correctness of the algorithm. For this purpose, we demonstrate that it maintains the uniqueness property and terminates after finite number of steps. One can easily validate that the termination property holds since both Algorithm 2 and Algorithm 3 terminate after finite number of steps. It is also easy to verify that the uniqueness property is maintained. This follows by recalling that both Algorithm 2 and Algorithm 3 maintain the uniqueness property, and noticing that each case of the if statement (see lines 5–14) utilizes a distinct set of new names. To be precise, one should observe that any processor that executes Algorithm 3 in line 6 is assigned a new name in the range $\{1,\ldots,n/2 - \sqrt{n}/2\}$, any processor that executes

Algorithm 2 in line 9 is assigned a new name in the range $\{n/2 - \sqrt{n}/2 + 1, \ldots, 5n/2 - \sqrt{n}/2 - 1\}$, and any processor that executes Algorithm 3 in line 12 is assigned a new name whose value is at least $5n/2 - \sqrt{n}/2$. The first claim results by the outer scope properties of Algorithm 3 and the fact that processors from less than $\sqrt{n}$ groups may execute this algorithm. The second argument follows by the outer scope properties of Algorithm 2, combined with the observation that $m \leq n$, and the fact that the value of the name returned by the algorithm is increased by $n/2 - \sqrt{n}/2$ in line 10. Finally, the last claim holds since Algorithm 3 is guaranteed to attain a positive-valued integer name, and the value of this name is increased by $5n/2 - \sqrt{n}/2 - 1$ in line 13.

Algorithm 4. Adjusting group renaming algorithm: code for processor $i \in [N]$.

```
In shared memory: $\mathsf{SS}[1, \ldots, N]$ array of swmr registers, initially $\perp$.
 1: $\mathsf{SS}[i] \leftarrow \mathsf{GID}_i$
 2: $(\mathsf{GID}_1, \ldots, \mathsf{GID}_N) \leftarrow$ Snapshot($\mathsf{SS}$)
 3: $\tilde{m} \leftarrow$ the number of distinct GIDs in $\{\mathsf{GID}_j \neq \perp : j \in [N]\}$
 4: $\tilde{g} \leftarrow$ the number of processors $j \in [N]$ having $\mathsf{GID}_j = \mathsf{GID}_i$
 5: **if** $\tilde{m} < \sqrt{n}$ **then**
 6:     $x \leftarrow$ the outcome of Algorithm 3 (using shared memory $\mathsf{SS}_1[1, \ldots, N]$)
 7:     return $x$
 8: **else if** $\tilde{g} \leq \sqrt{n}$ **then**
 9:     $x \leftarrow$ the outcome of Algorithm 2 (using shared memory $\mathsf{SS}_2[1, \ldots, N]$)
10:     return $x + n/2 - \sqrt{n}/2$
11: **else**
12:     $x \leftarrow$ the outcome of Algorithm 3 (using shared memory $\mathsf{SS}_3[1, \ldots, N]$)
13:     return $x + 5n/2 - \sqrt{n}/2 - 1$
14: **end if**
```

We now turn to establish the performance properties of the algorithm. We demonstrate that it is self-adjusting and has the following (inner scope, outer scope) properties:

$$\begin{cases} (\ \min\{m, g\},\ \ m(m+1)/2\ ) & m < \sqrt{n} \\ (\ g,\ \ 3n/2 + m - \sqrt{n}/2 - 1\ ) & m \geq \sqrt{n} \text{ and } g \leq \sqrt{n} \\ (\min\{g, 2\sqrt{n}\},\ 3n - \sqrt{n} - 1) & m \geq \sqrt{n} \text{ and } g > \sqrt{n} \end{cases}$$

Case I: $\mathbf{m < \sqrt{n}}$. The estimation value $\tilde{m}$ always satisfy $\tilde{m} \leq m$. Therefore, all the processors execute Algorithm 3 in line 6. The properties of Algorithm 3 guarantee that the inner scope size is $\min\{m, g\}$ and the outer scope size is $m(m+1)/2$. Take notice that $\min\{m, g\} \leq \min\{g, 2m, 2\sqrt{n}\}$ and $m(m+1)/2 \leq 3n - \sqrt{n} - 1$ since $m < \sqrt{n}$. Thus, the performance properties of the algorithm in this case support the worst case analysis.

Case II: $\mathbf{m \geq \sqrt{n}}$ and $\mathbf{g \leq \sqrt{n}}$. The estimation values never exceed their real values, namely, $\tilde{m} \leq m$ and $\tilde{g} \leq g$. Consequently, some processors may execute Algorithm 3 in line 6 and some may execute Algorithm 2 in line 9, depending on the concrete execution sequence. The inner scope size guarantee is trivially satisfied since there are at most g processors in each group. Furthermore, one can establish the outer scope size guarantee by simply summing the size of the name space that may be used by Algorithm 3, which is $n/2 - \sqrt{n}/2$, with the size of the name space that may be used by Algorithm 2, which is $n + m - 1$. Notice that $g \leq \min\{g, 2m, 2\sqrt{n}\}$ since $g \leq \sqrt{n} \leq m$, and $3n/2 + m - \sqrt{n}/2 - 1 \leq 3n - \sqrt{n} - 1$ as $m \leq n$. Hence, the performance properties of the algorithm in this case support the worst case analysis.

Case III: $\mathbf{m \geq \sqrt{n}}$ and $\mathbf{g > \sqrt{n}}$. Every processors may execute any of the algorithms, depending of the concrete execution sequence. The first observation one should make is that no more than $\sqrt{n}$ new names may be collectively assigned to processors of the same group by Algorithm 3 in line 6 and Algorithm 2 in line 9. Moreover, one should notice that any processor that executes Algorithm 3 in line 12 is part of a group of size greater than $\sqrt{n}$. Consequently, processors from less than $\sqrt{n}$ groups may execute it. This implies, in conjunction with the properties of Algorithm 3, that no more than $\sqrt{n}$ new names may be assigned to each group, and at most $n/2 - \sqrt{n}/2$ names are assigned by this algorithm. Putting everything together, we attain that the inner scope size is $\min\{g, 2\sqrt{n}\}$ and the outer scope size is $3n - \sqrt{n} - 1$. It is easy to see that $\min\{g, 2\sqrt{n}\} \leq \min\{g, 2m, 2\sqrt{n}\}$ since $m \geq \sqrt{n}$, and thus the performance properties of the algorithm in this case also support the worst case analysis. □

4.2 The Uniform Case

In what follows, we study the problem when the groups are guaranteed to be uniform in size. We refine the analysis of Algorithm 4 by establishing that it is a loose group renaming algorithm having a worst case inner scope size of $\min\{m, g\}$, and an outer scope size of $3n/2 + m - \sqrt{n}/2 - 1$. Note that $\min\{m, g\} \leq \sqrt{n}$ in this case. In particular, we demonstrate that the algorithm is self-adjusting and has the following (inner scope, outer scope) properties:

$$\begin{cases} (\ \min\{m, g\},\ m(m+1)/2\ ) & m < \sqrt{n} \\ (\ g,\ 3n/2 + m - \sqrt{n}/2 - 1\ ) & m \geq \sqrt{n} \end{cases}$$

This result settles, to some extent, an open question posed by Gafni [10], which called for a self-adjusting group renaming algorithm that requires at most $m(m+1)/2$ names on one extreme, and no more than $2n - 1$ names on the other.

The key observation required to establish this refinement is that $n = m \cdot g$ when the groups are uniform in size. Consequently, either $m < \sqrt{n}$ or $g \leq \sqrt{n}$. Since the estimation values that each processor sees cannot exceed the corresponding real values, no processor can ever reach the second execution of Algorithm 3 in line 12. Now, the proof of the performance properties follows the same line of argumentation presented in the proof of Theorem 5.

5 Discussion

This paper has considered and investigated the tight and loose variants of the group renaming problem. Below we discuss few ways in which our results can be extended. An immediate open question is whether a g-consensus task can be constructed from group renaming tasks for groups of size g, in a system with g processes. Another question is to design an *adaptive* group renaming algorithm in which a processor is assigned a new group name, from the range 1 through k where k is a constant multiple of the contention (i.e., the number of different active groups) that the processor experiences. We have considered only one-shot tasks (i.e., solutions that can be used only once), it would be interesting to design long-lived group renaming algorithms. We have focused in this work mainly on reducing the new name space as much as possible, it would be interesting to construct algorithms also with low space and time (step) complexities. Finally, the *k-set* consensus task, a generalization of the consensus task, enables for each processor that starts with an input value from some domain, to choose some participating processor' input as its output, such that all processors together may choose no more than k distinct output values. It is interesting to find out what type of group renaming task, if any, can be implemented using k-set consensus tasks and registers.

References

1. Afek, Y., Attiya, H., Fouren, A., Stupp, G., Touitou, D.: Long-lived renaming made adaptive. In: Proc. 18th ACM Symp. on Principles of Distributed Computing, pp. 91–103 (May 1999)
2. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distributed Computing 30, 67–86 (2002)
3. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37(3), 524–548 (1990)
4. Attiya, H., Fouren, A.: Algorithms adapting to point contention. Journal of the ACM 50(4), 144–468 (2003)
5. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley Interscience, Chichester (2004)
6. Bar-Noy, A., Dolev, D.: Shared memory versus message-passing in an asynchronous. In: Proc. 8th ACM Symp. on Principles of Distributed Computing, pp. 307–318 (1989)
7. Bar-Noy, A., Dolev, D.: 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. Burns, J., Peterson, G.: The ambiguity of choosing. In: Proc. 8th ACM Symp. on Principles of Distributed Computing, pp. 145–158 (August 1989)
9. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
10. Gafni, E.: Group-solvability. In: Proceedings 18th International Conference on Distributed Computing, pp. 30–40 (2004)
11. Gafni, E., Merritt, M., Taubenfeld, G.: The concurrency hierarchy, and algorithms for unbounded concurrency. In: Proc. 20th ACM Symp. on Principles of Distributed Computing, pp. 161–169 (August 2001)
12. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
14. Inoue, M., Umetani, S., Masuzawa, T., Fujiwara, H.: Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 123–135. Springer, Heidelberg (2001)
15. Moir, M., Anderson, J.H.: Wait-free algorithms for fast, long-lived renaming. Science of Computer Programming 25(1), 1–39 (1995)

A Tight Group Renaming

A.1 An Impossibility Result

In what follows, we establish the proof of Theorem 2. Our impossibility proof follows the high level FLP-approach employed in the context of the consensus problem (see, e.g., [12,9]). Namely, we assume the existence of a tight group renaming algorithm, and then derive a contradiction by constructing a sequential execution in which the algorithm fails, either because it is inconsistent, or since it runs forever. Prior to delving into technicalities, we introduce some terminology.

The *decision value* of a processor is the new group name selected by that processor. Analogously, the decision value of a group is the new group name selected by all processors of that group. An algorithm state is *multivalent* with respect to group $\mathcal{G}$ if the decision value of $\mathcal{G}$ is not yet fixed, namely, the current execution can be extended to yield different decision values of $\mathcal{G}$. Otherwise, it is *univalent*. In particular, an *x-valent* state with respect to $\mathcal{G}$ is a univalent state with respect to $\mathcal{G}$ yielding a decision value of x. A *decision step* with respect to $\mathcal{G}$ is an execution step that carries the algorithm from a multivalent state with respect to $\mathcal{G}$ to a univalent state with respect to $\mathcal{G}$. A processor is *active* with respect to a algorithm state if its decision value is still not fixed. A algorithm state is *critical* with respect to $\mathcal{G}$ if it is multivalent with respect to $\mathcal{G}$ and any step of any active processor is a decision step with respect to $\mathcal{G}$.

Lemma 6. *Every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group.*

Proof. We begin by establishing that every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to *some* group. Consider some group renaming algorithm, and assume by contradiction that the initial algorithm state is univalent with respect to all groups for every input instance. We argue that all processors implement some function $f : [M] \rightarrow [\ell]$ for computing their new group name. For this purpose, consider some processor whose group name is $a \in [M]$. Notice that this processor may be scheduled to execute a "solo run". Let us assume that its decision value in this case is $x \in [\ell]$. Since the initial algorithm state is univalent with respect to the group of that processor, it follows that in any execution this processor must decide x, regardless of the other groups, their name, and their scheduling. The above-mentioned argument follows by recalling that all processors execute the same algorithm, and noticing that a could have been any initial group name. Now, recall that $M > \ell$. This implies that there are at least two group names $a_1, a_2 \in [M]$ such that $f(a_1) = f(a_2)$. Correspondingly, there are input instances in which two processors from two different groups decide on the same new group name, violating the uniqueness property.

We now turn to prove that every group renaming algorithm admits an input instance whose initial algorithm state is multivalent with respect to a maximal size group. Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group $\mathcal{G}$. Namely, there are two execution sequences σ_1, σ_2 that lead to different decision values of $\mathcal{G}$. Now, if $\mathcal{G}$ is maximal in size then we are done.

Otherwise, consider the input instance obtained by adding processors to $\mathcal{G}$ until it becomes maximal in size. Notice that the execution sequences σ_1 and σ_2 are valid with respect to the new input instance. In addition, observe that each possessor must decide on the same value as in the former instance. This follows by the assumption that none of the processors has prior knowledge about the other processors and groups, and thus each processor cannot distinguish between the two instances. Hence, the initial algorithm state is also multivalent with respect to $\mathcal{G}$ in this new instance. □

Lemma 7. *Every group renaming algorithm admits an input instance for which a critical state with respect to a maximal size group may be reached.*

Proof. We prove that every group renaming algorithm which admits an input instance whose initial algorithm state is multivalent with respect to some group may reach a critical state with respect to that group. Notice that having this claim proved, the lemma follows as consequence of Lemma 6. Consider some group renaming algorithm, and suppose its initial algorithm state is multivalent with respect to group $\mathcal{G}$. Consider the following sequential execution, starting from this state. Initially, some arbitrary processor executes until it reaches a state where its next operation leaves the algorithm in a univalent state with respect to $\mathcal{G}$, or until it terminates and decides on a new group name. Note that the latter case can only happen if the underlying processor is not affiliated to $\mathcal{G}$. Also note that the processor must eventually reach one of the above-mentioned states since the algorithm is wait-free and cannot run forever. Later on, another arbitrary processor executes until it reaches a similar state, and so on. This sequential execution continues until reaching a state in which any step of any active processor is a decision step with respect to $\mathcal{G}$. Again, since the algorithm cannot run forever, it must eventually reach such state, which is, by definition, critical. □

We are now ready to prove the impossibility result.

Proof of Theorem 2. Assume that there is a group renaming algorithm implemented from atomic registers and r-consensus objects, where $r < g$. We derive a contradiction by constructing an infinite sequential execution that keeps such algorithm in a multivalent state with respect to some maximal size group. By Lemma 7, we know that there is an input instance and a corresponding execution of the algorithm that leads to a critical state s with respect to some group $\mathcal{G}$ of size g. Keep in mind that there are at least g active processors in this critical state since, in particular, all the processors of $\mathcal{G}$ are active. Let p and q be two active processors in the critical state which respectively carry the algorithm into an x-valent and a y-valent states with respect to $\mathcal{G}$, where x and y are distinct. We now consider four cases, depending on the nature of the decision steps taken by the processors:

Case I: One of the processors reads a register. Let us assume without loss of generality that this processor is p. Let s' be the algorithm state reached if p's read step is immediately followed by q's step, and let s'' be the algorithm state following q's step. Notice that s' and s'' differ only in the internal state of p. Hence, any processor $p' \in \mathcal{G}$, other than p, cannot distinguish between these states. Thus, if it executes a "solo run", it must decide on the same value. However, an impossibility follows since s' is

x-valent with respect to $\mathcal{G}$ whereas s'' is y-valent. This case is schematically described in Figure 1(a).

Case II: Both processors write to the same register. Let s' be the algorithm state reached if p's write step is immediately followed by q's write step, and let s'' be the algorithm state following q's write step. Observe that in the former scenario q overwrites the value written by p. Hence, s' and s'' differ only in the internal state of p. Therefore, any processor $p' \in \mathcal{G}$, other than p, cannot distinguish between these states. The impossibility follows identically to Case I.

Case III: Each processor writes to or competes for a distinct register or consensus object. In what follows, we prove impossibility for the scenario in which both processors write to different registers, noting that impossibility for other scenarios can be easily established using nearly identical arguments. The algorithm state that results if p's write step is immediately followed by q's write step is identical to the state which results if the write steps occur in the opposite order. This is clearly impossible as one state is x-valent and the other is y-valent. This case is schematically illustrated in Figure 1(b).

Case IV: All active processors compete for the same consensus object. As mentioned above, there are at least g active processors in the critical state. Additionally, we assumed that the algorithm uses r-consensus objects, where $r < g$. This implies that the underlying consensus object is accessed by more processors then its capacity, which is illegal. ■

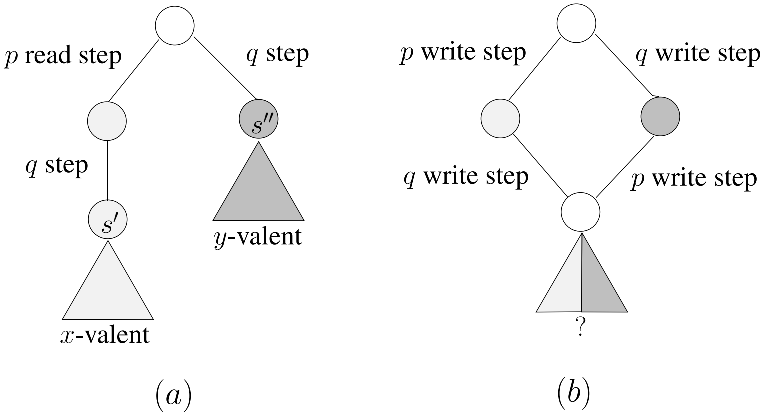


Fig. 1. The decision steps cases

Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%

Björn Andersson

IPP Hurray Research Group,
Polytechnic Institute of Porto, Portugal

Abstract. Consider the problem of scheduling real-time tasks on a multiprocessor with the goal of meeting deadlines. Tasks arrive sporadically and have implicit deadlines, that is, the deadline of a task is equal to its minimum inter-arrival time. Consider this problem to be solved with global static-priority scheduling. We present a priority-assignment scheme with the property that if at most 38% of the processing capacity is requested then all deadlines are met.

1 Introduction

Consider the problem of preemptively scheduling n sporadically arriving tasks on $m \geq 2$ identical processors. A task τ_i is uniquely indexed in the range $1..n$ and a processor likewise in the range $1..m$. A task τ_i generates a (potentially infinite) sequence of jobs. The arrival times of these jobs cannot be controlled by the scheduling algorithm and are a priori unknown. We assume that the arrival time between two successive jobs by the same task τ_i is at least T_i. Every job by τ_i requires at most C_i time units of execution over the next T_i time units after its arrival. We assume that T_i and C_i are real numbers and $0 \leq C_i \leq T_i$. A processor executes at most one job at a time and a job is not permitted to execute on multiple processors simultaneously. The utilization is defined as $U_s = (1/m) \cdot \sum_{i=1}^{n} \frac{C_i}{T_i}$. The utilization bound UB_A of an algorithm A is the maximum number such that all tasks meet their deadlines when scheduled by A, if $U_s \leq UB_A$.

Static-priority scheduling is a specific class of algorithms where each task is assigned a priority, a number which remains unchanged during the operation of the system. At every moment, the highest-priority task is selected for execution among tasks that are ready to execute and has remaining execution. Static-priority scheduling is simple to implement in operating systems and it can be implemented efficiently. Therefore, it is implemented in virtually all real-time operating systems and many desktop operating systems support it as well, accessible through system calls specified according to the POSIX-standard [1]. Because of these reasons, a comprehensive toolbox (see [2, 3]) of results (priority-assignment schemes, schedulability analysis algorithms, etc) has been developed

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 73–88, 2008.

for static-priority scheduling on a single processor. The success story of static-priority scheduling on a single processor started with the development of the rate-monotonic (RM) priority-assignment scheme [4]. It assigns task τ_j a higher priority than task τ_i if $T_j < T_i$. RM is an optimal priority-assignment scheme, meaning that for every task set, it holds that if there is an assignment of priorities that causes deadlines to be met then deadlines are met as well when RM is used. It is also known [4] that $UB_{RM} = 0.69$ for the case that $m = 1$. This result is important because it gives designers an intuitive idea of how much a processor can be utilized without missing a deadline.

Multiprocessor scheduling algorithms are often categorized as partitioned or global. Global scheduling stores tasks which have arrived but not finished execution in one queue, shared by all processors. At any moment, the m highest-priority tasks among those are selected for execution on the m processors. In contrast, partitioned scheduling algorithms partition the task set such that all tasks in a partition are assigned to the same processor. Tasks may not migrate from one processor to another. The multiprocessor scheduling problem is thus transformed to many uniprocessor scheduling problems.

Real-time scheduling on a multiprocessor is much less developed than real-time scheduling on a single processor. And this applies to static-priority scheduling as well. In particular, it is known that it is impossible to design a partitioned algorithm with $UB > 0.5$ [5]. It is also known that for global static-priority scheduling, RM is not optimal. In fact, global RM can miss a deadline although U_s approaches zero [6]. For a long time, the research community dismissed global static-priority scheduling for this reason. But later, it was realized that other priority-assignment schemes (not necessarily RM) can be used for global static-priority scheduling and the research community developed such schemes. Many priority-assignment schemes and analysis techniques for global static-priority scheduling are available (see for example [7, 8, 9, 10]) but so far, only two priority-assignment schemes, RM-US($m/(3m-2)$) [11] and RM-US(x) [12] have known (and non-zero) utilization bounds. These two algorithms categorize a task as heavy or light. A task is said to be heavy if $\frac{C_i}{T_i}$ exceeds a certain threshold number and a task is said to be light otherwise. Heavy tasks are assigned the highest priority and the light tasks are assigned a lower priority; the relative priority order among light tasks is given by RM. It was shown that among the algorithms that separate heavy and light tasks and use RM for light tasks, no algorithm can achieve a utilization bound greater than 0.374 [12]. And in fact, the current state-of-art offers no algorithm with utilization bound greater than 0.374.

In this paper, we present a new priority-assignment scheme SM-US($2/(3+\sqrt{5})$). It categorizes tasks as heavy and light and assigns the highest priority to heavy tasks. The relative priority order of light tasks is given by slack-monotonic (SM) though, meaning that task τ_j is assigned higher priority than task τ_i if T_j - $C_j < T_i$ - C_i. We prove that the utilization bound of SM-US($2/(3+\sqrt{5})$) is $2/(3+\sqrt{5})$, which is approximately 0.382.

We consider this result to be significant because (i) the new algorithm SM-US($2/(3+\sqrt{5})$) breaks free from the performance limitations of the RM-US framework, (ii) the utilization bound of SM-US($2/(3+\sqrt{5})$) is higher than the utilization bound of the previously-known best algorithm in global static-priority scheduling and (iii) the utilization bound of SM-US($2/(3+\sqrt{5})$) is reasonably close to the limit $\sqrt{2}-1 \approx 0.41$ which is known (from Theorem 8 in [13]) to be an upper bound on the utilization bound of every global static-priority scheduling algorithm which assigns a priority to a task τ_i as a function only of T_i and C_i.

Section 2 gives a background on the subject, presenting the main ideas behind algorithms that achieve a utilization bound greater than zero. It also presents results that we will use, in particular (i) lemmas expressing inequalities, (ii) a lemma from previous research on the amount of execution performed and (iii) a new schedulability test. Section 3 presents the new algorithm SM-US($2/(3+\sqrt{5})$) and proves its utilization bound using the schedulability test in Section 2. Conclusions are given in Section 4.

2 Background

2.1 Understanding Global Static-Priority Scheduling

The inventor of RM observed [14] that

> Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.

Example 1 gives a good illustration of this.

Example 1. [From [6]]. Consider a task set with n=m+1 tasks to be scheduled on m processors. The tasks are characterized as $\forall i \in \{1, 2, \ldots, m\} : T_i = 1, C_i = 2\epsilon$ and $T_{m+1} = 1 + \epsilon, C_{m+1} = 1$. If we assign priorities according to RM then τ_{m+1} is given the lowest priority and when all tasks arrive simultaneously then τ_{m+1} misses a deadline. Letting $\epsilon \to 0$ and $m \to \infty$ gives us a task set with $U_s \to 0$ and it misses a deadline.

Based on Example 1, one can see that better performance can be achieved by giving high priority to tasks with high $\frac{C_i}{T_i}$. And in fact this is what the algorithms, RM-US($m/(3m-2)$) [11] and RM-US(x) [12] do. The algorithm RM-US(x) [12] computes the value of x and its utilization bound is x. The value of x depends on the number of processors; it is given as (1-y)/($m \cdot$ (1+y))+ln(1+y)=(1-y)/(1+y)=x. Solving it for $m \to \infty$ gives us that y=0.454 and x=0.375. One can see that $m \to \infty$ gives us the least value of x. Hence the utilization bound of RM-US(0.375) is 0.375. And there is no other choice of x which gives a higher utilization bound. Example 2 illustrates this.

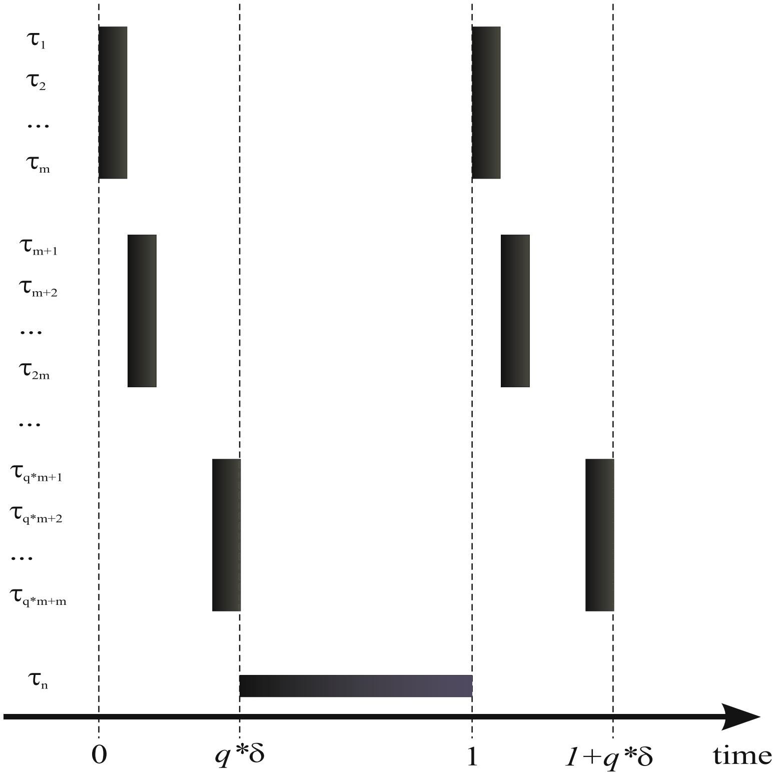


Fig. 1. An example of a task set where RM-US(0.375) performs poorly. All tasks arrive at time 0. Tasks τ_1, τ_2,. . ., τ_m are assigned the highest priority and execute on the m processors during [0,δ). Then the tasks τ_{m+1}, τ_{m+2},. . ., τ_{2m} execute on the m processors during [δ,2δ). The other groups of tasks execute in analogous manner. Task τ_n executes then until time 1. Then the groups of tasks arrive again. The task set meets its deadlines but an arbitrarily small increase in execution times causes a deadline miss.

Example 2. [Partially taken from [12]]. Figure 1 illustrates the example. Consider $n = m \cdot q + 1$ tasks to be scheduled on m processors, where q is a positive integer. The task τ_n is characterized by $T_n = 1 + y$ and $C_n = 1 - y$. The tasks with index $i \in \{1, 2, \ldots, n - 1\}$ are organized into groups, where each group comprises m tasks. One group is the tasks with index $i \in \{1, 2, \ldots, m\}$. Another group is the tasks with index $i \in \{m + 1, m + 2, \ldots, 2 \cdots m\}$ and so on. The r:th group comprises the tasks with index $i \in \{r \cdot m + 1, r \cdot m + 2, \ldots, r \cdot m + m\}$. All tasks belonging to the same group have the same T_i and C_i. Clearly there are q groups. The tasks in the r:th group have the parameters $T_i = 1 + r \cdot \delta$ and $C_i = \delta$, where δ is selected as $y = q \cdot \delta$. Hence, specifying m and y gives us the task set. By letting $y = 0.454$ and $m \to \infty$ we have a task set that where all tasks are light. The resulting task set is depicted in Figure 1. Also, all tasks meet their deadlines but an arbitrarily small increase in execution time of τ_n causes it to miss a deadline. That is, RM-US(0.375) misses a deadline at a utilization just slightly higher than 0.375.

One can see that if the light tasks in Example 2 would have been assigned priorities such that $T_j - C_j < T_i - C_i$ implies that τ_j has higher priority than τ_i then deadlines would have been met. In fact, we will use this idea when we design the new algorithm in Section 3.

2.2 Results We Will Use

Lemma 1-4 state four simple inequalities that we will find useful; their proofs are available in the Appendix.

Lemma 1. *Let m denote a positive integer. Consider u_i to be a real number such that $0 \leq u_i < \frac{2}{3+\sqrt{5}}$ and consider S to denote a set of non-negative real numbers u_j such that*

$$(\sum_{j \in S} u_j) + u_i \leq \frac{2}{3+\sqrt{5}} \cdot m \tag{1}$$

then it follows that

$$\frac{1}{m} \cdot (\sum_{j \in S} (2 - u_i) \cdot u_j) + u_i \leq 1 \tag{2}$$

Lemma 2. *Consider two non-negative real numbers u_j and u_i such that $0 \leq u_j < 1$ and $0 \leq u_i < 1$. For those numbers, it holds that:*

$$u_j \cdot \frac{1 - u_i}{1 - u_j} + (1 - u_j \cdot \frac{1 - u_i}{1 - u_j}) \cdot u_j \leq (2 - u_i) \cdot u_j \tag{3}$$

Lemma 3. *Consider two non-negative real numbers u_j and u_i such that $0 \leq u_j < 1$ and $0 \leq u_i < 1$. And two non-negative real numbers T_j and T_i such that*

$$T_j \cdot (1 - u_j) \leq T_i \cdot (1 - u_i) \tag{4}$$

For those numbers, it holds that:

$$u_j \cdot \frac{T_j}{T_i} + (1 - u_j \cdot \frac{T_j}{T_i}) \cdot u_j \leq u_j \cdot \frac{1 - u_i}{1 - u_j} + (1 - u_j \cdot \frac{1 - u_i}{1 - u_j}) \tag{5}$$

Lemma 4. *Consider two integers T_j and C_j such that $0 \leq C_j \leq T_j$. For every $t > 0$ it holds that:*

$$\lfloor \frac{t}{T_j} \rfloor \cdot C_j + \min(t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j, C_j) \leq C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{6}$$

Predictable scheduling. Ha and Liu [15] have studied real-time scheduling of jobs on a multiprocessor; a job is characterized by its arrival time, its deadline, its minimum execution time and its maximum execution time. The execution time of a job is unknown but it is no less than its minimum execution time and no greater than its maximum execution time. A scheduling algorithm A is said to be predictable if for every set J of jobs it holds that:

> Scheduling all jobs by A with execution times equal to their maximum execution times causes deadlines to be met. ⇒ Scheduling all jobs by A with execution times being at least their minimum execution times and at most their maximum execution times causes deadlines to be met.

Intuitively, the notion of predictability means that we only need to analyze the case when all jobs execute according to their maximum execution time. Ha and Liu also found that global static priority scheduling of jobs on a multiprocessor is predictable. Our paper deals with tasks that generate jobs with a certain constraint (given by the minimum inter-arrival time, T_i). But since our model is a special case of the model used by Ha and Liu, it also follows that global static-priority scheduling with our model is predictable as well.

The notion of active. We let $active(t, \tau_i)$ be true if at time t, there is a job of τ_i which has arrived no later than t and has a deadline no earlier than t; otherwise $active(t, \tau_i)$ is false. Observe that a task τ_i may release a job and at time t this job has no remaining execution but its deadline is greater than t. Because of our notion active, this task τ_i is active at time t. Note that with our notion of active, a periodically arriving task is active all the time after its first arrival. Because we study sporadically arriving tasks, there may be moments when a task is not active though. The notion of gap measures that.

The notion of gap. We let gap($[t_0,t_1)$, τ_i) denote the amount of time during $[t_0,t_1)$ where $active(t, \tau_i)$ is false.

Optimal algorithm. Consider a task τ_i and a time interval of duration ϵ such that the task τ_i is active during the entire time interval. Let OPT denote an algorithm which executes task τ_i for $(C_i/T_i)\cdot\epsilon$ time unit during the time interval of duration ϵ, where ϵ is arbitrarily small.

Work-conserving. We say that a scheduling algorithm is work-conserving if it holds for every t that: if there are at least k tasks with unfinished execution at time t then at least k processors are busy at time t. In particular, we note that global static-priority scheduling is work-conversing.

Execution. Let t_0 denote a time such that no tasks have arrived before t_0. Let W(A, τ, $[t_0,t_1)$) denote the amount of execution performed by tasks in τ during $[t_0,t_1)$ when scheduled by algorithm A. Philips et al. [16] studied the amount of execution performed by a work-conserving algorithm. They found that the amount of execution in a time interval performed by work-conserving algorithm is at least as much as the amount of execution performed by any other algorithm assuming that the work-conserving algorithm is given processors that are $(2m-1)/m$ times faster. Previous research [11] in real-time computing has used this result by comparing the amount of execution performed by global static-priority scheduling against the algorithm OPT but that work considered only the model of periodically arriving tasks. That result can be extended in a straightforward manner to the model we use in this paper (the sporadic model) though, as expressed by Lemma 5.

Lemma 5. *Let G denote an algorithm with global static-priority scheduling. If*

$$\forall j : \frac{C_j}{T_j} \leq \frac{m}{2m-1} \tag{7}$$

and

$$\sum_{\tau_j \in \tau} \frac{C_j}{T_j} \leq \frac{m}{2m-1} \cdot m \tag{8}$$

then

$$W(G, \tau, [t_0, t_1)) \geq \sum_{\tau_j \in \tau} (t_1 - t_0 - gap([t_0, t_1], \tau_j)) \cdot \frac{C_j}{T_j} \tag{9}$$

Proof. From Equation 7 and Equation 8 it follows that the task set τ can be scheduled to meet deadlines by OPT on a multiprocessor with m processors of speed $m/(2m-1)$. The amount of execution during $[t_0,t_1)$ is then given by the right-hand side of Equation 9. And the result by Philips et al gives us that also algorithm G performs as much execution during $[t_0,t_1)$. Hence Equation 9 is true and it gives us that the lemma is true.

Schedulability analysis. Let t_0 denote a time such that no tasks arrive before t_0. Let us consider a time interval that begins at time t_0; let $[t_0,\ t_2)$ denote this time interval. We obtain that the amount of execution performed by the task set τ during $[t_0,\ t_2)$ is at most:

$$\sum_{\tau_j \in hp(i)} \Big(\lfloor \frac{t_2 - t_0 - gap([t_0, t_2), \tau_j)}{T_j} \rfloor \cdot C_j + \\ \min(t_2 - t_0 - gap([t_0, t_2), \tau_j) - \lfloor \frac{t_2 - t_0 - gap([t_0, t_2), \tau_j)}{T_j} \rfloor \cdot T_j, C_j) \Big) \tag{10}$$

From Lemma 5 we obtain that the amount of execution performed by the task set τ during $[t_0,\ t_1)$ is at least:

$$\sum_{\tau_j \in hp(i)} (t_1 - t_0 - gap([t_0, t_1], \tau_j)) \cdot \frac{C_j}{T_j} \tag{11}$$

Let us consider the case that a deadline was missed. Let us consider the earliest time when a deadline was missed. Let t_1 denote the arrival time of the job that missed this deadline and let τ_i denote the task that generated this job. Let $hp(i)$ denote the set of tasks with higher priority than τ_i. Let t_2 denote the deadline that was missed; that is, $t_2{=}t_1{+}T_i$. Applying Equation 8 and Equation 9 on $hp(i)$ gives us that the amount of execution by $hp(i)$ during $[t_1,t_2)$ is at most:

$$\sum_{\tau_j \in hp(i)} \Big(\lfloor \frac{t_2 - t_0 - gap([t_0, t_2), \tau_j)}{T_j} \rfloor \cdot C_j + \\ \min(t_2 - t_0 - gap([t_0, t_2), \tau_j) - \lfloor \frac{t_2 - t_0 - gap([t_0, t_2), \tau_j)}{T_j} \rfloor \cdot T_j, C_j) \Big) \\ - \sum_{\tau_j \in hp(i)} (t_1 - t_0 - gap([t_0, t_1], \tau_j)) \cdot \frac{C_j}{T_j} \tag{12}$$

Using $t_2 = t_1 + T_i$ and rewriting gives us that the amount of execution by $hp(i)$ during $[t_1,t_2)$ is at most:

$$\begin{aligned}\sum_{\tau_j \in hp(i)} \Big(\lfloor \frac{T_i + t_1 - t_0 - gap([t_0,t_1),\tau_j) - gap([t_1,t_2),\tau_j)}{T_j} \rfloor \cdot C_j + \\ \min(T_i + t_1 - t_0 - gap([t_0,t_1),\tau_j) - gap([t_1,t_2),\tau_j) - \\ \lfloor \frac{T_i + t_1 - t_0 - gap([t_0,t_1),\tau_j) - gap([t_1,t_2),\tau_j)}{T_j} \rfloor \cdot T_j, C_j) \Big) \\ - \sum_{\tau_j \in hp(i)} (t_1 - t_0 - gap([t_0,t_1],\tau_j)) \cdot \frac{C_j}{T_j}\end{aligned} \tag{13}$$

Applying Lemma 4 on Equation 13 gives us that the amount of execution by $hp(i)$ during $[t_1,t_2)$ is at most:

$$\begin{aligned}\sum_{\tau_j \in hp(i)} \Big(C_j + (T_i + t_1 - t_0 - gap([t_0,t_1),\tau_j) - gap([t_1,t_2),\tau_j) - C_j) \cdot \frac{C_j}{T_j} \Big) \\ - \sum_{\tau_j \in hp(i)} (t_1 - t_0 - gap([t_0,t_1],\tau_j)) \cdot \frac{C_j}{T_j}\end{aligned} \tag{14}$$

Simplifying Equation 14 gives us that the amount of execution by $hp(i)$ during $[t_1,t_2)$ is at most:

$$\sum_{\tau_j \in hp(i)} \Big(C_j + (T_i - gap([t_1,t_2),\tau_j) - C_j) \cdot \frac{C_j}{T_j} \Big) \tag{15}$$

Relaxing gives that the amount of execution by tasks in $hp(i)$ during $[t_1,t_2)$ is at most:

$$\sum_{\tau_j \in hp(i)} \Big(C_j + (T_i - C_j) \cdot \frac{C_j}{T_j} \Big) \tag{16}$$

From Equation 16 it follows that the amount of time during during $[t_1,t_2)$ where all processors are busy executing tasks in $hp(i)$ is at most:

$$\frac{1}{m} \cdot \sum_{\tau_j \in hp(i)} \Big(C_j + (T_i - C_j) \cdot \frac{C_j}{T_j} \Big) \tag{17}$$

Lemma 6. *Consider global static-priority scheduling. Consider a task τ_i. If all tasks in $hp(i)$ meet their deadlines and*

$$\forall j \in hp(i) : \frac{C_j}{T_j} \leq \frac{m}{2m-1} \tag{18}$$

and

$$\frac{C_i}{T_i} \leq \frac{m}{2m-1} \cdot m \tag{19}$$

and

$$\Big(\sum_{\tau_j \in hp(i)} \frac{C_j}{T_j} \Big) + \frac{C_i}{T_i} \leq \frac{m}{2m-1} \cdot m \tag{20}$$

and

$$\frac{1}{m} \cdot \Big(\sum_{\tau_j \in hp(i)} \Big(C_j + (T_i - C_j) \cdot \frac{C_j}{T_j} \Big) \Big) + C_i \leq T_i \tag{21}$$

then all deadline of τ_i are met.

Proof. Follows from the discussion above.

3 The New Algorithm

Section 3.1 presents Slack-monotonic (SM) scheduling and analyzes its performance for restricted task sets (called light tasks). This restriction is then removed in Section 3.2; the new algorithm is presented and its utilization bound is proven.

3.1 Light Tasks

We say that a task τ_i is light if $\frac{C_i}{T_i} \leq \frac{2}{3+\sqrt{5}}$. We let Slack-Monotonic (SM) denote a priority assignment scheme which assigns priorities such that task τ_j is assigned higher priority than task τ_i if $T_j - C_j < T_i - C_i$.

Lemma 7. *Consider global static-priority scheduling with SM. Consider a task i. If all tasks in hp(i) meet their deadlines and*

$$\forall j \in hp(i) : \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \tag{22}$$

and

$$\frac{C_i}{T_i} \leq \frac{2}{3+\sqrt{5}} \tag{23}$$

and

$$(\sum_{\tau_j \in hp(i)} \frac{C_j}{T_j}) + \frac{C_i}{T_i} \leq \frac{2}{3+\sqrt{5}} \cdot m \tag{24}$$

then all deadline of τ_i are met.

Proof. The Inequalities 22,23 and 24 imply that Inequalities 18,19 and 20 are true. Applying Lemma 1 on Inequalities 24 gives us:

$$\frac{1}{m} \cdot (\sum_{j \in hp(i)} (2 - \frac{C_i}{T_i}) \cdot \frac{C_j}{T_j}) + \frac{C_i}{T_i} \leq 1 \tag{25}$$

Applying Lemma 2 on Inequalities 25 gives us:

$$\frac{1}{m} \cdot \left(\sum_{j \in hp(i)} \left(\frac{C_j}{T_j} \cdot \frac{1-\frac{C_i}{T_i}}{1-\frac{C_j}{T_j}} + (1 - \frac{C_j}{T_j} \cdot \frac{1-\frac{C_i}{T_i}}{1-\frac{C_j}{T_j}}) \cdot \frac{C_j}{T_j}\right) \right) + \frac{C_i}{T_i} \leq 1 \tag{26}$$

From the fact that SM is used we obtain that

$$\forall j \in hp(i) : T_j - C_j < T_i - C_i \tag{27}$$

Considering Inequality 26 and Inequality 27 and Lemma 3 gives us:

$$\frac{1}{m} \cdot \left(\sum_{j \in hp(i)} \left(\frac{C_j}{T_j} \cdot \frac{T_j}{T_i} + (1 - \frac{C_j}{T_j} \cdot \frac{T_j}{T_i}) \cdot \frac{C_j}{T_j}\right) \right) + \frac{C_i}{T_i} \leq 1 \tag{28}$$

Multiplying both the left-hand side and the right-hand side of Inequality 28 by T_i and rewriting yields:

$$\frac{1}{m} \cdot \left(\sum_{j \in hp(i)} \left(C_j + (T_i - C_j) \cdot \frac{C_j}{T_j}\right) \right) + C_i \leq T_i \tag{29}$$

Using Inequality 29 and Lemma 6 gives us that all deadline of τ_i are met. This states the lemma.

Lemma 8. *Consider global static-priority scheduling with SM. If it holds for the task set that*

$$\forall \tau_j \in \tau : \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \tag{30}$$

and

$$\sum_{\tau_j \in \tau} \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \cdot m \tag{31}$$

then all deadline of τ_i are met.

Proof. Follows from Lemma 7.

3.2 Light and Heavy Tasks

We say that a task is heavy if it is not light. We let the algorithm SM-US$(2/(3+\sqrt{5}))$ denote a priority assignment scheme which assigns the highest priority to heavy tasks and assigns a lower priority to light tasks; the priority order between light tasks is given by SM.

Theorem 1. *Consider global static-priority scheduling with SM-US$(2/(3+\sqrt{5}))$. If it holds for the task set that*

$$\forall \tau_j \in \tau : \frac{C_j}{T_j} \leq 1 \tag{32}$$

and

$$\sum_{\tau_j \in \tau} \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \cdot m \tag{33}$$

then all deadlines are met.

Proof. The proof is by contradiction. If the lemma was false then it follows that there is a task set such that Inequality 32 and Inequality 33 are true and when this task set was scheduled by SM-US($2/(3+\sqrt{5})$) a deadline was missed. Let τ^{failed} failed denote this task set and let m denote the number of processors. Let k denote the number of heavy tasks. Because of Inequality 33 it follows that $k \leq m$. Also, because of Lemma 8 is follows that $k \geq 1$.

Let $\tau^{failed2}$ denote a set which is constructed from τ^{failed} as follows. For every light task in τ^{failed} there is a light task in $\tau^{failed2}$ and their T_i and C_i are the same. For every heavy task in τ^{failed} there is a heavy task in $\tau^{failed2}$ and its T_i is the same. For the heavy tasks in $\tau^{failed2}$ it holds that $C_i = T_i$. From Inequality 33 it follows that

$$\sum_{\tau_j \in light(\tau^{failed})} \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \cdot (m-k) \tag{34}$$

where $light(\tau^{failed})$ denotes the set of light tasks in τ^{failed}. Since the light tasks are the same in τ^{failed} and $\tau^{failed2}$ it clearly follows that

$$\sum_{\tau_j \in light(\tau^{failed2})} \frac{C_j}{T_j} \leq \frac{2}{3+\sqrt{5}} \cdot (m-k) \tag{35}$$

If the task set $\tau^{failed2}$ would meet all deadlines when scheduled by SM-US($2/(3+\sqrt{5})$) then it would follow (from the fact that global static-priority scheduling is predictable) that all deadlines would have been met when τ^{failed} was scheduled by SM-US($2/(3+\sqrt{5})$). Hence it follows that at least one deadline was missed by $\tau^{failed2}$. And since there are at most $k \leq m-1$ heavy tasks it follows that no deadline miss occurs for the heavy tasks. Hence it must have been that a deadline miss occurred from a light task in $\tau^{failed2}$. But the scheduling of the light tasks in $\tau^{failed2}$ is identical to what is would have been if we deleted the heavy tasks in $\tau^{failed2}$ and deleted the k processors. That is, we have that scheduling the light tasks on $m-k$ processor causes a deadline miss. But Inequality 35 and Lemma 8 gives that no deadline miss occurs. This is a contradiction. Hence the theorem is correct.

4 Conclusions

We have presented a new priority-assignment scheme, SM-US($2/(3+\sqrt{5})$), for global static-priority multiprocessor scheduling and proven that its utilization bound is $2/(3+\sqrt{5}$, which is approximately, 0.382. We left open the question whether it is possible to achieve a utilization bound of $\sqrt{2}-1$ with global static-priority scheduling.

Acknowledgements

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the ARTIST2 Network of Excellence on Embedded Systems Design.

References

[1] Gallmeister, B.: POSIX.4 Programmers Guide: Programming for the Real World. O'Reilly Media, Sebastopol (1995)

[2] Sha, L., Rajkumar, R., Sathaye, S.: Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. Proceedings of the IEEE 82, 68–82 (1994)

[3] Tindell, K.W.: An Extensible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Technical Report, Department of Computer Science, University of York, UK YCS 189 (1992)

[4] Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Technical Report, Department of Computer Science, University of York, UK YCS 189., 1992. Journal of the ACM, vol. 20, pp. 46–61 (1973)

[5] Oh, D., Baker, T.P.: Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. Real-Time Systems 5, 183–192 (1998)

[6] Dhall, S., Liu, C.: On a real-time scheduling problem. Operations Research 6, 127–140 (1978)

[7] Baker, T.P.: An Analysis of Fixed-Priority Schedulability on a Multiprocessor. Real-Time Systems 32, 49–71 (2006)

[8] Bertogna, M., Cirinei, M.: Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In: IEEE Real-Time Systems Symposium, Tucson, Arizona (2007)

[9] Bertogna, M., Cirinei, M., Lipari, G.: New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors. In: 9th International Conference on Principles of Distributed Systems, Pisa, Italy (2005)

[10] Cucu, L.: Optimal priority assignment for periodic tasks on unrelated processors. In: Euromicro Conference on Real-Time Systems (ECRTS 2008), WIP session, Prague, Czech Republic (2008)

[11] Andersson, B., Baruah, S., Jonsson, J.: Static-Priority Scheduling on Multiprocessors. In: IEEE Real-Time Systems Symposium, London, UK (2001)

[12] Lundberg, L.: Analyzing Fixed-Priority Global Multiprocessor Scheduling. In: Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002) (2002)

[13] Andersson, B., Jonsson, J.: The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In: Euromicro Conference on Real-Time Systems, Porto, Portugal (2003)

[14] Liu, C.L.: Scheduling algorithms for multiprocessors in a hard real-time environment. JPL Space Programs Summary 37-60, 28–31 (1969)

[15] Ha, R., Liu, J.W.S.: Validating timing constraints in multiprocessor and distributed real-time systems. In: Proceedings of the 14th International Conference on Distributed Computing Systems, Pozman, Poland (1994)

[16] Phillips, C.A., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation. In: ACM Symposium on Theory of Computing, El Paso, Texas, United States (1997)

Appendix

Lemma 1. *Let m denote a positive integer. Consider u_i to be a real number such that $0 \leq u_i < \frac{2}{3+\sqrt{5}}$ and consider S to denote a set of non-negative real numbers u_j such that*

$$(\sum_{j\in S} u_j) + u_i \leq \frac{2}{3+\sqrt{5}} \cdot m \tag{36}$$

then it follows that

$$\frac{1}{m} \cdot (\sum_{j\in S}(2-u_i)\cdot u_j) + u_i \leq 1 \tag{37}$$

Proof. Let us define f as:

$$f = (2-u_i)\cdot \frac{2}{3+\sqrt{5}} \cdot m + m \cdot u_i - m - u_i + \frac{2}{3+\sqrt{5}} \tag{38}$$

We have:

$$\frac{\partial f}{\partial u_i} = -\frac{2}{3+\sqrt{5}} \cdot m + m - 1 > 0 \tag{39}$$

From Inequality 39 and the constraint $u_i \leq \frac{2}{3+\sqrt{5}}$ we obtain that f is no greater than f for the value $u_i = \frac{2}{3+\sqrt{5}}$. And we have $f(u_i = \frac{2}{3+\sqrt{5}}) = 0$. This gives us:

$$f \leq (2-u_i)\cdot \frac{2}{3+\sqrt{5}} \cdot m + m \cdot u_i - m - u_i + \frac{2}{3+\sqrt{5}} \leq 0 \tag{40}$$

Applying Inequality 40 to Inequality 36 and rewriting yields:

$$(2-u_i)\cdot \left((\sum_{j\in S} u_j) + u_i\right) + m\cdot u_i - u_i + \frac{2}{3+\sqrt{5}} \leq m \tag{41}$$

Rearranging terms in Inequality 41 gives us:

$$\frac{1}{m}\cdot\left(\sum_{j\in S}(2-u_i)\cdot u_j\right) + u_i + \frac{(2-u_i)\cdot u_i - u_i + \frac{2}{3+\sqrt{5}}}{m} \leq 1 \tag{42}$$

Recall that $u_i \leq \frac{2}{3+\sqrt{5}}$. Clearly this gives us $2 - u_i \geq 1$. And hence the last term in the left-hand side of Inequality 42 is non-negative. This gives us:

$$\frac{1}{m}\cdot\left(\sum_{j\in S}(2-u_i)\cdot u_j\right) + u_i \leq 1 \tag{43}$$

And this states the lemma. Hence the lemma is correct.

Lemma 2. *Consider two non-negative real numbers u_j and u_i such that $0 \leq u_j < 1$ and $0 \leq u_i < 1$. For those numbers, it holds that:*

$$u_j \cdot \frac{1-u_i}{1-u_j} + (1 - u_j \cdot \frac{1-u_i}{1-u_j}) \cdot u_j \leq (2-u_i)\cdot u_j \tag{44}$$

Proof. The proof is by contradiction. Suppose that the lemma is false. Then we have:

$$u_j \cdot \frac{1-u_i}{1-u_j} + (1 - u_j \cdot \frac{1-u_i}{1-u_j}) \cdot u_j > (2-u_i) \cdot u_j \tag{45}$$

Let us explore the following cases.

1. $u_i = 0$ and $u_j = 0$
   Applying this case on Inequality 45 gives us:
   $$0 > 0 \tag{46}$$
   which is a contradiction. (end of Case 1)
2. $u_i = 0$ and $u_j > 0$
   Applying this case on Inequality 45 gives us:
   $$u_j \cdot \frac{1}{1-u_j} + (1 - u_j \cdot \frac{1}{1-u_j}) \cdot u_j > 2 \cdot u_j \tag{47}$$
   Since $u_j > 0$ we can divide Inequality 47 by u_j and this gives us:
   $$\frac{1}{1-u_j} + 1 - u_j \cdot \frac{1}{1-u_j} > 2 \tag{48}$$
   Rewriting Inequality 48 yields:
   $$\frac{1}{1-u_j} \cdot (1-u_j) > 1 \tag{49}$$
   which is a contradiction. (end of Case 2)
3. $u_i > 0$ and $u_j = 0$
   Applying this case on Inequality 45 gives us:
   $$0 > 0 \tag{50}$$
   which is a contradiction. (end of Case 3)
4. $u_i > 0$ and $u_j > 0$
   Since $u_j > 0$ we can divide Inequality 45 by u_j and this gives us:
   $$\frac{1-u_i}{1-u_j} + (1 - u_j \cdot \frac{1-u_i}{1-u_j}) > 2 - u_i \tag{51}$$
   Rewriting Inequality 51 yields:
   $$\frac{1-u_i}{1-u_j} - u_j \cdot \frac{1-u_i}{1-u_j} > 1 - u_i \tag{52}$$
   Further rewriting yields:
   $$\frac{1}{1-u_j} - u_j \cdot \frac{1}{1-u_j} > 1 \tag{53}$$
   Further rewriting yields:
   $$1 > 1 \tag{54}$$
   which is a contradiction. (end of Case 4)

Since a contradiction occurs for every case we obtain that the lemma is false.

Lemma 3. *Consider two non-negative real numbers u_j and u_i such that $0 \leq u_j < 1$ and $0 \leq u_i < 1$. And two non-negative real numbers T_j and T_i such that*

$$T_j \cdot (1-u_j) \leq T_i \cdot (1-u_i) \tag{55}$$

For those numbers, it holds that:

$$u_j \cdot \frac{T_j}{T_i} + (1 - u_j \cdot \frac{T_j}{T_i}) \cdot u_j \leq u_j \cdot \frac{1-u_i}{1-u_j} + (1 - u_j \cdot \frac{1-u_i}{1-u_j}) \tag{56}$$

Proof. Rewriting Inequality 55 yields:

$$\forall j \in hp(i) : \frac{T_j}{T_i} \leq \frac{1-u_i}{1-u_j} \tag{57}$$

Let $q_{i,j}$ denote the left-hand side of Inequality 57. There are two occurrences $q_{i,j}$ in the left-hand side of Inequality 56. Also observe that the left-hand side of Inequality 56 is increasing with increasing $q_{i,j}$. For this reason, combining Inequality 57 and the left-hand side of inequality 56 gives us that the lemma is true.

Lemma 4. *Consider two integers T_j and C_j such that $0 \leq C_j \leq T_j$. For every $t > 0$ it holds that:*

$$\lfloor \frac{t}{T_j} \rfloor \cdot C_j + \min(t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j, C_j) \leq C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{58}$$

Proof. The proof is by contradiction. Suppose that the lemma is false. Then there is a $t > 0$ such that:

$$\lfloor \frac{t}{T_j} \rfloor \cdot C_j + \min(t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j, C_j) > C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{59}$$

Let us consider two cases:

1. $t - \lfloor t/T_j \rfloor \cdot T_j \leq C_j$
   Let Δ be defined as: $\Delta = C_i - (t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j)$. Let us increase t by Δ. Then the left-hand side of Inequality 59 increases by Δ and the right-hand side increases by $(C_j/T_j) \cdot \Delta$. Since $C_j/T_j \leq 1$ it follows that Inequality 59 still true. That is:

$$\lfloor \frac{t}{T_j} \rfloor \cdot C_j + \min(t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j, C_j) > C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{60}$$

   Repeating this argument gives us that $t - \lfloor t/T_j \rfloor \cdot T_j = C_j$. Applying it on Inequality 60 yields:

$$\frac{t - C_j}{T_j} \cdot C_j + C_j > C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{61}$$

   Rewriting Inequality 61 gives us:

$$(t - C_j) > (t - C_j) \tag{62}$$

   which is impossible. (end of Case 1)

2. $t - \lfloor t/T_j \rfloor \cdot T_j \geq C_j$
   Let Δ be defined as: $\Delta = (t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j) - C_j$. Let us decrease t by Δ. Then the left-hand side of Inequality 59 is unchanged and the right-hand side decreases by $(C_j/T_j) \cdot \Delta$. Since $0 \leq C_j/T_j$ it follows that Inequality 59 still true. That is:

$$\lfloor \frac{t}{T_j} \rfloor \cdot C_j + \min(t - \lfloor \frac{t}{T_j} \rfloor \cdot T_j, C_j) > C_j + (t - C_j) \cdot \frac{C_j}{T_j} \tag{63}$$

   Repeating this argument gives us that $t - \lfloor t/T_j \rfloor \cdot T_j = C_j$. Applying it on Inequality 63 and applying similar rewriting as in Inequality 61 and Inequality 62 yields:

$$(t - C_j) > (t - C_j) \tag{64}$$

   which is impossible. (end of Case 2)

We can see that regardless of which case occurs a contradiction occurs and hence the lemma is correct.

Deadline Monotonic Scheduling on Uniform Multiprocessors*

Sanjoy Baruah[1] and Joël Goossens[2]

[1] University of North Carolina at Chapel Hill, NC, USA
baruah@cs.unc.edu
[2] Université Libre de Bruxelles, Brussels, Belgium
joel.goossens@ulb.ac.be

Abstract. The scheduling of sporadic task systems upon uniform multiprocessor platforms using global Deadline Monotonic algorithm is studied. A sufficient schedulability test is presented and proved correct. It is shown that this test offers non-trivial quantitative guarantees, in the form of a processor speedup bound.

1 Introduction

A multiprocessor computer platform is comprised of several processors. A platform in which all the processors have the same capabilities is referred to as an *identical* multiprocessor, while those in which different processors have different capabilities are called *heterogeneous* multiprocessors. Heterogeneous multiprocessors may be further classified into *uniform* and *unrelated* multiprocessors. The only difference between the different processors in a uniform multiprocessor is the rate at which they can execute work: each processor is characterized by a speed or computing capacity parameter s, and any job executing on the processor for t time units completes $t \times s$ units of execution. In unrelated multiprocessors, on the other hand, the amount of execution completed by a particular job executing on a given processor depends upon the identities of both the job and the processor.

A real-time system is often modeled as a finite collection of independent recurrent tasks, each of which generates a potentially infinite sequence of jobs. Every job is characterized by an arrival time, an execution requirement, and a deadline, and it is required that a job completes execution between its arrival time and its deadline. Different formal models for recurring tasks place different restrictions on the values of the parameters of jobs generated by each task. One of the more commonly used formal models is the *sporadic task model* [1,2]. Each recurrent task τ_i in this model is characterized by three parameters: $\tau_i = (C_i, D_i, T_i)$, with the interpretation that τ_i may generate an infinite sequence of jobs with successive jobs arriving at least T_i time units apart, each with an execution

* Supported in part by NSF Grant Nos. CNS-0834270, CNS-0834132, CCF-0541056, and CCR-0615197, ARO Grant No. W911NF-06-1-0425, and funding from IBM and the Intel Corporation.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 89–104, 2008.

requirement at most C_i and a deadline D_i time units after its arrival time. A sporadic task system τ is comprised of a finite collection of such sporadic tasks. Sporadic task systems in which each task is required to have its relative deadline and period parameters the same ($D_i = T_i$ for all i) are called *implicit-deadline* task systems, and ones in which each task is required to have its relative deadline be no larger than its period parameter ($D_i \leq T_i$ for all i) are called *constrained-deadline* task systems. A task system that is not constrained-deadline is said to be an *arbitrary-deadline* task system.

Several results have been obtained over the past decade, concerning the scheduling of *implicit-deadline* systems on *identical* [3,4,5,6,7,8,9] and on *uniform* [10,11,12,13,14,15,16,17] multiprocessors, of *constrained-deadline* systems on *identical* multiprocessors [18, 19, 20, 21, 22, 23, 24], and of *arbitrary-deadline* systems on *identical* multiprocessors, [25, 26]. This paper seeks to extend this body of work, by addressing *the scheduling of constrained and arbitrary-deadline sporadic task systems upon uniform multiprocessors.* We assume that the platform is fully *preemptive* — an executing job may be interrupted at any instant in time and have its execution resumed later with no cost or penalty. We study the behavior of the well-known and very widely-used *Deadline Monotonic* scheduling algorithm [27] when scheduling systems of sporadic tasks upon such preemptive platforms. We will refer to Deadline Monotonic scheduling with global interprocessor migration as *global* DM (or simply DM).

Contributions. We obtain a new test – to our knowledge, this is the first such tests – for determining whether a given constrained or arbitrary-deadline sporadic task system is guaranteed to meet all deadlines upon a specified uniform multiprocessor platform, when scheduled using DM. This test is derived by applying techniques that have previously been used for the schedulability analysis of constrained-deadline task systems on uniform multiprocessors when scheduled using EDF [28] and by integrating techniques used for schedulability analysis of sporadic arbitrary-deadline systems on identical multiprocessors using DM [25].

Organization. The remainder of this paper is organized as follows. In Sect. 2 we formally define the sporadic task model and uniform multiprocessor platforms, and provide some additional useful definitions, notation, and terminology concerning sporadic tasks and uniform multiprocessors. We also provide a specification of the behavior of global DM is to be implemented upon uniform multiprocessors. In Sect. 3 we derive, and prove the correctness of, a schedulability test for determining whether a given sporadic task system is DM-schedulable on a specified uniform multiprocessor platform. In Sect. 4 we provide a quantitative characterization of the efficacy of this new schedulability test in terms of the resource augmentation metric.

2 Task and Platform Model

§1. Sporadic task systems. A *sporadic task* $\tau_i = (C_i, D_i, T_i)$ is characterized by a *worst-case execution requirement* C_i, a *(relative) deadline* D_i, and a *minimum*

inter-arrival separation parameter T_i, also referred to as the *period* of the task. Such a sporadic task generates a potentially infinite (legal) sequence of jobs, with successive job-arrivals separated by at least T_i time units. Each job has a worst-case execution requirement equal to C_i and a deadline that occurs D_i time units after its arrival time. We refer to the interval, of size D_i, between such a job's arrival instant and deadline as its *scheduling window.* We assume a fully *preemptive* execution model: any executing job may be interrupted at any instant in time, and its execution resumed later with no cost or penalty. A *sporadic task system* is comprised of a finite number of such sporadic tasks. Let τ denote a system of such sporadic tasks: $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$, with $\tau_i = (C_i, D_i, T_i)$ for all i, $1 \leq i \leq n$. Without loss of generality, we assume that *tasks are indexed in non-increasing order of their relative deadline parameters*: $D_i \leq D_{i+1} (\forall\, i \in [1, n-1])$.

We find it convenient to define some properties and parameters for individual sporadic tasks, and for sporadic task systems.

Utilization: The utilization u_i of a task τ_i is the ratio C_i/T_i of its execution requirement to its period. The total utilization $u_{\text{sum}}(\tau)$ and the largest utilization $u_{\max}(\tau)$ of a task system τ are defined as follows:

$$u_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i; \qquad u_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau}(u_i) \enspace .$$

Density: The density δ_i of a task τ_i is the ratio $(C_i/\min(D_i, T_i))$ of its execution requirement to the smaller of its relative deadline and its period. The total density $\delta_{\text{sum}}(\tau)$ of a task system τ is defined as follows:

$$\delta_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \delta_i \enspace .$$

For each k, $1 \leq k \leq n$, $\delta_{\max}(k)$ denotes the *largest density* from among the tasks $\tau_1, \tau_2, \ldots, \tau_k$:

$$\delta_{\max}(k) \stackrel{\text{def}}{=} \max_{i=1}^{k}(\delta_i) \enspace .$$

DBF: For any interval length t, the demand bound function $\text{DBF}(\tau_i, t)$ of a sporadic task τ_i bounds the maximum cumulative execution requirement by jobs of τ_i that both arrive in, and have deadlines within, any interval of length t. It has been shown [2] that

$$\text{DBF}(\tau_i, t) = \max\left(0, (\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1)\, C_i\right) \enspace .$$

Load: A load parameter, based upon the DBF function, may be defined for any sporadic task system τ as follows:

$$\text{LOAD}(k) \stackrel{\text{def}}{=} \max_{t>0}\left(\frac{\sum_{i=1}^{k} \text{DBF}(\tau_i, t)}{t}\right) \enspace .$$

Computing DBF (and thereby, LOAD) will turn out to be a critical component of the schedulability analysis test proposed in this paper; hence, it is important that DBF be efficiently computable if this schedulability test is to be efficiently implementable as claimed. Fortunately, computing DBF is a well-studied subject, and algorithms are known for computing DBF *exactly* [2,29], or *approximately* to any arbitrary degree of accuracy [30,31,32].

The following Lemma relates the density of a task to its DBF:

Lemma 1 ([25]). *For all tasks τ_i and for all $t \geq 0$,*

$t \times \delta_i \geq \text{DBF}(\tau_i, t)$. □

In constrained task systems — those in which $D_i \leq T_i \; \forall i$ — a job becomes eligible to execute upon arrival, and remains eligible until it completes execution[1]. In systems with $D_i > T_i$ for some tasks τ_i, we require that at most one job of each task be eligible to execute at each time instant. We assume that jobs of the same task are considered in first-come first-served order; hence, a job only becomes eligible to execute after both these conditions are satisfied: *(i)* it has arrived, and *(ii)* all previous jobs generated by the same task that generated it have completed execution. This gives rise to the notion of an active task: briefly, a task is active at some instant if it has some eligible job awaiting execution at that instant. More formally,

Definition 1 (active task). *A task is said to be* active *in a given schedule at a time-instant t if some job of the task is eligible to execute at time-instant t. That is,* (i) *$t \geq$ the greater of the job's arrival time and the completion time of the previous job of the same task, and* (ii) *the job has not completed execution prior to time-instant t.* □

§*2. Uniform multiprocessors.* A uniform multiprocessor $\pi = (s_1, s_2, \ldots, s_m)$ is comprised of $m > 1$ processors, with the i'th processor characterized by *speed* or *computing capacity* s_i. The interpretation is that a job executing on the i'th processor for a duration of t units of time completes $t \times s_i$ units of execution. Without loss of generality, we assume that the speeds are specified in non-increasing order: $s_i \geq s_{i+1}$ for all i. We will also use the following notation:

$$S_i(\pi) \stackrel{\text{def}}{=} \sum_{j=1}^{i} s_j \quad . \tag{1}$$

That is, $S_i(\pi)$ denotes the sum of the computing capacities of the i fastest processors in π (and $S_m(\pi)$ hence denotes the total computing capacity of π).

An additional parameter that turns out to be useful in describing the properties of a uniform multiprocessor is the "lambda" parameter [12,10]:

$$\lambda(\pi) \stackrel{\text{def}}{=} \max_{i=1}^{m} \frac{\sum_{j=i+1}^{m} s_j}{s_i} \quad . \tag{2}$$

[1] Or its deadline has elapsed, in which case the system is deemed to have failed.

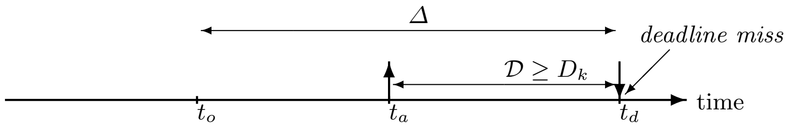


Fig. 1. Notation. A job of task τ_k arrives at t_a. Task τ_k is not active immediately prior to t_a, and is continually active over $[t_a, t_d)$.

This parameter ranges in value between 0 and $(m-1)$ for an m-processor uniform multiprocessor platform, with a value of $(m-1)$ corresponding to the degenerate case when all the processors are of the same speed (i.e., the platform is an identical multiprocessor).

§*3. Deadline Monotonic scheduling.* Priority-driven scheduling algorithms operate on uniform multiprocessors as follows: at each instant in time they assign a priority to each job that is awaiting execution, and favor for execution the jobs with the greatest priorities. Specifically, *(i)* no processor is idled while there is an active job awaiting execution; *(ii)* when there are fewer active jobs than processors, the jobs execute on the fastest processors and the slowest ones are idled; and *(iii)* greater-priority jobs execute on the faster processors. The Deadline Monotonic (DM) scheduling algorithm [33] is a priority-driven scheduling algorithm that assigns priority to tasks according to their (relative) deadlines: the smaller the deadline, the greater the priority.

With respect to a specified platform, a given sporadic task system is said to be *feasible* if there exists a schedule meeting all deadlines for every collection of jobs that may be generated by the task system. A given sporadic task system is said to be *(global)* DM *schedulable* if DM meets all deadlines for every collection of jobs that may be generated by the task system.

3 A DM Schedulability Test for Sporadic Task Systems

We now derive (Theorem 1) a sufficient condition for determining whether a sporadic task system τ is DM-schedulable upon a specified uniform multiprocessor platform π. This sufficient schedulability condition is in terms of the load and maximum density parameters — the LOAD(k)'s and the $\delta_{\max}(k)$'s defined above — of the task system, and the total computing capacity and the lambda parameter — $S_m(\pi)$ and $\lambda(\pi)$ defined above — of the platform.

Our strategy for deriving, and proving the correctness of, our sufficient schedulability condition is the following: for any legal sequence of job requests of task system τ, on which DM misses a deadline we obtain a *necessary* condition for that deadline miss to occur by bounding from above the total amount of execution that the DM schedule needs (but fails) to execute before the deadline miss. Negating this condition yields a *sufficient* condition for global-DM schedulability.

Consider any legal sequence of job requests of task system τ, on which DM misses a deadline. Suppose that a job of task τ_k is the one to first miss a deadline, and that this deadline miss occurs at time-instant t_d (see Fig. 1).

Discard from the legal sequence of job requests all jobs of tasks with priority lower than τ_k's, and consider the DM schedule of the remaining (legal) sequence of job requests. Since lower-priority jobs have no effect on the scheduling of greater-priority ones under preemptive DM, it follows that a deadline miss of τ_k occurs at time-instant t_d (and this is the earliest deadline miss), in this new DM schedule. We will focus henceforth on this new DM schedule.

Let t_a denote the earliest time-instant prior to t_d, such that τ_k is continuously active[2] over the interval $[t_a, t_d]$. It must be the case that t_a is the arrival time of some job of τ_k since τ_k is, by definition, not active just prior to t_a and becomes active at t_a.

It must also be the case that $t_a \leq t_d - D_k$. This follows from the observation that the job of τ_k that misses its deadline at t_d arrives at $t_d - D_k$. If $D_k < T_k$, then t_a is equal to this arrival time of the job of τ_i that misses its deadline at t_d. If $D_k \geq T_k$, however, t_k may be the arrival-time of an earlier job of τ_k. Let $\mathcal{D} \stackrel{\text{def}}{=} t_d - t_a$.

Let $\mathcal{C}$ denote the cumulative execution requirement of all jobs of τ_k that arrive $\geq t_a$, and have deadline $\leq t_d$. By definition of DBF and Lemma 1, we have

$$\mathcal{C} \leq \text{DBF}(\tau_k, t_d - t_a) \leq \delta_k \times (t_d - t_a) \ . \tag{3}$$

We introduce some notation now. For any time-instant $t \leq t_d$,

- let $W(t)$ denote the cumulative execution requirement of all the jobs in this legal sequence of job requests, *minus* the total amount of execution completed by the DM schedule prior to time-instant t.
- Let $\Omega(t)$ denote $W(t)$ normalized by the interval-length: $\Omega(t) \stackrel{\text{def}}{=} W(t)/(t_d - t)$.
- let I_ℓ denote the total duration over $[t_a, t_d)$ for which exactly ℓ processors are busy in this DM schedule, $0 \leq \ell \leq m$. (We note that I_o is necessarily zero, since τ_k's job does not complete by its deadline.)

Observe that the amount of execution that τ_k's jobs receive over $[t_a, t_d)$ is at least $\sum_{\ell=1}^{m-1} s_\ell I_\ell$, since τ_k's jobs must be executing at any time-instant when some processor is idle; therefore

$$\mathcal{C} > \sum_{\ell=1}^{m-1} s_\ell I_\ell \ , \tag{4}$$

Since $\left[S_m(\pi) \cdot \mathcal{D} - \sum_{\ell=1}^{m-1}(S_m(\pi) - S_\ell(\pi))I_\ell\right]$ denotes the total amount of execution completed over $[t_a, t_d)$ and this is not enough for τ_k's jobs to complete the execution requirement $\mathcal{C}$ before t_d, we have the following relationship:

[2] See Definition 1 to recall the definition of an active task.

$$\begin{aligned}W(t_a) &> S_m(\pi)\mathcal{D} - \sum_{\ell=1}^{m-1}(S_m(\pi) - S_\ell(\pi))I_\ell \\ &= S_m(\pi)\mathcal{D} - \sum_{\ell=1}^{m-1}\frac{S_m(\pi) - S_\ell(\pi)}{s_\ell}s_\ell I_\ell \\ &\geq S_m(\pi)\mathcal{D} - \sum_{\ell=1}^{m-1}\lambda(\pi)s_\ell I_\ell \\ &= S_m(\pi)\mathcal{D} - \lambda(\pi)\sum_{\ell=1}^{m-1}s_\ell I_\ell \ . \end{aligned} \tag{5}$$

From (5) and (4) above, we conclude that

$$\begin{aligned}W(t_a) &> S_m(\pi)\mathcal{D} - \lambda(\pi)\mathcal{C} \\ &\geq S_m(\pi)\mathcal{D} - \lambda(\pi)\delta_k\mathcal{D} \\ &\equiv \Omega(t_a) > S_m(\pi) - \lambda(\pi)\delta_k \\ &\Rightarrow \Omega(t_a) > S_m(\pi) - \lambda(\pi)\delta_{\max}(k) \ .\end{aligned}$$

Let

$$\mu_k \overset{\text{def}}{=} S_m(\pi) - \lambda(\pi)\delta_{\max}(k) \tag{6}$$

—observe that the value of μ_k depends upon the parameters of both the task system τ being scheduled, and the uniform multiprocessor $\pi = (s_1, s_2, \ldots, s_m)$ upon which it is scheduled.

Let t_o denote the smallest value of $t \leq t_a$ such that $\Omega(t) \geq \mu_k$. Let $\Delta \overset{\text{def}}{=} t_d - t_o$ (see Fig. 1).

By definition, $W(t_o)$ denotes the amount of work that the DM schedule needs (but fails) to execute over $[t_o, t_d)$. This work in $W(t_o)$ arises from two sources: those jobs that arrived at or after t_o, and those that arrived prior to t_o but have not completed execution in the DM schedule by time-instant t_o. We will refer to jobs arriving prior to t_o that need execution over $[t_o, t_d)$ as ***carry-in jobs.***

We wish to obtain an upper bound on the total contribution of all the carry-in jobs to the $W(t_o)$ term. We achieve this in two steps: we first bound the number of tasks that may have carry-in jobs (Lemma 2), and then we bound the amount of work that all the carry-in jobs of any one such task may contribute to $W(t_o)$ (Lemma 3).

Lemma 2. *The number of tasks that have carry-in jobs is strictly bounded from above by*

$$\nu_k \overset{\text{def}}{=} \max\{\ell \ : \ S_\ell(\pi) < \mu_k\} \ . \tag{7}$$

Proof. Let ϵ denote an arbitrarily small positive number. By definition of the instant t_o, $\Omega(t_o - \epsilon) < \mu_k$ while $\Omega(t_o) \geq \mu_k$. It must therefore be the case that strictly less than $\mu_k \times \epsilon$ work was executed over $[t_o - \epsilon, t_o)$; i.e., the total computing capacity of all the busy processors over $[t_o - \epsilon, t_o)$ is $< \mu_k$. And since

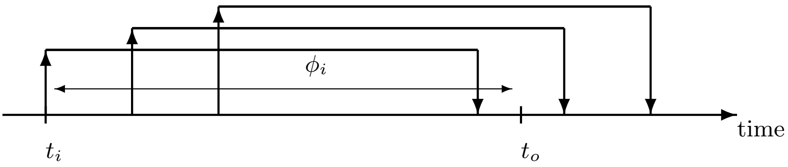


Fig. 2. Example: defining t_i for a task τ_i with $D_i \geq T_i$. Three jobs of τ_i are shown. Task τ_i is not active prior to the arrival of the first of these 3 jobs, and the first job completes execution only after the next job arrives. This second job does not complete execution prior to t_o. Thus, the task is continuously active after the arrival of the first job shown, and t_i is hence set equal to the arrival time of this job.

$\mu_k < S_m(\pi)$ (as can be seen from (6) above), it follows that some processor was idled over $[t_o - \epsilon, t_o)$, implying that all jobs active at this time would have been executing. This allows us to conclude that there are strictly fewer than ν_k tasks with carry-in jobs.

Lemma 3. *The total remaining execution requirement of all the carry-in jobs of each task τ_i (that has carry-in jobs at time-instant t_o) is $< \Delta \times \delta_{\max}(k)$.*

Proof. Let us consider some task τ_i $(i < k)$ that has a carry-in job. Let $t_i < t_o$ denote the earliest time-instant such that τ_i is active throughout the interval $[t_i, t_o]$. Observe that t_i is necessarily the arrival time of some job of τ_i. If $D_i < T_i$, then t_i is the arrival time of the (sole) carry-in job of τ_i. If $D_i \geq T_i$, however, t_i may be the arrival-time of a job that is not a carry-in job — see Fig. 2.

Let $\phi_i \stackrel{\text{def}}{=} t_o - t_i$ (see Fig. 2). All the carry-in jobs of τ_i have their arrival-times and their deadlines within the $(\phi_i + \Delta)$-sized interval $[t_i, t_d)$, and consequently their cumulative execution requirement is $\leq \text{DBF}(\tau_i, \phi_i + \Delta)$; in what follows, we will quantify how much of this must have been completed prior to t_o (and hence cannot contribute to the carry-in). We thus obtain an upper bound on the total work that all the carry-in jobs of τ_i contribute to $W(t_o)$, as the difference between $\text{DBF}(\tau_i, \phi_i + \Delta)$ and the amount of execution received by τ_i over $[t_i, t_o)$.

By definition of t_o, it must be the case that $\Omega(t_i) < \mu_k$. That is,

$$W(t_i) < \mu_k(\Delta + \phi_i) \ . \tag{8}$$

On the other hand, $\Omega(t_o) \geq \mu_k$, meaning that

$$W(t_o) \geq \mu_k \Delta \ . \tag{9}$$

Let C'_i denote the amount of execution received by τ_i's carry-in jobs over the duration $[t_i, t_o)$; the difference $\text{DBF}(\tau_i, \phi_i + \Delta) - C'_i$ thus denotes an upper bound on the amount of carry-in execution. Let J_ℓ denote the total duration over $[t_i, t_o)$ for which exactly ℓ processors are busy in this DM schedule, $0 \leq \ell \leq m$. Observe that the amount of execution that τ_i's carry-in jobs receive over $[t_i, t_o)$ is at least

$\sum_{\ell=1}^{m-1} s_\ell J_\ell$ since τ_i's job must be executing on one of the processors during any instant when some processor is idle; therefore

$$C_i' \geq \sum_{\ell=1}^{m-1} s_\ell J_\ell \ . \tag{10}$$

Since $\left[S_m(\pi)\phi_i - \sum_{\ell=1}^{m-1} J_\ell(S_m(\pi) - S_\ell(\pi))\right]$ denotes the total amount of execution completed over $[t_i, t_o)$, the difference $\big(W(t_i) - W(t_o)\big)$ — the amount of execution completed over $[t_i, t_o)$ — is given by

$$\begin{aligned}
W(t_i) - W(t_o) &= S_m(\pi)\phi_i - \sum_{\ell=1}^{m-1}(S_m(\pi) - S_\ell(\pi))J_\ell \\
\Rightarrow \mu_k(\Delta + \phi_i) - \mu_k\Delta &> \\
& S_m(\pi)\phi_i - \sum_{\ell=1}^{m-1}(S_m(\pi) - S_\ell(\pi))J_\ell \\
& \quad \text{(By (8) and (9))} \\
\equiv \mu_k\phi_i &> S_m(\pi)\phi_i - \sum_{\ell=1}^{m-1}\frac{S_m(\pi) - S_\ell(\pi)}{s_\ell}s_\ell J_\ell \\
\equiv \mu_k\phi_i &> S_m(\pi)\phi_i - \sum_{\ell=1}^{m-1}\lambda(\pi)s_\ell J_\ell \\
\equiv \mu_k\phi_i &> S_m(\pi)\phi_i - \lambda(\pi)\sum_{\ell=1}^{m-1}s_\ell J_\ell \\
\equiv \mu_k\phi_i &> S_m(\pi)\phi_i - \lambda(\pi)C_i' \ .
\end{aligned}$$

Substituting for μ_k (Equation 6 above), we have

$$(S_m(\pi) - \lambda(\pi)\delta_{\max}(k))\phi_i \ > \ S_m(\pi)\phi_i - \lambda(\pi)C_i' \ \equiv \ C_i' \ > \ \delta_{\max}(k)\phi_i \ . \tag{11}$$

Inequality 11 is important – it tells us that task τ_i must have already completed a significant amount of its execution *before* time-instant t_o. More specifically, the remaining work of all carry-in jobs of τ_i contribute to $W(t_o)$, is given by

$$\begin{aligned}
(\text{DBF}(\tau_i, \phi_i + \Delta) - C_i') &< (\phi_i + \Delta)\delta_i - \phi_i\delta_{\max}(k) \\
& \quad \text{(from Lemma 1)} \\
& \leq (\phi_i + \Delta)\delta_{\max}(k) - \phi_i\delta_{\max}(k)) = \Delta\delta_{\max}(k)
\end{aligned}$$

as claimed in this lemma.

Based upon Lemmas 2 and 3 we obtain our desired result —a sufficient schedulability condition for global DM:

Theorem 1. *Sporadic task system τ is global-*DM *schedulable upon a platform comprised of m uniform processors, provided that for all k, $1 \leq k \leq n$,*

$$2 \cdot \textsc{load}(k) + \nu_k \delta_{\max}(k) \leq \mu_k \ , \tag{12}$$

where μ_k and ν_k are as defined in (6) and 7 respectively.

Proof. The proof is by contradiction: we obtain necessary conditions for the scenario above — when τ_k's job misses its deadline at t_d – to occur. Negating these conditions yields a sufficient condition for global-DM schedulability.

Let us bound the total amount of execution that contributes to $W(t_o)$.

- First, there are the carry-in jobs: by Lemmas 3 and 2, there are at most ν_k distinct tasks with carry-in jobs, with the total carry-in work for all the jobs of each task bounded from above by $\Delta\ \delta_{\max}(k)$ units of work. Therefore their total contribution to $W(t_o)$ is bounded from above by $\nu_k \Delta\ \delta_{\max}(k)$.
- All other jobs that contribute to $W(t_o)$ arrive within the Δ-sized interval $[t_o, t_d)$, and hence have their deadlines within $[t_o, t_d + D_k)$, since their relative deadlines are all $\leq D_k$. Their total execution requirement is therefore bounded from above by $(\Delta + D_k) \times \textsc{load}(k)$.

We consequently obtain the following bound on $W(t_o)$:

$$W(t_o) < (\Delta + D_k) \times \textsc{load}(k) + \nu_k \Delta \delta_{\max}(k) \ . \tag{13}$$

Since, by the definition of t_o, it is required that $\Omega(t_o)$ be at least as large as μ_k, we must have

$$\left(1 + \frac{D_k}{\Delta}\right) \textsc{load}(k) + \nu_k \delta_{\max}(k) > \mu_k$$

as a necessary condition for DM to miss a deadline; equivalently, the negation of this condition is sufficient to ensure DM-schedulability:

$$\begin{aligned} &\left(1 + \frac{D_k}{\Delta}\right) \textsc{load}(k) + \nu_k \delta_{\max}(k) \leq \mu_k \\ &\Leftarrow \ (\text{since } D_k \leq \Delta) \\ &\quad 2 \cdot \textsc{load}(k) + \nu_k \delta_{\max}(k) \leq \mu_k \end{aligned}$$

which is as claimed in the theorem.

4 A Speedup Bound

In this section, we provide a quantitative evaluation of the effectiveness of the sufficient schedulability condition of Theorem 1. There are several approaches to quantifying the "goodness" or the effectiveness of different scheduling algorithms and schedulability tests. One relatively recent novel approach is centered on *processor speedup bounds*. A sufficient schedulability test is said to have a processor speedup bound of c if

- Any task system deemed schedulable by the test is guaranteed to actually be so; and
- For any task system that is not deemed schedulable by the test, it is the case that the task system is actually not schedulable upon a platform in which each processor is $\frac{1}{c}$ times as fast.

Intuitively speaking, a processor speedup bound of c for a sufficient schedulability test implies that the inexactness of the test penalizes its user by at most a speedup factor of c when compared to an exact test. The smaller the processor speedup bound, the better the sufficient schedulability test: a processor speedup bound of 1 would mean that the test is in fact an exact one.

We introduce some notation now. For any uniform multiprocessor platform $\pi = (s_1, s_2, \ldots, s_m)$ and any positive real number x, let $x \cdot \pi$ denote the uniform multiprocessor platform comprised of the same number of processors as π, with the i'th processor having a speed of $s_i \cdot x$.

The following two lemmas relate DM-schedulability on a uniform multiprocessor platform π, as validated by the test of Theorem 1, with feasibility on platform $x \cdot \pi$.

Lemma 4. *Any sporadic task system τ that is feasible upon a uniform multiprocessor platform $x \cdot \pi$ must satisfy*

$$\delta_{\max}(k) \leq s_1 x \quad \text{and} \quad \textsc{load}(k) \leq S_m(\pi) x \tag{14}$$

for all k, $1 \leq k \leq n$.

Proof. Suppose that task system τ is feasible upon $x \cdot \pi$. To prove that $\delta_{\max}(k) \leq xs_1$, consider each task τ_i separately:

- In order to be able to meet all deadlines of τ_i if τ_i generates jobs exactly T_i time units apart, it is necessary that $C_i/T_i \leq xs_1$.
- Since any individual job of τ_i can receive at most $D_i \times xs_1$ units of execution by its deadline, we must have $C_i \leq D_i \times x \times s_1$; i.e., $C_i/D_i \leq xs_1$.

Putting both conditions together, we get $(C_i/\min(T_i, D_i)) \leq xs_1$. Taken over all the tasks $\tau_1, \tau_2, \ldots, \tau_k$, this observation yields the condition that $\delta_{\max}(k) \leq xs_1$.

Since any individual job of τ_i can receive at most $D_i \times xs_1$ units of execution by its deadline, we must have $C_i \leq D_i \times s_1 x$; i.e., $C_i/D_i \leq s_1 x$. Taken over all tasks in τ, this observation yields the first condition.

To prove that $\textsc{load}(k) \leq S_m(\pi)x$, recall the definition of $\textsc{load}(k)$ from Sect. 1. Let t' denote some value of t which defines $\textsc{load}(k)$:

$$t' \overset{\text{def}}{=} \operatorname{argmax} \left(\frac{\sum_{i=1}^{k} \text{DBF}(\tau_i, t)}{t} \right) .$$

Suppose that all tasks in $\{\tau_1, \tau_2, \ldots, \tau_k\}$ generate a job at time-instant zero, and each task τ_i generates subsequent jobs exactly T_i time units apart. The total amount of execution that is available over the interval $[0, t')$ on this platform is equal to $S_m(\pi)xt'$; hence, it is necessary that $\textsc{load}(k) \leq S_m(\pi)x$ if all deadlines are to be met.

Lemma 5. *Any sporadic task system that is feasible upon a multiprocessor platform $x \cdot \pi$ is determined to be global-*DM *schedulable on π by the* DM*-schedulability test of Theorem 1, provided*

$$\begin{aligned} x \le (2\lambda s_1) / \Big[& S_m(\pi)s_1 + 2S_m(\pi)s_m + \lambda s_1 s_m - \big(S_m^2(\pi)s_1^2 + 4S_m^2(\pi)s_1 s_m \\ & + 2S_m(\pi)s_1^2\lambda s_m + 4S_m^2(\pi)s_m^2 + 4\lambda s_1 s_m^2 S_m(\pi) + \lambda^2 s_1^2 s_m^2 \\ & - 4\lambda s_1 S_m(\pi)s_m\big)^{1/2}\Big] \ . \end{aligned} \tag{15}$$

Proof. Suppose that τ is feasible upon a platform $\pi \cdot x$. From Lemma 4, it must be the case that $\text{LOAD}(k) \le S_m(\pi)x$ and $\delta_{\max}(k) \le s_1 x$ for all k. For τ to be determined to be DM-schedulable upon π by the test of Theorem 1, it is sufficient that for all k, $1 \le k \le n$:

$$\begin{aligned} \text{LOAD}(k) &\le \frac{1}{2}(\mu_k - \nu_k \delta_{\max}(k)) \\ &\Leftarrow \text{(since } (\lceil \frac{\mu_k}{s_m} \rceil - 1) \ge \nu_k \text{ and from Lemma 2)} \\ \text{LOAD}(k) &\le \frac{1}{2}(\mu_k - (\lceil \frac{\mu_k}{s_m} \rceil - 1)\delta_{\max}(k)) \\ &\Leftarrow \text{(since } \lceil \alpha \rceil - 1 \le \alpha \text{ for all } \alpha) \\ \text{LOAD}(k) &\le \frac{1}{2}(\mu_k - (\frac{\mu_k}{s_m})\delta_{\max}(k)) \\ &\equiv \\ \text{LOAD}(k) &\le \frac{1}{2}(\mu_k(1 - \frac{\delta_{\max}(k)}{s_m})) \\ &\equiv \text{(by (6))} \\ \text{LOAD}(k) &\le \frac{1}{2}((S_m(\pi) - \lambda\delta_{\max}(k))(1 - \frac{\delta_{\max}(k)}{s_m})) \\ &\Leftarrow \\ S_m(\pi)x &\le \frac{1}{2}((S_m(\pi) - \lambda s_1 x)(1 - \frac{s_1 x}{s_m})) \\ &\equiv \\ S_m(\pi)x &\le \frac{1}{2}(S_m(\pi) - \frac{S_m(\pi)s_1 x}{s_m} - \lambda s_1 x + \frac{\lambda s_1^2 x^2}{s_m}) \\ &\equiv \\ 0 \le & \lambda s_1 x^2 - [S_m(\pi)(s_1 + 2s_m) + \lambda s_1 s_m]x \\ & + S_m(\pi)s_m \ . \end{aligned}$$

Solving for x using standard techniques for the solution of quadratic inequalities yields (15).

A processor-speedup bound for the DM-schedulability test of Theorem 1 immediately follows from Lemma 5:

Table 1. Speedup bound for various uniform platforms

heterogeneity	m	$S_m(\pi)$	λ	s_m	s_1	speedup
$H_{1/2}$	4	12	2	4	2	4.59
$H_{1/2}$	20	60	14	4	2	4.84
$H_{1/2}$	100	300	74	4	2	4.89
$H_{1/2}$	1000	3000	749	4	2	4.90
$H_{1/4}$	4	15	0.875	8	1	10.42
$H_{1/4}$	20	75	8.345	8	1	10.81
$H_{1/4}$	100	375	45.875	8	1	10.89
$H_{1/4}$	1000	3750	467.75	8	1	10.91

Theorem 2. *The* DM*-schedulability test of Theorem 1 has a processor speedup bound of the value of the right side of (15).* □

4.1 Analysis of the Speedup Bound

We first observe that the processor speedup bound of Theorem 2 generalizes previously-obtained bounds for *identical* multiprocessors. It may be verified that by setting $\lambda = (m-1), s_1 = s_2 = \cdots = s_m = 1$, and $S_m(\pi) = m$, Theorem 1 reduces to the DM-schedulability test for identical multiprocessors, *and* Theorem 2 reduces[3] to the speedup bound for identical multiprocessors, derived in [25]. It consequently follows that our result here is a generalization of the identical multiprocessor DM test and speedup bound from [25].

Evaluation by simulation experiments. Equation (15) expresses the processor speedup bound as a function of the following platform parameters: λ, s_1, s_m, and $S_m(\pi)$. In order to get a more intuitive feel for the bounds, we computed the speedup bound for various uniform multiprocessor platforms. Due to the large number of parameters we restricted our study to platforms with four distinct processor speeds: $8, 4, 2, 1$ and two kinds of heterogeneity: 25 % of each kind of processor speed, 50 % of processor speed 4 plus 50 % of processor speed 2 (labeled $H_{1/4}$ and $H_{1/2}$ in Table 1, respectively).

Table 1 gives the speedup bound for the various uniform platforms considered in this study. As seen from this table, the bound increases with increasing heterogeneity, and increases with increasing size of the platform (for a given heterogeneity).

5 Conclusions

Most research on multiprocessor real-time scheduling has focused on the simplest model — systems of *implicit-deadline tasks* that are scheduled on *identical*

[3] Notice that the discriminant presented in [25] is actually $12m^2 - \mathbf{4}m + 1$.

multiprocessors. More recent research has attempted to generalize this work in two different directions, by either generalizing the task model (to constrained-deadline and arbitrary-deadline sporadic task systems), or by generalizing the processor model (to uniform and unrelated multiprocessors).

Very recently [28], efforts have been made to generalize along both the task-model and the processor axes, by considering the scheduling of sporadic task systems upon uniform multiprocessors. However, the only scheduling algorithm that was considered in [28] is Earliest Deadline First (EDF). In this work, we have applied the techniques from [28] to DM scheduling. We have obtained a new schedulability test for the global DM scheduling of sporadic task systems upon preemptive uniform multiprocessor platforms. This test characterizes a task system by its LOAD and $\delta_{\max}$ parameters, and a platform by its total computing capacity and its λ parameter. We have also obtained a characterization of the effectiveness of this schedulability test in terms of its processor speedup factor.

References

1. Mok, A.K.: Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Available as Technical Report No. MIT/LCS/TR-297 (1983)
2. Baruah, S., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th Real-Time Systems Symposium, Orlando, Florida, pp. 182–190. IEEE Computer Society Press, Los Alamitos (1990)
3. Baruah, S., Cohen, N., Plaxton, G., Varvel, D.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15(6), 600–625 (1996)
4. Oh, D.I., Baker, T.P.: Utilization bounds for N-processor rate monotone scheduling with static processor assignment. Real-Time Systems: The International Journal of Time-Critical Computing 15, 183–192 (1998)
5. Lopez, J.M., Garcia, M., Diaz, J.L., Garcia, D.F.: Worst-case utilization bound for EDF scheduling in real-time multiprocessor systems. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Stockholm, Sweden, pp. 25–34. IEEE Computer Society Press, Los Alamitos (2000)
6. Andersson, B., Jonsson, J.: Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In: Proceedings of the International Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea, pp. 337–346. IEEE Computer Society Press, Los Alamitos (2000)
7. Andersson, B., Baruah, S., Jonsson, J.: Static-priority scheduling on multiprocessors. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 193–202. IEEE Computer Society Press, Los Alamitos (2001)
8. Goossens, J., Funk, S., Baruah, S.: Priority-driven scheduling of periodic task systems on multiprocessors. Real Time Systems 25(2–3), 187–205 (2003)
9. Lopez, J.M., Diaz, J.L., Garcia, D.F.: Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Systems: The International Journal of Time-Critical Computing 28(1), 39–68 (2004)
10. Funk, S.H.: EDF Scheduling on Heterogeneous Multiprocessors. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill (2004)

11. Baruah, S.: Scheduling periodic tasks on uniform processors. In: Proceedings of the EuroMicro Conference on Real-time Systems, Stockholm, Sweden, pp. 7–14 (June 2000)
12. Funk, S., Goossens, J., Baruah, S.: On-line scheduling on uniform multiprocessors. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 183–192. IEEE Computer Society Press, Los Alamitos (2001)
13. Baruah, S., Goossens, J.: Rate-monotonic scheduling on uniform multiprocessors. IEEE Transactions on Computers 52(7), 966–970 (2003)
14. Funk, S., Baruah, S.: Task assignment on uniform heterogeneous multiprocessors. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Palma de Mallorca, Balearic Islands, Spain, pp. 219–226. IEEE Computer Society Press, Los Alamitos (2005)
15. Darera, V.N., Jenkins, L.: Utilization bounds for RM scheduling on uniform multiprocessors. In: RTCSA 2006: Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Washington, DC, USA, pp. 315–321. IEEE Computer Society, Los Alamitos (2006)
16. Andersson, B., Tovar, E.: Competitive analysis of partitioned scheduling on uniform multiprocessors. In: Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, Long Beach, CA (March 2007)
17. Andersson, B., Tovar, E.: Competitive analysis of static-priority scheduling on uniform multiprocessors. In: Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Daegu, Korea. IEEE Computer Society Press, Los Alamitos (2007)
18. Baker, T.: Multiprocessor EDF and deadline monotonic schedulability analysis. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 120–129. IEEE Computer Society Press, Los Alamitos (2003)
19. Baker, T.P.: An analysis of EDF schedulability on a multiprocessor. IEEE Transactions on Parallel and Distributed Systems 16(8), 760–768 (2005)
20. Bertogna, M., Cirinei, M., Lipari, G.: Improved schedulability analysis of EDF on multiprocessor platforms. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Palma de Mallorca, Balearic Islands, Spain, pp. 209–218. IEEE Computer Society Press, Los Alamitos (2005)
21. Bertogna, M., Cirinei, M., Lipari, G.: New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors. In: Proceedings of the 9th International Conference on Principles of Distributed Systems, Pisa, Italy. IEEE Computer Society Press, Los Alamitos (2005)
22. Cirinei, M., Baker, T.P.: EDZL scheduling analysis. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Pisa, Italy. IEEE Computer Society Press, Los Alamitos (2007)
23. Fisher, N.: The Multiprocessor Real-Time Scheduling of General Task Systems. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill (2007)
24. Baruah, S., Baker, T.: Schedulability analysis of global EDF. Real- Time Systems (to appear, 2008)
25. Baruah, S., Fisher, N.: Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 204–216. Springer, Heidelberg (2007)
26. Baruah, S., Baker, T.: Global EDF schedulability analysis of arbitrary sporadic task systems. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Prague, Czech Republic. IEEE Computer Society Press, Los Alamitos (2008)

27. Leung, J., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Performance Evaluation 2, 237–250 (1982)
28. Baruah, S., Goossens, J.: The EDF scheduling of sporadic task systems on uniform multiprocessors. Technical report, University of North Carolina at Chapel Hill (2008)
29. Ripoll, I., Crespo, A., Mok, A.K.: Improvement in feasibility testing for real-time tasks. Real-Time Systems: The International Journal of Time-Critical Computing 11, 19–39 (1996)
30. Baker, T.P., Fisher, N., Baruah, S.: Algorithms for determining the load of a sporadic task system. Technical Report TR-051201, Department of Computer Science, Florida State University (2005)
31. Fisher, N., Baruah, S., Baker, T.: The partitioned scheduling of sporadic tasks according to static priorities. In: Proceedings of the EuroMicro Conference on Real-Time Systems, Dresden, Germany. IEEE Computer Society Press, Los Alamitos (2006)
32. Fisher, N., Baker, T., Baruah, S.: Algorithms for determining the demand-based load of a sporadic task system. In: Proceedings of the International Conference on Real-time Computing Systems and Applications, Sydney, Australia. IEEE Computer Society Press, Los Alamitos (2006)
33. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM 20(1), 46–61 (1973)

A Comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\text{RT}}$

Björn B. Brandenburg and James H. Anderson

The University of North Carolina at Chapel Hill
Dept. of Computer Science
Chapel Hill, NC 27599-3175 USA
{bbb,anderson}@cs.unc.edu

Abstract. This paper presents a performance comparison of three multiprocessor real-time locking protocols: the multiprocessor priority ceiling protocol (M-PCP), the distributed priority ceiling protocol (D-PCP), and the flexible multiprocessor locking protocol (FMLP). In the FMLP, blocking is implemented via either suspending or spinning, while in the M-PCP and D-PCP, all blocking is by suspending. The presented comparison was conducted using a UNC-produced Linux extension called LITMUS$^{\text{RT}}$. In this comparison, schedulability experiments were conducted in which runtime overheads as measured on LITMUS$^{\text{RT}}$ were used. In these experiments, the spin-based FMLP variant *always* exhibited the best performance, and the M-PCP and D-PCP almost always exhibited poor performance. These results call into question the practical viability of the M-PCP and D-PCP, which have been the de-facto standard for real-time multiprocessor locking for the last 20 years.

1 Introduction

With the continued push towards multicore architectures by most (if not all) major chip manufacturers [19,26], the computing industry is facing a paradigm shift: in the near future, multiprocessors will be the norm. Current off-the-shelf systems now routinely contain chips with two, four, and even eight cores, and chips with up to 80 cores are envisioned within a decade [26]. Not surprisingly, with multicore platforms becoming so widespread, real-time applications are already being deployed on them. For example, systems processing time-sensitive business transactions have been realized by Azul Systems on top of the highly-parallel Vega2 platform, which consists of up to 768 cores [4].

Motivated by these developments, research on multiprocessor real-time scheduling has intensified in recent years (see [13] for a survey). Thus far, however, few proposed approaches have actually been implemented in operating systems and evaluated under real-world conditions. To help bridge the gap between algorithmic research and real-world systems, our group recently developed LITMUS$^{\text{RT}}$, a multiprocessor real-time extension of Linux [8,11,12]. Our choice of Linux as a development platform was influenced by recent efforts to introduce real-time-oriented features in stock Linux (see, for example, [1]). As Linux evolves, it could

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 105–124, 2008.

undoubtedly benefit from recent algorithmic advances in real-time scheduling-related research.

LITMUS$^{\mathrm{RT}}$ has been used in several scheduling-related performance studies [5,8,12]. In addition, a study was conducted to compare synchronization alternatives under global and partitioned earliest-deadline-first (EDF) scheduling [11]. This study was partially motivated by the relative lack (compared to scheduling) of research on real-time multiprocessor synchronization. It focused more broadly on comparing suspension- and spin-based locking on the basis of schedulability. Spin-based locking was shown to be the better choice.

Focus of this paper. In this paper, we present follow-up work to the latter study that focuses on systems where *partitioned, static-priority* (P-SP) scheduling is used. This is an important category of systems, as both partitioning and static priorities tend to be favored by practitioners. Moreover, the earliest and most influential work on multiprocessor real-time synchronization was directed at such systems. This work resulted in two now-classic locking protocols: the *multiprocessor priority ceiling protocol* (M-PCP) and the *distributed priority ceiling protocol* (D-PCP) [23]. While these protocols are probably the most widely known (and taught) locking protocols for multiprocessor real-time applications, they were developed at a time (over 20 years ago) when such applications were deemed to be mostly of "academic" interest only. With the advent of multicore technologies, this is clearly no longer the case. Motivated by this, we take a new look at these protocols herein with the goal of assessing their practical viability. We also examine the subject of our prior EDF-based study, the *flexible multiprocessor locking protocol* (FMLP) [6,9,11]. We seek to assess the effectiveness of these protocols in managing memory-resident resources on P-SP-scheduled shared-memory multiprocessors.

Tested protocols. The M-PCP, D-PCP, and FMLP function very differently. In both the M-PCP and D-PCP, blocked tasks are suspended, *i.e.*, such a task relinquishes its assigned processor. The main difference between these two protocols is that, in the D-PCP, resources are assigned to processors, and in the M-PCP, such an assignment is not made.[1] In the D-PCP, a task accesses a resource via an RPC-like invocation of an agent on the resource's processor that performs the access. In the M-PCP, a global semaphore protocol is used instead. In both protocols, requests for global resources (*i.e.*, resources accessed by tasks on multiple processors) cannot appear in nested request sequences. Invocations on such resources are ordered by priority and execute at elevated priority levels so that they complete more quickly. In contrast to these two protocols, the FMLP orders requests on a FIFO basis, allows arbitrary request nesting (with one slight restriction, described later), and is agnostic regarding whether blocking is via spinning (busy-waiting) or suspension. While spinning wastes processor time, in our prior work on EDF-scheduled systems [11], we found that its use almost always results in better schedulability than suspending. This is because

[1] Because the D-PCP assigns resources to processors, it can potentially be used in loosely-coupled distributed systems—hence its name.

it can be difficult to predict which scheduler events may affect a task while it is suspended, so needed analysis tends to be pessimistic. (Each of the protocols considered here is described more fully later.)

Methodology and results. The main contribution of this paper is an assessment of the performance of the three protocols described above in terms of P-SP schedulability. Our methodology in conducting this assessment is similar to that used in our earlier work on EDF-scheduled systems [11]. The performance of any synchronization protocol will depend on runtime overheads, such as preemption costs, scheduling costs, and costs associated with performing various system calls. We determined these costs by analyzing trace data collected while running various workloads under LITMUS$^{\mathrm{RT}}$ (which, of course, first required implementing each synchronization protocol in LITMUS$^{\mathrm{RT}}$). We then used these costs in schedulability experiments involving randomly-generated task systems. In these experiments, a wide range of task-set parameters was considered (though only a subset of our data is presented herein, due to space limitations). In each experiment, schedulability was checked for each scheme using a demand-based schedulability test [15], augmented to account for runtime overheads. In these experiments, we found that the spin-based FMLP variant *always* exhibited the best performance (usually, by a wide margin), and the M-PCP and D-PCP almost always exhibited poor performance. These results reinforce our earlier finding that spin-based locking is preferable to suspension-based locking under EDF scheduling [11]. They also call into question the practical viability of the M-PCP and D-PCP.

Organization. In the next two sections, we discuss needed background and the results of our experiments. In an appendix, we describe how runtime overheads were obtained.

2 Background

We consider the scheduling of a system of *sporadic tasks*, denoted $T_1, \ldots, T_N$, on m processors. The j^{th} job (or invocation) of task T_i is denoted T_i^j. Such a job T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$. Each task T_i is specified by its *worst-case* (*per-job*) *execution cost*, $\mathsf{e}(T_i)$, and its *period*, $\mathsf{p}(T_i)$. The job T_i^j should complete execution by its *absolute deadline*, $\mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. The spacing between job releases must satisfy $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. Task T_i's *utilization* reflects the processor share that it requires and is given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$.

In this paper, we consider only *partitioned static-priority* (P-SP) scheduling, wherein each task is statically assigned to a processor and each processor is scheduled independently using a static-priority uniprocessor algorithm. A well-known example of such an algorithm is the *rate-monotonic* (RM) algorithm, which gives higher priority to tasks with smaller periods. In general, we assume that tasks are indexed from 1 to n by decreasing priority, *i.e.*, a lower index implies higher priority. We refer to T_i's index i as its *base priority.* A job is scheduled using its *effective priority*, which can sometimes exceed its base priority under certain resource-sharing policies (*e.g.*, priority inheritance may raise a job's effective priority). After its release, a job T_i^j is said to be *pending* until it completes. While it is pending, T_i^j is either *runnable* or *suspended.* A suspended job cannot be scheduled. When a job transitions from suspended to runnable (runnable to suspended), it is said to *resume* (*suspend*). While runnable, a job is either *preemptable* or *non-preemptable.* A newly-released or resuming job can preempt a scheduled lower-priority job only if it is preemptable.

Resources. When a job T_i^j requires a *resource* ℓ, it *issues* a *request* $\mathcal{R}_\ell$ for ℓ. $\mathcal{R}_\ell$ is *satisfied* as soon as T_i^j *holds* ℓ, and *completes* when T_i^j *releases* ℓ. $|\mathcal{R}_\ell|$ denotes the maximum time that T_i^j will hold ℓ. T_i^j becomes *blocked* on ℓ if $\mathcal{R}_\ell$ cannot be satisfied immediately. (A resource can be held by at most one job at a time.) A resource ℓ is *local* to a processor p if all jobs requesting ℓ execute on p, and *global* otherwise.

If T_i^j issues another request $\mathcal{R}'$ before $\mathcal{R}$ is complete, then $\mathcal{R}'$ is *nested* within $\mathcal{R}$. In such cases, $|\mathcal{R}|$ includes the cost of blocking due to requests nested in $\mathcal{R}$. Some synchronization protocols disallow nesting. If allowed, nesting is proper, *i.e.*, $\mathcal{R}'$ must complete no later than $\mathcal{R}$ completes. An *outermost* request is not nested within any other request. Inset (b) of Fig. 1 illustrates the different phases of a resource request. In this and later figures, the legend shown in inset (a) of Fig. 1 is assumed.

Resource sharing introduces a number of problems that can endanger temporal correctness. *Priority inversion* occurs when a high-priority job T_h^i cannot proceed due to a lower-priority job T_l^j either being non-preemptable or holding a resource requested by T_h^i. T_h^i is said to be *blocked by* T_l^j. Another source of delay is *remote blocking*, which occurs when a global resource requested by a job is already in use on another processor.

In each of the synchronization protocols considered in this paper, local resources can be managed by using simpler uniprocessor locking protocols, such as the priority ceiling protocol [25] or stack resource policy [3]. Due to space constraints, we do not consider such functionality further, but instead focus our attention on global resources, as they are more difficult to support and have the greatest impact on performance. We explain below how such resources are handled by considering each of the D-PCP, M-PCP, and FMLP in turn. It is not possible to delve into every detail of each protocol given the space available. For such details, we refer the reader to [6,9,11,21].

The D-PCP and M-PCP. The D-PCP implements global resources by providing *local agents* that act on behalf of requesting jobs. A local agent A_i^q, located on remote processor q where jobs of T_i request resources, carries out requests on behalf of T_i on processor q. Instead of accessing a global remote resource ℓ on processor q directly, a job T_i^j submits a request $\mathcal{R}$ to A_i^q and suspends. T_i^j resumes when A_i^q has completed $\mathcal{R}$. To expedite requests, A_i^q executes with an effective priority higher than that of any normal task (see [16,21] for details). However, agents of lower-priority tasks can still be preempted by agents of higher-priority

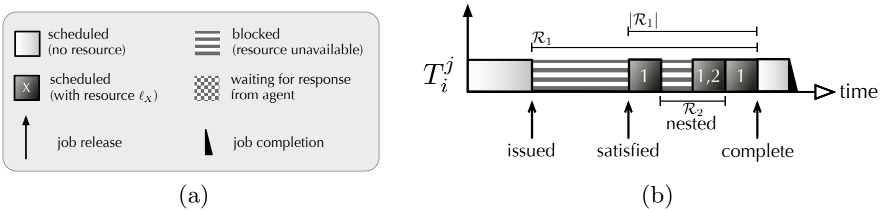


Fig. 1. (a) Legend. **(b)** Phases of a resource request. T_i^j issues $\mathcal{R}_1$ and blocks since $\mathcal{R}_1$ is not immediately satisfied. T_i^j holds the resource associated with $\mathcal{R}_1$ for $|\mathcal{R}_1|$ time units, which includes blocking incurred due to nested requests.

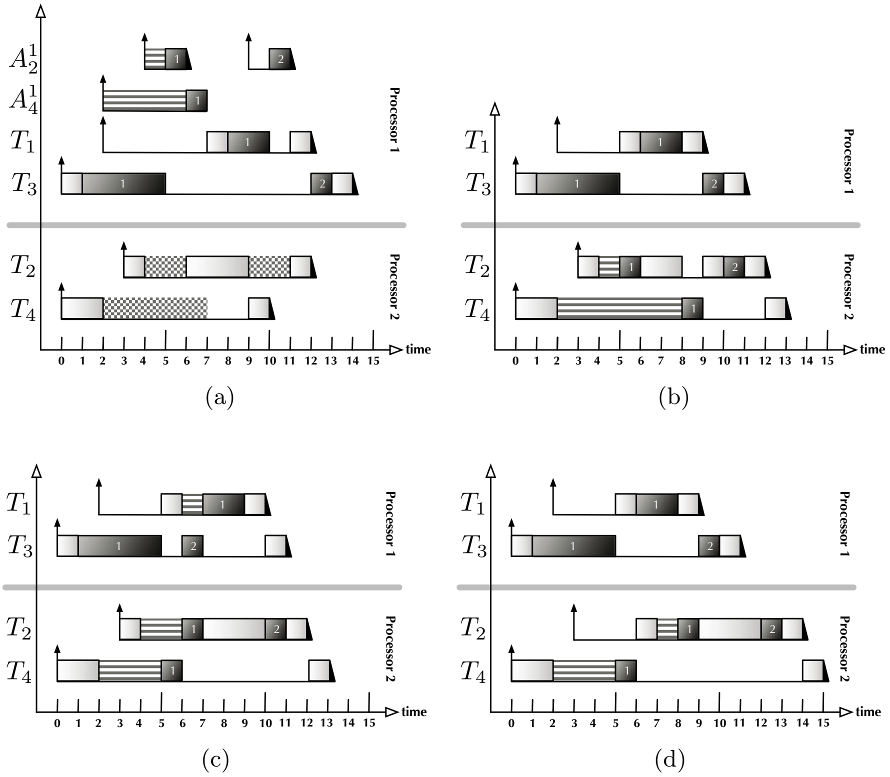
tasks. When accessing global resources residing on T_i's assigned processor, T_i^j serves as its own agent.

The M-PCP relies on shared memory to support global resources. In contrast to the D-PCP, global resources are not assigned to any particular processor but are accessed directly. Local agents are thus not required since jobs execute requests themselves on their assigned processors. Competing requests are satisfied in order of job priority. When a request is not satisfied immediately, the requesting job suspends until its request is satisfied. Under the M-PCP, jobs holding global resources execute with an effective priority higher than that of any normal task.

The D-PCP and M-PCP avoid deadlock by *prohibiting the nesting of global resource requests*—a global request $\mathcal{R}$ cannot be nested within another request (local or global) and no other request (local or global) may be nested within $\mathcal{R}$.

Example. Fig. 2 depicts global schedules for four jobs (T_1^1,...,T_4^1) sharing two resources (ℓ_1, ℓ_2) on two processors. Inset (a) shows resource sharing under the D-PCP. Both resources reside on processor 1. Thus, two agents (A_2^1, A_4^1) are also assigned to processor 1 in order to act on behalf of T_2 and T_4 on processor 2. A_4^1 becomes active at time 2 when T_4^1 requests ℓ_1. However, since T_3^1 already holds ℓ_1, A_4^1 is blocked. Similarly, A_2^1 becomes active and blocks at time 4. When T_3^1 releases ℓ_1, A_2^1 gains access next because it is the highest-priority active agent on processor 1. Note that, even though the highest-priority job T_1^1 is released at time 2, it is not scheduled until time 7 because agents and resource-holding jobs have an effective priority that exceeds the base priority of T_1^1. A_2^1 becomes active at time 9 since T_2^1 requests ℓ_2. However, T_1^1 is accessing ℓ_1 at the time, and thus has an effective priority that exceeds A_2^1's priority. Therefore, A_2^1 is not scheduled until time 10.

Inset (b) shows the same scenario under the M-PCP. In this case, T_2^1 and T_4^1 access global resources directly instead of via agents. T_4^1 suspends at time 2 since T_2^1 already holds ℓ_1. Similarly, T_2^1 suspends at time 4 until it holds ℓ_1 one time unit later. Meanwhile, on processor 1, T_1^1 is scheduled at time 5 after T_2^1 returns to normal priority and also requests ℓ_1 at time 6. Since resource requests are satisfied in priority order, T_1^1's request has precedence over T_4^1's request, which was issued much earlier at time 2. Thus, T_4^1 must wait until time 8 to access ℓ_1.



(a)

(b)

(c)

(d)

Fig. 2. Example schedules of four tasks sharing two global resources. **(a)** D-PCP schedule. **(b)** M-PCP schedule. **(c)** FMLP schedule (ℓ_1, ℓ_2 are long). **(d)** FMLP schedule (ℓ_1, ℓ_2 are short).

Note that T_4^1 preempts T_2^1 when it resumes at time 8 since it is holding a global resource.

The FMLP. The FMLP is considered to be "flexible" for several reasons: it can be used under either partitioned or global scheduling, with either static or dynamic task priorities, and it is agnostic regarding whether blocking is via spinning or suspension. Regarding the latter, resources are categorized as either "short" or "long." Short resources are accessed using queue locks (a type of spin lock) [2,14,18] and long resources are accessed via a semaphore protocol. Whether a resource should be considered short or long is user-defined, but requests for long resources may not be contained within requests for short resources. To date, we have implemented FMLP variants for both partitioned and global EDF and P-SP scheduling (the focus of the description given here).

Deadlock avoidance. The FMLP uses a *very* simple deadlock-avoidance mechanism that was motivated by trace data we collected involving the behavior of actual real-time applications [7]. This data (which is summarized later) suggests that nesting, which is required to cause a deadlock, is somewhat rare; thus, complex deadlock-avoidance mechanisms are of questionable utility. In the FMLP, deadlock is prevented by "grouping" resources and allowing only one job to access resources in any given group at any time. Two resources are in the same group iff they are of the same type (short or long) and requests for one may be nested within those of the other. A *group lock* is associated with each resource group; before a job can access a resource, it must first acquire its corresponding group lock. All blocking incurred by a job occurs when it attempts to acquire the group lock associated with a resource request that is outermost with respect to either short or long resources.[2] We let $\mathsf{G}(\ell)$ denote the group that contains resource ℓ.

We now explain how resource requests are handled in the FMLP. This process is illustrated in Fig. 3.

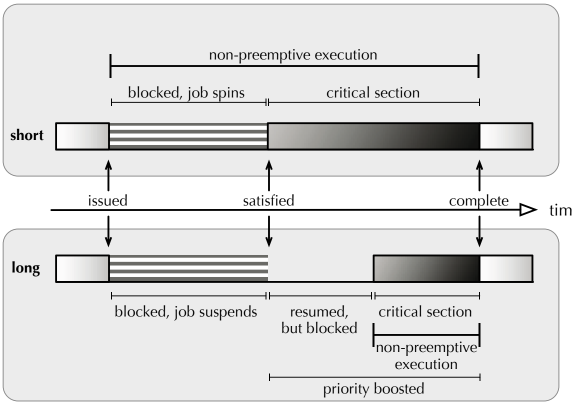


Fig. 3. Phases of short and long resource requests

Short requests. If $\mathcal{R}$ is short and outermost, then T_i^j becomes non-preemptable and attempts to acquire the queue lock protecting $\mathsf{G}(\ell)$. In a queue lock, blocked processes busy-wait in FIFO order.[3] $\mathcal{R}$ is satisfied once T_i^j holds ℓ's group lock. When $\mathcal{R}$ completes, T_i^j releases the group lock and leaves its non-preemptive section.

Long requests. If $\mathcal{R}$ is long and outermost, then T_i^j attempts to acquire the semaphore protecting $\mathsf{G}(\ell)$. Under a semaphore lock, blocked jobs are added to a FIFO queue and suspend. As soon as $\mathcal{R}$ is satisfied (*i.e.*, T_i^j holds ℓ's group lock), T_i^j resumes (if it suspended) and enters a non-preemptive section (which

[2] A short resource request nested within a long resource request but no short resource request is considered outermost.

[3] The desirability of FIFO-based real-time multiprocessor locking protocols has been noted by others [17], but to our knowledge, the FMLP is the first such protocol to be implemented in a real OS.

becomes effective as soon as T_i^j is scheduled). When $\mathcal{R}$ completes, T_i^j releases the group lock and becomes preemptive.

Priority boost. If $\mathcal{R}$ is long and outermost, then T_i^j's priority is boosted when $\mathcal{R}$ is satisfied (*i.e.*, T_i^j is scheduled with effective priority 0). This allows it to preempt jobs executing preemptively at base priority. If two or more priority-boosted jobs are ready, then they are scheduled in the order in which their priorities were boosted (FIFO).

Example. Insets (c) and (d) of Fig. 2 depict FMLP schedules for the same scenario previously considered in the context of the D-PCP and M-PCP. In (c), ℓ_1 and ℓ_2 are classified as long resources. As before, T_3^1 requests ℓ_1 first and forces the jobs on processor 2 to suspend (T_4^1 at time 2 and T_2^1 at time 4). In contrast to both the D-PCP and M-PCP, contending requests are satisfied in FIFO order. Thus, when T_3^1 releases ℓ_1 at time 5, T_4^1's request is satisfied before that of T_2^1. Similarly, T_1^1's request for ℓ_1 is only satisfied after T_2^1 completes its request at time 7. Note that, since jobs suspend when blocked on a long resource, T_3^1 can be scheduled for one time unit at time 6 when T_1^1 blocks on ℓ_1.

Inset (d) depicts the schedule that results when both ℓ_1 and ℓ_2 are short. The main difference from the schedule depicted in (c) is that jobs busy-wait non-preemptively when blocked on a short resource. Thus, when T_2^1 is released at time 3, it cannot be scheduled until time 6 since T_4^1 executes non-preemptively from time 2 until time 6. Similarly, T_4^1 cannot be scheduled at time 7 when T_2^1 blocks on ℓ_2 because T_2^1 does not suspend. Note that, due to the waste of processing time caused by busy-waiting, the last job only finishes at time 15. Under suspension-based synchronization methods, the last job finishes at either time 13 (M-PCP and FMLP for long resources) or 14 (D-PCP).

3 Experiments

In our study, we sought to assess the practical viability of the aforementioned synchronization protocols. To do so, we determined the schedulability of randomly-generated task sets under each scheme. (A task system is *schedulable* if it can be verified via some test that no task will ever miss a deadline.)

Task parameters were generated—similar to the approach previously used in [11]—as follows. Task utilizations were distributed uniformly over $[0.001, 0.1]$. To cover a wide range of timing constraints, we considered four ranges of periods: **(i)** $[3ms\text{-}33ms]$, **(ii)** $[10ms\text{-}100ms]$, **(iii)** $[33ms\text{-}100ms]$, and **(iv)** $[100ms\text{-}1000ms]$. Task execution costs (excluding resource requests) were calculated based on utilizations and periods. Periods were defined to be integral, but execution costs may be non-integral. All time-related values used in our experiments were defined assuming a target platform like that used in obtaining overhead values. As explained in the appendix, this system has four 2.7 GHz processors.

Given our focus on partitioned scheduling, task sets were obtained by first generating tasks for each processor individually, until either a per-processor *utilization cap* $\hat{U}$ was reached or 30 tasks were generated, and then generating resource requests. By eliminating the need to partition task sets, we prevent the effects of bin-packing heuristics from skewing our results. All generated task sets were determined to be schedulable before blocking was taken into account.

Resource sharing. Each task was configured to issue between 0 and K resource requests. The access cost of each request (excluding synchronization overheads) was chosen uniformly from $[0.1\mu s, L]$. K ranged from 0 to 9 and L from $0.5\mu s$ to $15.5\mu s$. The latter range was chosen based on locking trends observed in a prior study of locking patterns in the Linux kernel, two video players, and an interactive 3D video game (see [7] for details.). Although Linux is not a real-time system, its locking behavior should be similar to that of many complex systems, including real-time systems, where great care is taken to make critical sections short and efficient. The video players and the video game need to ensure that both visual and audio content are presented to the user in a timely manner, and thus are representative of the locking behavior of a class of soft real-time applications. The trace data we collected in analyzing these applications suggests that, with respect to both semaphores and spin locks, critical sections tend to be short (usually, just a few microseconds on a modern processor) and nested lock requests are somewhat rare (typically only 1% to 30% of all requests, depending on the application, with nesting levels deeper than two being very rare).

The total number of generated tasks N was used to determine the number of resources according to the formula $\frac{K \cdot N}{\alpha \cdot m}$, where the *sharing degree* α was chosen from $\{0.5, 1, 2, 4\}$. Under the D-PCP, resources were assigned to processors in a round-robin manner to distribute the load evenly. Nested resource requests were not considered since they are not supported by the M-PCP and D-PCP and also because allowing nesting has a similar effect on schedulability under the FMLP as increasing the maximum critical section length.

Finally, task execution costs and request durations were inflated to account for system overheads (such as context switching costs) and synchronization overheads (such as the cost of invoking synchronization-related system calls). The methodology for doing this is explained in the appendix.

Schedulability. After a task set was generated, the worst-case blocking delay of each task was determined by using methods described in [20,21] (M-PCP), [16,21] (D-PCP), and [9] (FMLP). Finally, we determined whether a task set was schedulable after accounting for overheads and blocking delay with a demand-based [15] schedulability test.

A note on the "period enforcer." When a job suspends, it defers part of its execution to a later instant, which can cause a lower-priority job to experience *deferral blocking*. In checking schedulability, this source of blocking must be accounted for. In [21], it is claimed that deferral blocking can be eliminated by using a technique called *period enforcer*. In this paper, we do not consider the use of the period enforcer, for a number of reasons. First, the period enforcer has not been described in published work (nor is a complete description available online). Thus, we were unable to verify its correctness[4] and were unable to obtain sufficient information to enable an implementation in LITMUS$^{\mathrm{RT}}$ (which obviously is a prerequisite for obtaining realistic overhead measurements). Second, from our understanding, it requires a task to be split into subtasks whenever it requests a resource. Such subtasks are eligible for execution at different times based on the resource-usage history of prior (sub-)jobs. We do not consider it feasible to efficiently maintain a sufficiently complete resource usage history in-kernel at runtime. (Indeed, to the best of our knowledge, the period enforcer has never been implemented in any real OS.) Third, all tested, suspension-based synchronization protocols are affected by deferral blocking to the same extent. Thus, even if it were possible to avoid deferral blocking altogether, the relative performance of the algorithms is unlikely to differ significantly from our findings.

3.1 Performance on a Four-Processor System

We conducted schedulability experiments assuming four to 16 processors. In all cases, we used overhead values obtained from our four-processor test platform. (This is perhaps one limitation of our study. In reality, overheads on larger platforms might be higher, *e.g.*, due to greater bus contention or different caching behavior. We decided to simply use our four-processor overheads in all cases rather than "guessing" as to what overheads would be appropriate on larger systems.) In this subsection, we discuss experiments conducted to address three questions: When (if ever) does either FMLP variant perform worse than either PCP variant? When (if ever) is blocking by suspending a viable alternative to blocking by spinning? What parameters affect the performance of the tested algorithms most? In these experiments, a four-processor system was assumed; larger systems are considered in the next subsection.

Generated task systems. To answer the questions above, we conducted extensive experiments covering a large range of possible task systems. We varied **(i)** L in 40 steps over its range ($[0.5\mu s, 15.5\mu s]$), **(ii)** K in steps of one over its range ($[0, 9]$), and **(iii)** $\hat{U}$ in 40 steps over its range ($[0.1, 0.5]$), while keeping (in each case) all other task-set generation parameters constant so that schedulability could be determined as a function of L, K, and $\hat{U}$. In particular, we conducted experiments (i)–(iii) for constant assignments from *all* combinations of $\alpha \in \{0.5, 1, 2, 4\}$, $\hat{U} \in \{0.15, 0.3, 0.45\}$, $L \in \{3\mu s, 9\mu s, 15\mu s\}$, $K \in \{2, 5, 9\}$, and the four task period ranges defined earlier. For each sampling point, we generated (and tested for schedulability under each algorithm) 1,000 task sets, for a total 13,140,000 task sets.

Trends. It is clearly not feasible to present all 432 resulting graphs. However, the results show clear trends. We begin by making some general observations

[4] In fact, we have confirmed that some existing scheduling analysis (e.g., [21]) that uses the period enforcer is flawed [22]. Interestingly, in her now-standard textbook on the subject of real-time systems, Liu does not assume the presence of the period enforcer in her analysis of the D-PCP [16].

concerning these trends. Below, we consider a few specific graphs that support these observations.

In all tested scenarios, suspending was *never* preferable to spinning. In fact, in the vast majority of the tested scenarios, *every* generated task set was schedulable under spinning (the short FMLP variant). In contrast, many scenarios could not be scheduled under any of the suspension-based methods. The only time that suspending was ever a viable alternative was in scenarios with a small number of resources (*i.e.*, small K, low $\hat{U}$, high α) and relatively lax timing constraints (long, homogeneous periods). Since the short FMLP variant is clearly the best choice (from a schedulability point of view), we mostly focus our attention on the suspension-based protocols in the discussion that follows.

Overall, the long FMLP variant exhibited the best performance among suspension-based algorithms, especially in low-sharing-degree scenarios. For $\alpha = 0.5$, the long FMLP variant *always* exhibited better performance than both the M-PCP and D-PCP. For $\alpha = 1$, the long FMLP variant performed best in 101 of 108 tested scenarios. In contrast, the M-PCP was *never* the preferable choice for any α. Our results show that the D-PCP hits a "sweet spot" (which we discuss in greater detail below) when $K = 2$, $\hat{U} \leq 0.3$, and $\alpha \geq 2$; it even outperformed the long FMLP variant in some of these scenarios (but never the short variant). However, the D-PCP's performance quickly diminished outside this narrow "sweet spot." Further, even in the cases where the D-PCP exhibited the best performance among the suspension-based protocols, schedulability was very low. The M-PCP often outperformed the D-PCP; however, in *all* such cases, the long FMLP variant performed better (and sometimes significantly so).

The observed behavior of the D-PCP reveals a significant difference with respect to the M-PCP and FMLP. Whereas the performance of the latter two is mostly determined by the task count and tightness of timing constraints, the D-PCP's performance closely depends on the number of resources—whenever the number of resources does not exceed the number of processors significantly, the D-PCP does comparatively well. Since (under our task-set generation method) the number of resources depends directly on both K and α (and indirectly on $\hat{U}$, which determines how many tasks are generated), this explains the observed "sweet spot." The D-PCP's insensitivity to total task count can be traced back to its distributed nature—under the D-PCP, a job can only be delayed by events on its local processor and on remote processors where it requests resources. In contrast, under the M-PCP and FMLP, a job can be delayed transitively by events on all processors where jobs reside with which the job shares a resource.

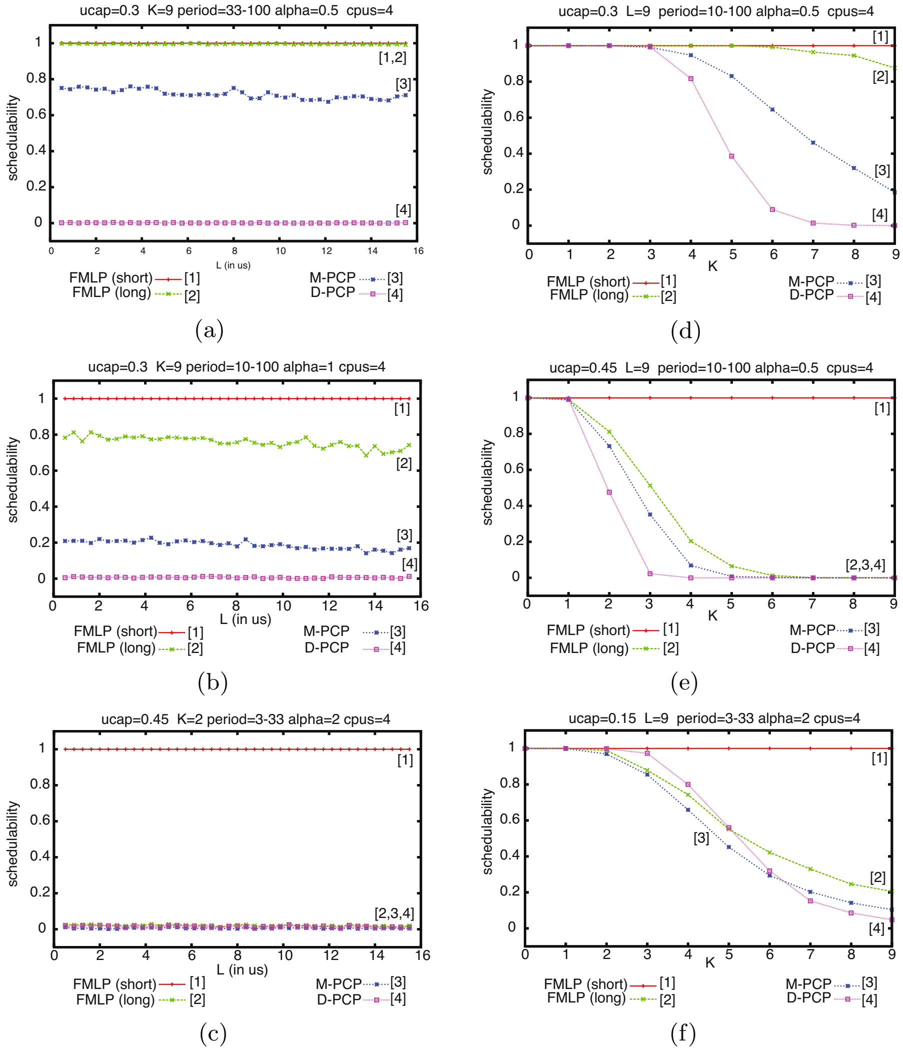
Example graphs. Insets (a)-(f) of Fig. 4 and (a)-(c) of Fig. 5 display nine selected graphs for the four-processor case that illustrate the above trends. These insets are discussed next.

Fig. 4 (a)-(c). The left column of graphs in Fig. 4 shows schedulability as a function of L for $K = 9$. The case depicted in inset (a), where $\hat{U} = 0.3$ and $\mathsf{p}(T_i) \in [33, 100]$, shows how both FMLP variants significantly outperform both the M-PCP and D-PCP in low-sharing-degree scenarios. Note how even the long variant achieves almost perfect schedulability. In contrast, the D-PCP fails to schedule any task set, while schedulability under the M-PCP hovers around 0.75. Inset (b), where $\hat{U} = 0.3$, $\mathsf{p}(T_i) \in [10, 100]$, and $\alpha = 1$, presents a more challenging situation: the wider range of periods and a higher sharing degree reduce the schedulablity of the FMLP (long) and the M-PCP significantly. Surprisingly, the performance of the D-PCP actually improves marginally (since, compared to inset (a), there are fewer resources). However, it is not a viable alternative in this scenario. Finally, inset (c) depicts a scenario where all suspension-based protocols fail due to tight timing constraints. Note that schedulability is largely independent of L; this is due to the fact that overheads outweigh critical-section lengths in practice.

Fig. 4 (d)-(f). The right column of graphs in Fig. 4 shows schedulability as a function of K for $L = 9\mu s$, $\mathsf{p}(T_i) \in [10, 100]$ (insets (d) and (e)) and $\mathsf{p}(T_i) \in [3, 33]$ (inset (g)), and $\hat{U}$ equal to 0.3 (inset (d)), 0.45 (inset (e)), and 0.15 (inset (f)). The graphs show that K has a significant influence on schedulability. Inset (d) illustrates the superiority of both FMLP variants in low-sharing-degree scenarios ($\alpha = 0.5$): the long variant exhibits a slight performance drop for $K \geq 6$, whereas the PCP variants are only viable alternatives for $K < 6$. Inset (e) depicts the same scenario with $\hat{U}$ increased to 0.45. The increase in the number of tasks (and thus resources too) causes schedulability under all suspension-based protocols to drop off quickly. However, relative performance remains roughly the same. Inset (f) presents a scenario that exemplifies the D-PCP's "sweet spot." With $\hat{U} = 0.15$ and $\alpha = 2$, the number of resources is very limited. Thus, the D-PCP actually offers somewhat better schedulability than the long FMLP variant for $K = 3$ and $K = 4$. However, the D-PCP's performance deteriorates quickly, so that it is actually the *worst* performing protocol for $K \geq 7$.

Fig. 5 (a)-(c). The left column of graphs in Fig. 5 shows schedulability as a function of $\hat{U}$. Inset (a) demonstrates, once again, the superior performance of both FMLP variants in low-sharing-degree scenarios ($\alpha = 0.5$, $K = 9$, $L = 3\mu s$, $\mathsf{p}(T_i) \in [10, 100]$). Inset (b) shows one of the few cases where the D-PCP outperforms the M-PCP at low sharing degrees ($\alpha = 1$, $K = 2$, $L = 3\mu s$, $\mathsf{p}(T_i) \in [3, 33]$). Note that the D-PCP initially performs as well as the long FMLP variant, but starting at $\hat{U} = 0.3$, fails more quickly. In the end, its performance is similar to that of the M-PCP. Finally, inset (c) presents one of the few cases where even the short FMLP variant fails to schedule all tasks sets. This graph represents the most taxing scenario in our study as each parameter is set to its worst-case value: $\alpha = 4$ and $K = 9$ (which implies high contention), $L = 15\mu s$, and $\mathsf{p}(T_i) \in [3, 33]$ (which leaves little slack for blocking terms). None of the suspension-based protocols can handle this scenario.

To summarize, in the four-processor case, the short FMLP variant was always the best-performing protocol, usually by a wide margin. Among the suspension-based protocols, the long FMLP variant was preferable most of the time, while the D-PCP was sometimes preferable if there were a small number (approximately four) resources being shared. The M-PCP was never preferable.



(a) (d)

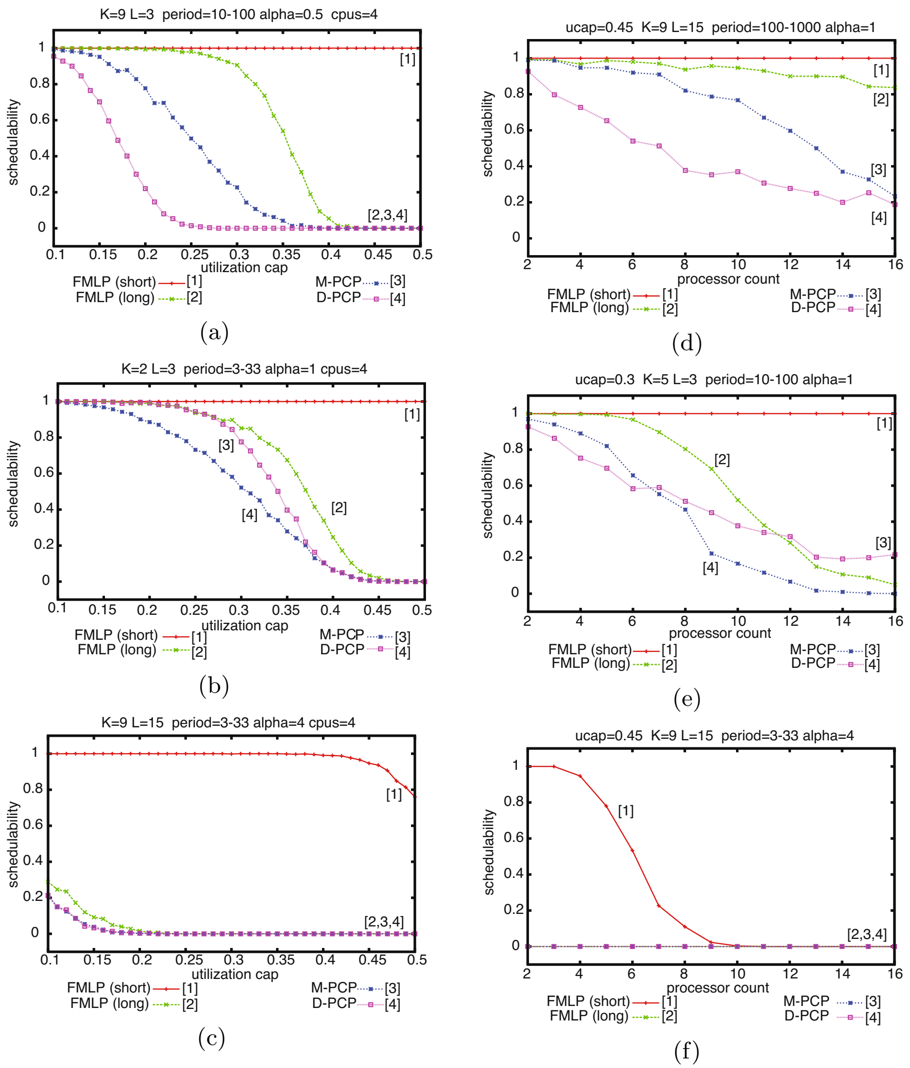
(b) (e)

(c) (f)

Fig. 4. Schedulability (the fraction of generated task systems deemed schedulable) as a function of **(a)-(c)** the maximum critical-section length L and **(d)-(f)** the per-job resource request bound K

3.2 Scalability

We now consider how the performance of each protocol scales with the processor count. To determine this, we varied the processor count from two to 16 for all possible combinations of α, $\hat{U}$, L, K, and periods (assuming the ranges for each defined earlier). This resulted in 324 graphs, three of which are shown in the right column of Fig. 5. The main difference between insets (d) and (e) of the figure is that task periods are large in (d) ($\mathsf{p}(T_i) \in [100, 1000]$) but small in (e)



(a)

(d)

(b)

(e)

(c)

(f)

Fig. 5. Schedulability as a function of **(a)-(c)** the per-processor utilization cap $\hat{U}$ and of **(d)-(f)** processor count

($\mathsf{p}(T_i) \in [10, 100]$). As seen, both FMLP variants scale well in inset (d), but the performance of the long variant begins to degrade quickly beyond six processors in inset (e). In both insets, the M-PCP shows a similar but worse trend as the long FMLP variant. This relationship was apparent in many (but not all) of the tested scenarios, as the performance of both protocols largely depends on the total number of tasks. In contrast, the D-PCP quite consistently does *not* follow the same trend as the M-PCP and FMLP. This, again, is due to the fact that the D-PCP depends heavily on the number of resources. Since, in this study, the total number of tasks increases at roughly the same rate as the number of processors, in each graph, the number of resources does not change significantly as the processor count increases (since α and K are constant in each graph). The fact that the D-PCP's performance does not remain constant indicates that its performance also depends on the total task count, but to a lesser degree. Inset (f) depicts the most-taxing scenario considered in this paper, *i.e.*, that shown earlier in Fig. 5 (c). None of the suspension-based protocols support this scenario (on any number of processors), and the short FMLP variant does not scale beyond four to five processors.

Finally, we repeated some of the four-processor experiments discussed in Sec. 3.1 for 16 processors to explore certain scenarios in more depth Although we are unable to present the graphs obtained for lack of space, we do note that blocking-by-suspending did not become more favorable on 16 processors, and the short FMLP variant still outperformed all other protocols in all tested scenarios. However, the relative performance of the suspension-based protocols did change, so that the D-PCP was favorable in more cases than before. This appears to be due to two reasons. First, as discussed above, among the suspension-based protocols, the D-PCP is impacted the least by an increasing processor count (given our task-set generation method). Second, the long FMLP variant appears to be somewhat less effective at supporting short periods for larger processor counts. However, schedulability was poor under all suspension-based protocols for tasks sets with tight timing constrains on a 16-processor system.

3.3 Impact of Overheads

In all experiments presented so far, all suspension-based protocols proved to be inferior to the short variant of the FMLP in all cases. As seen in Table 1 in the appendix, in the implementation of these protocols in LITMUS$^{\mathrm{RT}}$ that were used to measure overheads, the suspension-based protocols incur greater overheads than the short FMLP variant. Thus, the question of how badly suspension-based approaches are penalized by their overheads naturally arose. Although we believe that we implemented these protocols efficiently in LITMUS$^{\mathrm{RT}}$, perhaps it is possible to streamline their implementations further, reducing their overheads. If that were possible, would they still be inferior to the short FMLP variant?

To answer this question, we reran a significant subset of the experiments considered in Sec. 3.1, assuming *zero overheads* for all suspension-based protocols while charging full overheads for the short FMLP variant. The results obtained showed three clear trends: **(i)** given zero overheads, the suspension-based protocols achieve high schedulability for higher utilization caps before eventually degrading; **(ii)** when performance eventually degrades, it occurs less gradually than before (the slope is much steeper); and **(iii)** while the suspension-based protocols become more competitive (as one would expect), *they were still bested by the short FMLP variant in all cases*. Additionally, given zero overheads, the behavior of the M-PCP approached that of the suspension-based FMLP much more closely in many cases.

4 Conclusion

From the experimental study just described, two fundamental conclusions emerge. First, when implementing memory-resident resources (the focus of this paper), synchronization protocols that implement blocking by suspending are of questionable practical utility. This applies in particular to the M-PCP and D-PCP, which have been the de-facto standard for 20 years for supporting locks in multiprocessor real-time applications. Second, in situations in which the performance of suspension-based locks is not *totally* unacceptable (*e.g.*, the sharing degree is low, the processor count is not too high, or few global resources exist), the long-resource variant of the FMLP is usually the better choice than either the M-PCP or D-PCP (moreover, the FMLP allows resource nesting).

Although we considered a range of processor counts, overheads were measured only on a four-processor platform. In future work, we would like to obtain measurements on various larger platforms to get a more accurate assessment. On such platforms, overheads would likely be higher, which would more negatively impact suspension-based protocols, as their analysis is more pessimistic. Such pessimism is a consequence of difficulties associated with predicting which scheduling-related events may impact a task while it is suspended. In fact, it is known that suspensions cause intractabilities in scheduling analysis even in the uniprocessor case [24].

References

1. IBM and Red Hat announce new development innovations in Linux kernel. Press release (2007), http://www-03.ibm.com/press/us/en/pressrelease/21232.wss
2. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1(1), 6–16 (1990)
3. Baker, T.: Stack-based scheduling of real-time processes. Journal of Real-Time systems 3(1), 67–99 (1991)
4. Bisson, S.: Azul announces 192 core Java appliance (2006), http://www.itpro.co.uk/serves/news/99765/azul-announces-192-core-java-appliance.html
5. Block, A., Brandenburg, B., Anderson, J., Quint, S.: An adaptive framework for multiprocessor real-time systems. In: Proceedings of the 20th Euromicro Conference on Real-Time Systems, pp. 23–33 (2008)
6. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 71–80 (2007)
7. Brandenburg, B., Anderson, J.: Feather-Trace: A light-weight event tracing toolkit. In: Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp. 19–28 (2007)
8. Brandenburg, B., Anderson, J.: Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In: Proceedings of the 19th Euromicro Conference on Real-Time Systems, pp. 61–70 (2007)

9. Brandenburg, B., Anderson, J.: An implementation of the PCP, SRP, DPCP, MPCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 185–194 (2008)
10. Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., Anderson, J.: LITMUS^{RT}: A status report. In: Proceedings of the 9th Real-Time Linux Workshop, pp. 107–123. Real-Time Linux Foundation (2007)
11. Brandenburg, B., Calandrino, J., Block, A., Leontyev, H., Anderson, J.: Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 342–353 (2008)
12. Calandrino, J., Leontyev, H., Block, A., Devi, U., Anderson, J.: LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp. 111–123 (2006)
13. Devi, U.: Soft Real-Time Scheduling on Multiprocessors. PhD thesis, University of North Carolina, Chapel Hill, NC (2006)
14. Graunke, G., Thakkar, S.: Synchronization algorithms for shared-memory multiprocessors. IEEE Computer 23, 60–69 (1990)
15. Lehoczky, J., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In: Proceedings of the 10th IEEE International Real-Time Systems Symposium, pp. 166–171 (1989)
16. Liu, J.: Real-Time Systems. Prentice-Hall, Upper Saddle River (2000)
17. Lortz, V., Shin, K.: Semaphore queue priority assignment for real-time multiprocessor synchronization. IEEE Transactions on Software Engineering 21(10), 834–844 (1995)
18. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9(1), 21–65 (1991)
19. SUN Microsystems. SUN UltraSPARC T1 Marketing material (2008), http://www.sun.com/processors/UltraSPARC-T1/
20. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 116–123 (1990)
21. Rajkumar, R.: Synchronization In Real-Time Systems – A Priority Inheritance Approach. Kluwer Academic Publishers, Dordrecht (1991)
22. Rajkumar, R.: Private communication (April 2008)
23. Rajkumar, R., Sha, L., Lehoczky, J.P.: Real-time synchronization protocols for multiprocessors. In: Proceedings of the 9th IEEE International Real-Time Systems Symposium, pp. 259–269 (1988)
24. Ridouard, F., Richard, P., Cottet, F.: Negative results for scheduling independent hard real-time tasks with self-suspensions. In: Proceedings of the 25th IEEE International Real-Time Systems Symposium, pp. 47–56 (2004)
25. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers 39(9), 1175–1185 (1990)
26. Shankland, S., Kanellos, M.: Intel to elaborate on new multicore processor (2003), http://news.zdnet.co.uk/hardware/chips/0,39020354,39116043,00.htm

Appendix

To obtain the overheads required in this paper, we used the same methodology that we used in the prior study concerning EDF scheduling [11]. For the sake of completeness, the approach is summarized here.

In real systems, task execution times are affected by the following sources of overhead. At the beginning of each quantum, *tick scheduling overhead* is incurred, which is the time needed to service a timer interrupt. Whenever a scheduling decision is made, a *scheduling cost* is incurred, which is the time taken to select the next job to schedule. Whenever a job is preempted, *context-switching overhead* and *preemption overhead* are incurred; the former term includes any non-cache-related costs associated with the preemption, while the latter accounts for any costs due to a loss of cache affinity.

When jobs access shared resources, they incur an *acquisition* cost. Similarly, when leaving a critical section, they incur a *release* cost. Further, when a system call is invoked, a job will incur the cost of *switching from user mode to kernel mode* and back. Whenever a task should be preempted while it is executing a non-preemptive (NP) section, it must notify the kernel when it is *leaving its NP-section*, which entails some overhead. Under the D-PCP, in order to communicate with a remote agent, a job must *invoke* that agent. Similarly, the agent also incurs overhead when it receives a request and *signals* its completion.

Accounting for overheads. Task execution costs can be inflated using standard techniques to account for overheads in schedulability analysis [16]. Care must be taken to also properly inflate resource request durations. Acquire and release costs contribute to the time that a job holds a resource and thus can cause blocking. Similarly, suspension-based synchronization protocols must properly account for preemption effects *within* critical sections. Further, care must be taken to inflate task execution costs for preemptions and scheduling events due to suspensions in the case of contention. Whenever it is possible for a lower-priority job to preempt a higher-priority job and execute a critical section,[5] the event source (*i.e.*, the resource request causing the preemption) must be accounted for in the demand term of all higher-priority tasks. One way this can be achieved is by modeling such critical sections as special tasks with priorities higher than that of the highest-priority normal task [16].

Implementation. To obtain realistic overhead values, we implemented the M-PCP, D-PCP, and FMLP under P-SP scheduling in LITMUS$^{\mathrm{RT}}$. A detailed description of the LITMUS$^{\mathrm{RT}}$ kernel and its architecture is beyond the scope of this paper. Such details can be found in [10]. Additionally, a detailed account of the implementation issues encountered, and relevant design decisions made, when implementing the aforementioned synchronization protocols in LITMUS$^{\mathrm{RT}}$ can be found in [9]. LITMUS$^{\mathrm{RT}}$ is open source software that can be downloaded freely.[6]

[5] This is possible under all three suspension-based protocols considered in this paper: a blocked lower-priority job might resume due to a priority boost under the FMLP and M-PCP and might activate an agent under the D-PCP.

[6] http://www.cs.unc.edu/∼anderson/litmus-rt.

Limitations of real-time Linux. There is currently much interest in using Linux to support real-time workloads, and many real-time-related features have recently been introduced in the mainline Linux kernel (such as high-resolution timers, priority inheritance, and shortened non-preemptable sections). However, to satisfy the strict definition of hard real-time, all worst-case overheads must be known in advance and accounted for. Unfortunately, this is currently not possible in Linux, and it is highly unlikely that it ever will be. This is due to the many sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), as well as the lack of determinism on the hardware platforms on which Linux typically runs. The latter is especially a concern, regardless of the OS, on multiprocessor platforms. Indeed, research on timing analysis has not matured to the point of being able to analyze complex interactions between tasks due to atomic operations, bus locking, and bus and cache contention. Without the availability of timing-analysis tools, overheads must be estimated experimentally. Our methodology for doing this is discussed next.

Measuring overheads. Experimentally estimating overheads is not as easy as it may seem. In particular, in repeated measurements of some overhead, a small number of samples may be "outliers." This may happen due to a variety of factors, such as warm-up effects in the instrumentation code and the various non-deterministic aspects of Linux itself noted above. In light of this, we determined each overhead term by discarding the top 1% of measured values, and then taking the maximum of the remaining values. Given the inherent limitations associated with multiprocessor platforms noted above, we believe that this is a reasonable approach. Moreover, the overhead values that we computed should be more than sufficient to obtain a valid comparison of the D-PCP, M-PCP, and FMLP under consideration of real-world overheads, which is the focus of this paper.

The hardware platform used in our experiments is a cache-coherent SMP consisting of four 32-bit Intel Xeon(TM) processors running at 2.7 GHz, with 8K L1 instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory. Overheads were measured and recorded using *Feather-Trace*, a light-weight tracing toolkit developed at UNC [7]. We calculated overheads by measuring the system's behavior for task sets randomly generated as described in Sec. 3. To better approximate worst-case behavior, longer critical sections were considered in order to increase contention levels (most of the measured overheads increase with contention).

We generated a total of 100 task sets and executed each task set for 30 seconds under each of the suspension-based synchronization protocols[7] while recording system overheads. (In fact, this was done several times to ensure that the determined overheads are stable and reproducible.) Individual measurements were determined by using Feather-Trace to record timestamps at the beginning and end of the overhead-generating code sections, *e.g.*, we recorded a

[7] Overheads for the short FMLP variant were already known from prior work [11] and did not have to be re-determined.

Table 1. (a) Worst-case overhead values (in μs), on our four-processor test platform obtained in prior studies. **(b)** Newly measured worst-case overhead values, on our four-processor test platform, in μs. These values are based on 86,368,984 samples recorded over a total of 150 minutes.

Overhead	Worst-Case
Preemption	42.00
Context-switching	9.25
Switching to kernel mode	0.34
Switching to user mode	0.89
Leaving NP-section	4.12
FMLP short acquisition / release	2.00 / 0.87

(a)

Overhead	Worst-Case
Scheduling cost	6.39
Tick	8.08
FMLP long acquisition / release	2.74 / 8.67
M-PCP acquisition / release	5.61 / 8.27
D-PCP acquisition / release	4.61 / 2.85
D-PCP invoke / agent	8.36 / 7.15

(b)

timestamp before acquiring a resource and after the resource was acquired (however, no blocking is included in these overhead terms). Each overhead term was determined by plotting the measured values obtained to check for anomalies, and then computing the maximum value (discarding outliers, as discussed above).

Measurement results. In some case, we were able to re-use some overheads determined in prior work; these are shown in inset (a) of Tab. 1. In other cases, new measurements were required; these are shown in inset (b) of Tab. 1.

The preemption cost in Table 1 was derived in [12]. In [12], this cost is given as a function of working set size (WSS). These WSSs are *per quantum*, thus reflecting the memory footprint of a particular task during a 1-ms quantum, rather than over its entire lifetime. WSSs of 4K, 32K, and 64K were considered in [12], but we only consider the 4K case here, due to space constraints. Note that larger WSSs tend to decrease the competitiveness of methods that suspend, as preemption costs are higher in such cases. Thus, we concentrate on the 4K case to demonstrate that, even in cases where such methods are most competitive, spinning is still preferable. The other costs shown in inset (a) of Table 1 were determined in [11].

A Self-stabilizing Marching Algorithm for a Group of Oblivious Robots

Yuichi Asahiro[1], Satoshi Fujita[2], Ichiro Suzuki[3], and Masafumi Yamashita[4]

[1] Dept. of Social Information Systems, Faculty of Information Science, Kyushu Sangyo University, 2-3-1 Matsukadai, Higashi-ku, Fukuoka 813-8503, Japan
asahiro@is.kyusan-u.ac.jp

[2] Dept. of Electrical Engineering, Faculty of Engineering, Hiroshima University, Kagamiyama 1 chome 4-1, Higashi-Hiroshima 739-8527, Japan
fujita@se.hiroshima-u.ac.jp

[3] Dept. of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, PO Box 784, Milwaukee, WI 53201, USA
suzuki@cs.uwm.edu

[4] Dept. of Computer Science and Communication Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka, Fukuoka 819-0395, Japan
mak@csce.kyushu-u.ac.jp

Abstract. We propose a self-stabilizing marching algorithm for a group of oblivious robots in an obstacle-free workplace. To this end, we develop a distributed algorithm for a group of robots to transport a polygonal object, where each robot holds the object at a corner, and observe that each robot can simulate the algorithm, even after we replace the object by an imaginary one; we thus can use the algorithm as a marching algorithm. Each robot independently computes a velocity vector using the algorithm, moves to a new position with the velocity for a unit of time, and repeats this cycle until it reaches the goal position. The algorithm is oblivious, i.e., the computation depends only on the current robot configuration, and is constructed from a naive algorithm that generates only a selfish move, by adding two simple ingredients. For the case of two robots, we theoretically show that the algorithm is self-stabilizing, and demonstrate by simulations that the algorithm produces a motion that is fairly close to the time-optimal motion. For cases of more than two robots, we show that a natural extension of the algorithm for two robots also produces smooth and elegant motions by simulations as well.

Keywords: motion coordination, marching, flocking, self-stabilizing algorithm, oblivious algorithm.

1 Introduction

Motion coordination among mobile robots with distributed information is a common hot research area in automatic control, robotics and computer science [6, 19, 22, 26]. Among many challenging problems arising in this area, we tackle the marching problem for mobile robots, where marching means moving in formation.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 125–144, 2008.

Self-organized behavior of a school of fish and a flock of geese has attracted many researchers [8,20,30,41]. The snapshot of a school of fish however changes time by time, and the flocking problem in general is not interested in rigid relative positions among mobile agents. The formation problem for mobile robots, on the other hand, looks for local rules to form a given geometrical pattern from any initial configuration. The formation problem was first investigated by Sugihara and Suzuki [37,38], and the formable patterns are characterized using the terminology of distributed computing by Suzuki and Yamashita [39,40]. Then extensive works have carried out in particular in this decade [2,9,12,13,14,17,18,21,24,31,32,34,35]. They all modeled a robot as a dimensionless point and investigated the problem as a robot motion planning problem, ignoring mechanical constraints [27]. Cao et al. [10] and Debest [15] surveyed early researches in control and robot societies. Also many research taking into account mechanical constraints and obstacles then followed (e.g., [7,23,43]).

This paper considers the marching problem, which asks for a control algorithm for a flock of agents to move to a goal position keeping the formation. A typical survey paper by Cao et al. [10] could cite only a couple of papers on marching a decade ago. Quite a few research projects have been conducted, but many of them are classified into either a centralized or a leader-follower approach [1,20,25,29,36], a distributed approach using some navigation device [16,42], or an artificial potential approach [28,33]. In these previous works, the quality of marching route has not been discussed seriously, which motivates this paper.

In this paper we focus on the quality of route. Let us move two robots keeping their distance unchanged or carrying a "ladder." If we ask the robots to rotate the ladder about 180°, they are likely to rotate it with one robot being the center as in Fig. 4[1]. Fig. 1 (left) shows another move, which is definitely more elegant and smoother. Smoother motions can be more "efficient" than those that are not. Indeed, the motion shown in Fig. 1 (left) is time optimal[2]. It is worth emphasizing that the robots' moves are by no means straightforward; both of the robots can never move straight to their final positions in this instance to achieve the time optimal motion. In general, the robots can have conflicting interests, and they have to resolve the conflict to achieve good overall performance. Our goal is to design a marching algorithm that realizes such an elegant, smooth and time-efficient march, even under the presence of transient sensor and control errors. Fig. 9 (left) shows how our algorithm rotates the ladder about 180°.

Specifically, we develop a distributed algorithm called $G+$ "Greedy Plus" below for a group of robots to transport a polygonal object, where each robot holds the object at a corner, and observe that each robot can simulate the algorithm, even after we replace the object by an imaginary one; we thus can

[1] In Fig. 4, the circles and a line segment represent the robots and the ladder they carry, respectively.

[2] Parameters L_B, α and β will be explained later. An analytical method for computing a time-optimal motion of this problem for two robots, under the assumption that the robots' speed is either 0 or a given constant at any moment, is reported in [11]. We used this method to calculate this optimal motion.

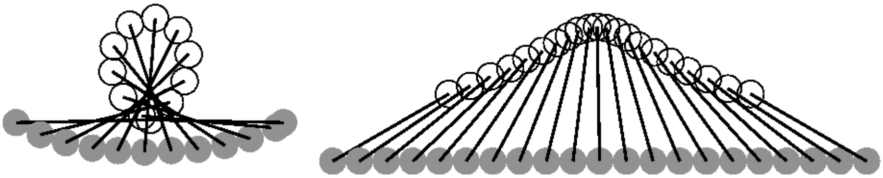


Fig. 1. Time-optimal motions for instances I_1 (left; $L_B = 2$, $\alpha = 1°$, $\beta = 179°$) and I_2 (right; $L_B = 4$, $\alpha = 30°$, $\beta = 150°$), respectively

use the algorithm as a marching algorithm. Each robot independently computes a velocity vector using the algorithm, moves to a new position with the velocity for a unit of time, and repeats this cycle until it reaches the goal position. The algorithm is oblivious, i.e., the computation depends only on the current robot configuration, and is constructed from a naive algorithm called G below that generates only a selfish move, by adding two simple ingredients. We first design and evaluate the algorithm $G+$ for two robots and will later apply the idea to the case of three or more robots.

To design $G+$, we first examine a straightforward greedy algorithm G, in which each robot simply tries to move toward its goal location without taking into account the goal of the other robot. Although G can generate a motion for successfully transporting a ladder in almost all instances we consider, the resulting motion tends to lack smoothness and efficiency (Fig. 4 (left) shows how G rotates the ladder about 180°); the finish time can be greater by up to 163.5% over that of a time-optimal motion. $G+$ is based on the following simple idea; at any moment each robot pursues its individual interest of moving toward its goal position, while at the same time making minor adjustments to its course based on the other robot's current and goal locations and the final orientation of the object. $G+$ generates a motion that is smooth and fairly close to the time-optimal motion (Fig. 9 (left) shows how $G+$ rotates the ladder about 180°), whose finish time is only up to 9.6% greater than the optimal.

In fact, the algorithm $G+$ is a refinement of the algorithm called ALG1 proposed in [3]; we can show that $G+$ is correct, while ALG1 is not. Combining it with the fact that $G+$ is oblivious, i.e., it determines the next position independently of the motions in the past, we can show that $G+$ is self-stabilizing – the system works stably even in the presence of transient sensor and control errors.

We next discuss how to apply $G+$ to three or more robots (up to nine robots), and demonstrate by computer simulations that $G+$ seems to work fairly well, although we do not have data on time-optimal motions against which the simulation results should be compared.

It is worth mentioning that our approach is not a variation of well-known negative feedback control, in which a robot attempts to reduce the deviation from an optimal trajectory that has been given a-priori. In our algorithm, the robots are not provided with any optimal path in advance. Their trajectories are determined only as a result of their interaction with each other.

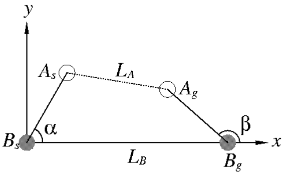


Fig. 2. The setup of the problem. We represent robots A and B by hollow and gray circles, respectively.

The paper is organized as follows: Section 2 explains the robot model. Section 3 examines algorithm G. In Sect. 4, we introduce algorithm $G+$, demonstrates its performance, and show that it is correct and self-stabilizing. Extensions of $G+$ for three or more robots are discussed in Sect. 5. We then conclude the paper by giving some remarks in Sect. 6.

2 Modeling Two Robots

Consider two robots A and B carrying a ladder of length ℓ in an obstacle-free workspace. We represent the robots and the ladder as two disks and a line segment, respectively, as is shown in Fig. 2. We assume that the robots are identical; they have the same maximum speed and execute the same algorithm.

Let A_s, B_s and A_g, B_g be the start and goal positions of the robots, respectively. We let $L_A = |A_sA_g|$ and $L_B = |B_sB_g|$. α and β denote the angles that the ladder makes with $\overline{B_sB_g}$ at the start and goal positions, respectively. We can describe instances of the problem using L_B, α and β.

Each end of the ladder is assumed to be attached to a force sensor as in [5], that we model as an ideal spring at the center of each robot. Since the distance D between the centers of the robots does not always equal ℓ during motion, the force sensor produces an *offset vector* $\mathbf{o}$ of size $|(D-\ell)/2|$ from the center of the robot to the end of the ladder. The offset vectors at both ends are equal in size and opposite in direction[3].

We assume that at any moment, robot A (resp. B) knows A_s, A_g, B_g, $\mathbf{o}_A$ (resp. B_s, B_g, A_g, $\mathbf{o}_B$) and ℓ (provided that there are no sensor errors). Since A (resp. B) can computes B_s (resp. A_s) using $\mathbf{o}_A = -\mathbf{o}_B$, without loss of generality, we may assume that at any moment, they know A_s, A_g, B_s, B_g, $\mathbf{o}_A$, $\mathbf{o}_B$, ℓ, L_A, L_B, α and β. A *(distributed) algorithm* for robot R with maximum speed V is any procedure that computes a *velocity vector* $\mathbf{v}_R$ with $|\mathbf{v}_R| \leq V$ from some of A_s, A_g, B_s, B_g, $\mathbf{o}_A$, $\mathbf{o}_B$, ℓ, L_A, L_B, α and β.

We assume that a robot R repeatedly computes $\mathbf{v}_R$ and moves to a new position with velocity $\mathbf{v}_R$ for unit time. For simplicity we use discrete time and

[3] Here we ignore the acceleration of the robots and assume that sufficiently long period of time for spring relaxation is given to the robots before sensing the current size of the offset vector.

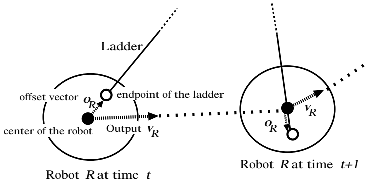


Fig. 3. The model of a robot

assume that both robots compute their respective velocity vectors and move to their new positions at time instances $0, 1, \cdots$. See Fig. 3.

The *finish time* is the time t_f when both robots arrive at their respective goal positions. The *delay* is then defined to be $(t_f - t_o)/t_o \times 100(\%)$, where t_o is the finish time of a time-optimal motion. We use the size of the offset vector during a motion to evaluate the "smoothness" of a motion.

3 Naive Algorithm *G* for Two Robots

In this section we describe and observe a naive greedy algorithm G. We shall later modify it in Sect. 4 to obtain algorithm $G+$. In the following description of G, R denotes either robot A or B, V is the maximum speed of both robots, and s is a spring constant.

[**Algorithm** G **for robot** R]

Step 1: If $L_R \leq V$ then move to R_g and terminate. Otherwise, let $\mathbf{t}_R$ be a vector directed from R_s to R_g such that $|\mathbf{t}_R| = V$.
Step 2: Scale the offset vector by $\mathbf{h}_R = s\mathbf{o}_R$.
Step 3: Set $\mathbf{T}_R = \mathbf{t}_R + \mathbf{h}_R$.
Step 4: Scale the size of $\mathbf{T}_R$ to V and move at velocity $\mathbf{T}_R$ for a unit of time. Go to Step 1.

Note that $\mathbf{T}_R$ consists of two components: $\mathbf{t}_R$ (move toward the goal) and $\mathbf{h}_R$ (move to reduce the offset). In **Step 4** $\mathbf{T}_R$ is scaled to V that makes the robot always move at the maximum speed.

Let us observe the performance of G by conducting computer simulation. For computer simulation, we use $\ell = 1$ and $V = 0.01$. The radius of a robot is 0.1. We use spring constant $s = 0.25$, which was found to be large enough to keep the endpoints of the ladder inside the circles representing robots' bodies during the simulation. To reduce the number of instances to examine, we consider only the cases $L_B = 2$ and 4, $0° \leq \alpha \leq 90°$, and $\beta = 180° - \alpha$ (see Fig. 2). These cases include two representative situations:

- The distance to the goal is small, and the robots must rotate the ladder quickly, as in instance I_1 ($L_B = 2$, $\alpha = 1°$, $\beta = 179°$), whose time-optimal motion is shown in Fig. 1 (left).

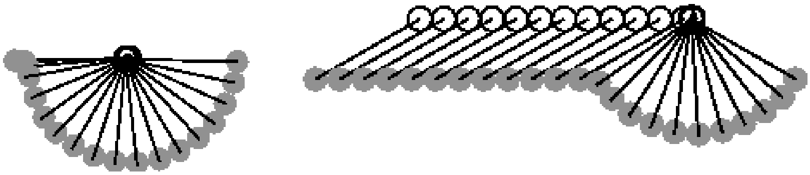


Fig. 4. The motions by G for instance I_1 (left) and I_2 (right), respectively

- The distance to the goal is large, and the robots do not need to rotate the ladder quickly, as in instance I_2 ($L_B = 4$, $\alpha = 30°$, $\beta = 150°$), whose time-optimal motion is shown in Fig. 1 (right).

Note that only the relative positioning of the initial and goal positions of the ladder is important. That is, by interchanging the initial and goal positions, and by interchanging endpoints A and B, the above setting covers the following cases also, hence, need not be discussed separately:

- The case $\alpha > 90°$ and $\beta = 180° - \alpha$ (rotating the ladder clockwise).
- The case $-90° \leq \alpha < 0$, and $\beta = -180° - \alpha$ (this is a symmetric case)

Although this configuration setting does not cover all the possibilities, we consider that it is a reasonable subset of the infinite instances: For example the case $\beta \neq 180° - \alpha$ for $0 < \alpha < 90°$ is not included explicitly in the above setting. However, the robots fall into such a configuration at some intermediate step during the motion, because the angle of the ladder gradually changes and the robots determine their motion based only on the current and goal positions. Hence if the algorithm works well for the above configuration setting, we can expect that it also works for the case $\beta \neq 180° - \alpha$.

The two figures of Fig. 4 show the motions by G for instances I_1 (left) and I_2 (right), respectively. The finish times are 340 and 439, respectively.

First, let us observe that the trajectories of the robots in these figures look quite different from the smoother motions shown in Fig. 1. Robot A temporarily "yields" to generate a smooth motion in Fig. 1 (left), while it does not in Fig. 4 (left). Both translation and rotation take place simultaneously in Fig. 1 (right), while in Fig. 4 (right), the ladder starts to rotate only toward the end of the motion. That is, rotation can occur only as a result of the robots' individual moves toward their goal positions, and that explains why in Fig. 4 (right) the ladder first translates without any rotation.

Here we would like to note the effect of the spring constant s: Intuitively speaking, if s is smaller, the gray robot tends to move (more) straight to the goal with narrowing the distance from the other robot, which largely breaks the formation of the robots (The evaluation of the motions based on this criteria is mentioned below).

The two figures of Figs. 5 show for $L_B = 2$ (left) and $L_B = 4$ (right), respectively, the finish times of the motions generated by G and those of time-optimal

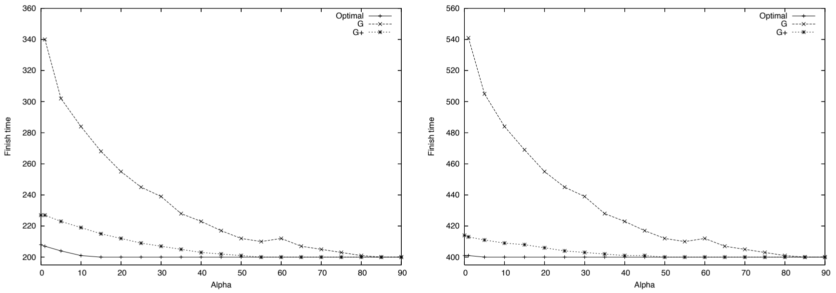


Fig. 5. Finish times of G, $G+$ and a time-optimal motion, for $L_B = 2$ (left) and $L_B = 4$ (right) with $0° \leq \alpha \leq 90°$ (excluding $\alpha = 0°$ for G)

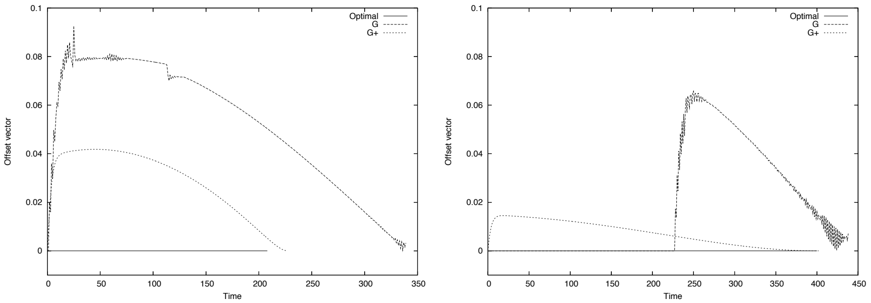


Fig. 6. Offset vector size for instances I_1 (left) and I_2 (right), by G (Fig. 4, left), $G+$ (Fig. 9, left), and a time-optimal motion (Fig. 1, left)

motions, for $\alpha = 1°, 5°, 10°, \cdots, 90°$. (Ignore the plot for $G+$ for now.) Note that for $L_B = 4$, the finish time of a time-optimal motion almost always equals $L_B/V(= 400)$, which indicates that the robot B can move straight to the goal position in an optimal motion.

For both $L_B = 2$ and 4, G generates a motion that is almost time-optimal if $\alpha \geq 70°$ (and hence, the required amount of rotation is small). However, the performance drops significantly as α becomes smaller (requiring more rotation). The worst case (among those we observed) is when $L_B = 2$ and $\alpha = 1°$ (Fig. 4, left), where the finish time of 340 by G is about 163.5% greater than a time-optimal motion's 208. (That is, the delay is 63.5%.)

Let us now discuss the smoothness of the motions that G generates. Figs. 6 (left) and 6 (right) show the offset vector size $|\mathbf{o}_A|$ $(= |\mathbf{o}_B|)$ during the motion generated by G and in a time-optimal motion, for instances I_1 and I_2, respectively. (Again, ignore the plot for $G+$ for now.) Note that the curves in these figures end at the finish times of the respective motions. It is worth noting that the offset vector size remains zero in the time-optimal motion.

The curves for G in these figures suddenly jump to a high peak when the ladder starts to rotate, and remains relatively high during rotation. This happens at time zero in Fig. 6 (left), and 229 in Fig. 6 (right). In contrast, the curve

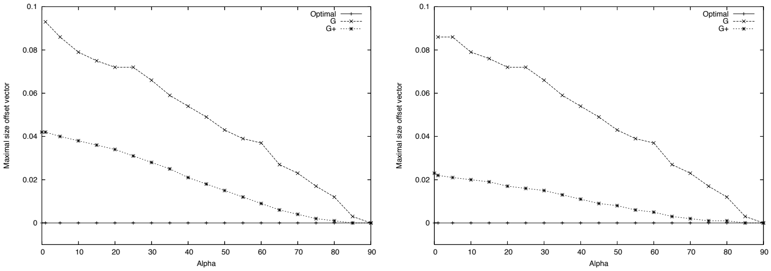


Fig. 7. Peak offset vector size by G, $G+$ and a time-optimal motion, for $L_B = 2$ (left) and $L_B = 4$ (right), respectively

in Fig. 6 (right) stays at zero before time 229 when the ladder is translated but not rotated. Furthermore, in Fig. 6 (left), the curve stays very high for a long period around time 20 to 150, indicating continuous severe stress that might be unacceptable to physical robots. Especially the motions by G produce oscillating offset vectors, e.g., around time 30 for I_1, that might be unacceptable for practical robots as well. The spring constant s affects this stress of motion. The above observation is obtained setting $s = 0.25$ and so we may be able to choose more suitable (actually larger) value for s in order to keep the size of offset vector small. However, the larger s causes the longer finish time because the robots have to detour compared to the motions obtained by $s = 0.25$. The difficult point here is that we need to choose a value for s which can achieve fastness and smoothness at the same time.

Figs. 7 (left) and 7 (right) show the peak offset vector size observed during a motion generated by G for $L_B = 2$ and $L_B = 4$, respectively, for various values of α. As observed above, the motion of G is divided into two parts, translation of the ladder followed by rotation. Since offset vectors become large during rotation, the distance to the goal does not have much effect here. Therefore the results for $L_B = 2$ and $L_B = 4$ are very similar. The maximum size of the offset vectors gradually decreases as α increases and hence smaller rotation is required.

To summarize, we observe that G tends to separate translation and rotation. This results in a motion that is less smooth because of a sudden transition between the two phases. We believe that there are two reasons for the separation.

- There is no explicit mechanism to rotate the ladder. Consequently, robot A never moves away from its goal location in Fig. 4 (left) to assist robot B to rotate the ladder.
- The robots do not utilize the information on the distances to their respective goals. Consequently, both robots move at the same speed (and hence, the ladder is not rotated at all), during translation in Fig. 4 (right), even though robot B is farther away from its goal than robot A.

It is conceivable that a smoother, faster motion can result if translation and rotation are merged by resolving these issues.

4 Algorithm $G+$ for Two Robots

Based on the observations in the last section, we add two features to G:

- When the distances to the goal positions differ between the robots, the robot closer to its goal reduces its speed.
- The velocity vector that each robot computes has a third component, called *rotation vector*, whose magnitude is proportional to the amount of rotation needed before the ladder reaches the goal location.

The resulting algorithm $G+$ is described below for robot R. As before, R is either A or B, and R' denotes the other robot, e.g., if $R = A$ then $R' = B$. V is the robots' maximum speed. Note again that V is the maximum distance that a robot can move in a unit of time. In $G+$, we move both robots to their goal positions, as soon as the ladder reaches a position sufficiently close to the goal position. More specifically, if $\max\{L_R, L_{R'}\} \le V$, then R moves to R_g and terminates. We assume that V has been chosen to make this move feasible. Parameters $s > 0$, $t \ge 0$ and $u \ge 0$ will be explained shortly.

[**Algorithm** $G+$ **for robot** R]

```
Step 1: Set $L_{max} = \max\{L_R, L_{R'}\}$. If $L_{max} \le V$ then move to $R_g$ and
    terminate. Otherwise, go to Step 2.
Step 2: Let $\mathbf{t}_R$ be a vector directed from $R_s$ to $R_g$ such that $|\mathbf{t}_R| = V \cdot (\frac{L_R}{L_{max}})^t$.
Step 3: Let $\mathbf{r}_R$ be a rotation vector of length $u(\beta - \alpha)/L_{max}$. The direction
    of $\mathbf{r}_R$, which is perpendicular to the ladder, is set to (i) $\alpha + \pi/2$ if $L_R \le$
    $L_{R'}$, and (ii) $\alpha - \pi/2$ otherwise, in an attempt to reduce $\beta - \alpha$ (favoring a
    counterclockwise rotation if $\beta - \alpha = 180°$).
Step 4: Scale the offset vector by $\mathbf{h}_R = s\mathbf{o}_R$.
Step 5: $\mathbf{T}_R = \mathbf{t}_R + \mathbf{r}_R + \mathbf{h}_R$.
Step 6: Compute $\mathbf{T}_{R'}$, following Steps 1–5 for $R'$.
Step 7: Let $\mathbf{v}_R = V \cdot \frac{\mathbf{T}_R}{\max\{|\mathbf{T}_R|, |\mathbf{T}_{R'}|\}}$. Move at velocity $\mathbf{v}_R$ for a unit of time.
    Go to Step 1.
```

$G+$ uses three parameters s, t and u. Parameter s, which is set to 0.25 as in G, determines the sensitivity of the robot to the offset vector. Larger values of t slows down further the robot that is closer to the goal, and parameter u determines the effect of the rotation vector. Note that if $t = 0$ and $u = 0$, then $G+$ reduces to G. Since the motion of the robots depends on t and u, it may seem at first that we need to set them to reasonably good values for each given instance, to obtain a smooth motion that is close to the time-optimal motion. As we demonstrate next, however, it turns out that a fixed pair of values for t and u can be used to such a smooth motion for a wide range of instances.

Figs. 8 (left) and 8 (right) show, for instances I_1 and I_2, respectively, the finish times of the motions generated by $G+$ for $t, u = 0, 1, \cdots, 10$. The results in these figures exemplify what we observed through extensive simulation using a large number of instances. That is, as is shown in Fig. 8 (left), in those instances in which the ladder must be rotated quickly, using any value of $t \ge 1$ and $u = 1$

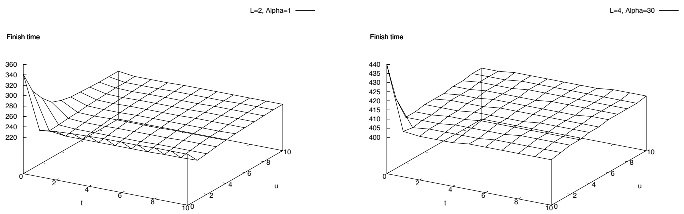


Fig. 8. Finish time of the motions by $G+$ for instance I_1 (left) and I_2 (right), for $t, u = 0, 1, \cdots, 10$

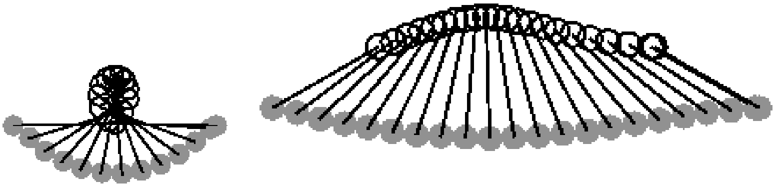


Fig. 9. The motions by $G+$ with $t = 1$ and $u = 1$, for instance I_1 (left) and I_2 (right), respectively

often gives a good finish time. If the ladder need not be rotated very quickly, then as shown in Fig. 8 (right) the finish time becomes less dependent on t and u, and we can expect good performance using any value of $t \geq 1$ and $u = 0$. In addition to I_1 and I_2, we tested more configurations, e.g., $L_B = 3, 4, 5, 6$ and $\alpha = 1°$, and also $L_B = 2$ and $\alpha = 5°, 10°, 15°, 20°$. We omit the detailed results here due to the space limitation, however for all the tested configurations the obtained charts look like those in Fig. 8. Based on the above observation, in the following we use $t = 1$ and $u = 1$ to evaluate $G+$[4].

Figs. 9 (left) and 9 (right) show the motions generated by $G+$ for instances I_1 and I_2, respectively. These motions resemble the time-optimal motions in Figs. 1 (left) and 1 (right) more closely than those by G shown in Figs. 4 (left) and 4 (right). Specifically, in Fig. 9 (left), robot A (the hollow circle) temporarily leaves its position and returns to it in order to rotate the ladder quickly. In Fig. 9, translation and rotation take place simultaneously, and robot B (that has to move more than the other robot) can move nearly straight to its destination.

The simulation results indicate that $G+$ generates a faster and smoother motion than G. See Fig. 5 and for a comparison of the finish times of $G+$, G and a time-optimal motion. For $\alpha \leq 80°$ the finish time of $G+$ is smaller than that of G. (For $\alpha > 80°$, both G and $G+$ generate a motion whose finish time equals that of a time-optimal motion.) The delay of $G+$ in the motion of Fig. 9

[4] Of course, $t = 1$, $u = 1$ is not optimal for all instances. For instance, setting $t = 2$ and $u = 1$ gives a slightly better finish time in Fig. 8 (right). Also, currently, we have no method to obtain "theoretically" best values for t and u. In practice, physical robots should be able to have a database of good t-u pairs for various L_B, α and β.

(left) for instance I_1 is only 9.6% (this is the worst case we observed for $G+$), as opposed to 63.5% of G in Fig. 4 (left). For instance I_2, the delay is 3.5% in the motion of Fig. 9 (right) by $G+$, as opposed to 9.8% in the motion of Fig. 4 (right) by G. Figs. 6 (left) and 6 (right), respectively, show that the offset vector size is considerably smaller in the motions of Figs. 9 (left) and 9 (right) by $G+$, compared to that of Figs. 4 (left) and 4 (right) by G. Figs. 7 show that the peak offset vector size is smaller in $G+$ than in G.

We next show that $G+$ is correct, i.e., for arbitrary initial and goal positions of the ladder, using Algorithm $G+$, robots A and B can transport the ladder to its goal position, provided that there are no sensor or control errors.

Theorem 1. *Suppose that $u > 0$. Then Algorithm $G+$ is correct.*

Proof. We give an outline of the proof. If $L_{max} = \max\{L_A, L_B\} \leq V$, then $G+$ terminates after the robots transport the ladder to the goal position in **Step 1**. We show that eventually $L_{max} \leq V$ holds. Suppose that $\alpha \neq \beta$. Since $u > 0$, α gets closer to β in each iteration of $G+$. Once α turns to be equal to β, the rest of the work for the robots is just moving straight to the goal position; the rotation vector is no longer needed and so it has size 0 in Step 3. Thus the target vectors $\mathbf{t}_A$ and $\mathbf{t}_B$ of the robots can have opposite directions only in the very first iterations, and hence in subsequent iterations, $\mathbf{t}_A$ and $\mathbf{t}_B$ together have the effect of moving the center of the ladder closer to its center in the goal position. The robots' offset correction vectors $\mathbf{h}_A$ and $\mathbf{h}_B$ are opposite in direction and equal in magnitude, and hence they do not affect the movement of the center of the ladder. Even if the center of the ladder stays exactly at the center of the goal positions of the ladder, it does not mean the robots reached the goal positions. However, after that, the vectors $\mathbf{t}_R$ and $\mathbf{h}_R$ help to move the robots to the goal positions without moving the center of the ladder. □

Consider the instance shown on the left in Fig. 10, where $G+$ would drive both robots straight to their respective goal positions, if there were no control errors. As illustrated on the right, in any physical experiment the robots will inevitably deviate from the intended trajectories for a number of reasons, including sensor and control errors. An advantage of $G+$ is that its future output depends only on the current and goal states, and is independent of the past history. An algorithm is said to be self-stabilizing if it tolerates any finite number of transient faults like sensor and control errors, i.e., the algorithm is correct, even in the presence of any finite number of transient faults.

Corollary 1. *Algorithm $G+$ is self-stabilizing.*

To confirm the robustness of $G+$, we conducted a series of simulations by replacing $\mathbf{T}_R$ in **Step 5** of $G+$ by $\mathbf{T}_R = \mathbf{t}_R + \mathbf{r}_R + \mathbf{h}_R + \mathbf{n}_R$, where $\mathbf{n}_R$ is a random noise vector of size $0.1V$. Since Corollary 1 guarantees that $G+$ eventually transports the ladder to the goal position, our main concern here is the finish time.

Figs. 11 (left) and 11 (right), respectively, show the finish times of the motions by $G+$ in the presence of noise (as well as the results for the noise-free case and

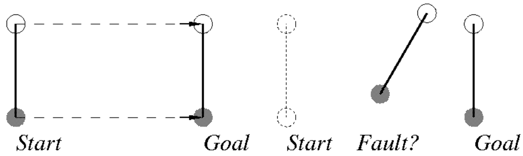


Fig. 10. A simple instance (left), and deviation from the intended path (right)

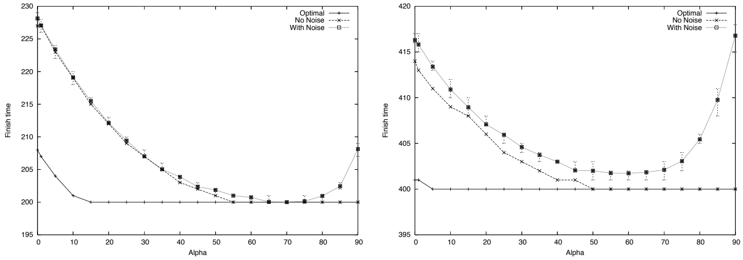


Fig. 11. Finish times of $G+$ with noise for $L_B = 2$ (left) and $L_B = 4$ (right), respectively

time-optimal motions), for $L_B = 2$ and 4. For each value of α, a vertical bar and a small box show, respectively, the range of finish times and their average we observed in 100 runs. (Actual trajectories with noise differ very little visibly from those in the noise-free case, and thus are omitted.)

The delay due to noise, over the noise-free case, can be as large as 4.5% for both the cases $L_B = 2$ and $L_B = 4$ (when $\alpha = 90°$). It is not clear why the delay due to noise increases slightly as α approaches 90°. However, we think that the delay is sufficiently small in all cases and is acceptable in practice. Especially (as Corollary 1 guarantees) the robots never failed to reach the goal positions in our simulation even in the presence of noise.

Finally, we observe that each robot can simulate $G+$ even after we replace the ladder by an imaginary one, and thus we can use $G+$ as a marching algorithm. However, it is obvious from the fact that $G+$ requires as input the current and goal positions of the robots.

5 Algorithm $G+$ for Three and More Robots

In this section we extend Algorithm $G+$ to more than two robots. First, we modify the problem formulation defined in Section 2. Consider for example the case of three robots carrying a regular triangle (equilateral). As in Fig. 2, let A_s, B_s, C_s and A_g, B_g, C_g be the start and goal positions of the robots, respectively (see Fig. 12). Then instances are identified with three parameters L, α, and β, where L is defined as $|A_sA_g|$ and α and β are the angles that $\overline{A_sB_s}$ and $\overline{A_gB_g}$ makes with $\overline{A_sA_g}$, respectively. For simplicity, we restrict our attention to

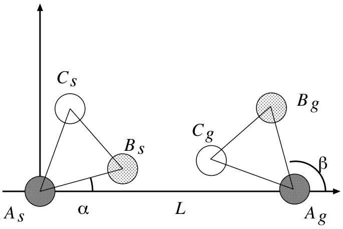


Fig. 12. The setup of the problem with three robots

a setting of the problem, in which $\beta = 180° - \alpha$ $(0° \leq \alpha \leq 90°)$, and $L = 2$ and 4 as before.

Algorithm $G+$ for more than two robots is basically the same as before, except **Step 4**. In the case of two robots, we calculated the offset vector $\mathbf{o}_R$ considering that the ladder is placed exactly in the middle of the two robots. When the number of robots is more than two, we use the method given in pp.752–753 of [27] to this end, and then calculate the offset vector: First, the center of mass of the object is supposed to coincide with the center of mass of the robots' positions. Then, we consider that the total external force to the object produced by the robots is defined as sum of force vector produced by the difference between each robot's current position and a corresponding corner of the object. Then, we determine the position of the object such that the total moment of the external forces is turned to be zero, that is, rotational forces caused by the force vectors are canceled. For comparison purpose, we also extend G in a straight forward way to the case with more than two robots.

In the following, let us observe the performance of $G+$ for more than two robots (up to nine robots) carrying a regular polygon. First we pick up 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9 as the values of the spring constant s, respectively for three to nine robots which were found to be large enough to keep the endpoints of the object inside the circles representing robots' bodies. Then, again we can choose a reasonable pair of values for the parameters t and u such as $t = 1$ and $u = 1$ by simulation (similar to the simulations in Section 4). Note again that these values are not always the best for each instance, but they give reasonable motions: Fig. 13 and Fig. 14 show example motions by G and $G+$ with three robots, for the instances I_1 and I_2. Roughly speaking, like the case of two robots, the motions by G are divided into two parts, translation and rotation, while these two are done simultaneously in the motions by $G+$ that derives smoother motions.

Because of the space limitation, we only show a limited number of detailed simulation results of G and $G+$ with three robots for $L = 2$ and $L = 4$ in Figs. 15 and 16. In addition, for four to nine robots, we only show the figures of motions by G and $G+$ for the instances I_1 and I_2 in Figs. 17 through 25 without charts showing finish times and formation errors. Since there is no previous research on time-optimal motions with more than two robots, the figures

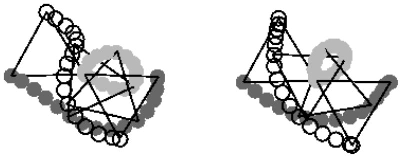


Fig. 13. Motions with three robots by G (left) and $G+$ (right) for instance I_1

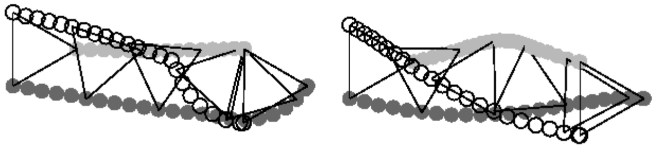


Fig. 14. Motions with three robots by G (left) and $G+$ (right) for instance I_2

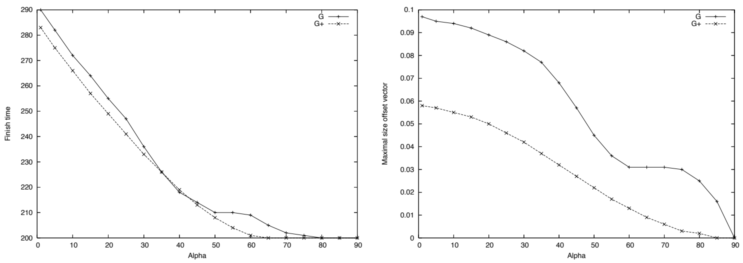


Fig. 15. Finish times (left) and peak offset vector size (right) of G and $G+$ for $L = 2$ and $0^\circ \leq \alpha \leq 90^\circ$ with three robots, respectively

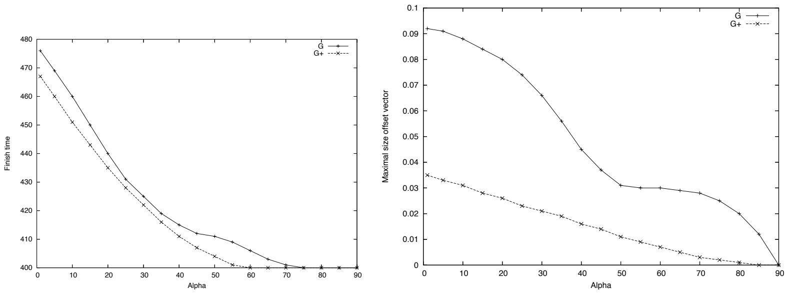


Fig. 16. Finish times (left) and peak offset vector size (right) of G and $G+$ for $L = 4$ and $0^\circ \leq \alpha \leq 90^\circ$ with three robots, respectively

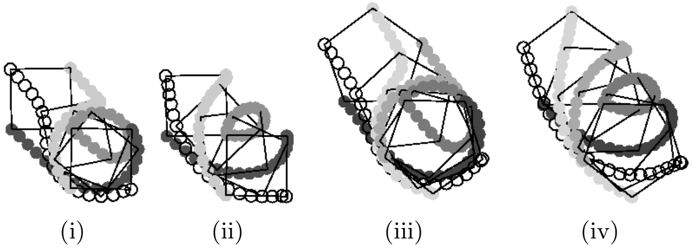


Fig. 17. Motions for instance I_1 with four robots by (i) G and (ii) $G+$, and with five robots by (iii) G and (iv) $G+$, respectively

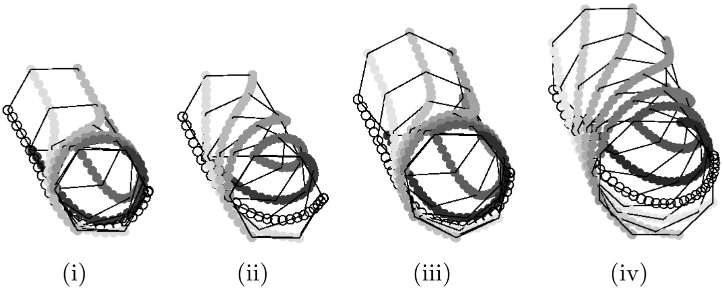


Fig. 18. Motions for instance I_1 with six robots by (i) G and (ii) $G+$, and with seven robots by (iii) G and (iv) $G+$, respectively

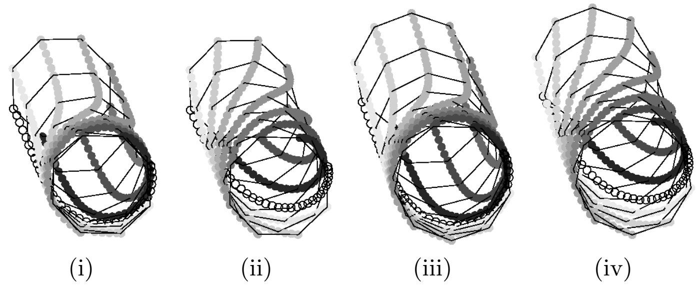


Fig. 19. Motions for instance I_1 with eight robots by (i) G and (ii)$G+$, and with nine robots by (iii) G and (iv) $G+$, respectively

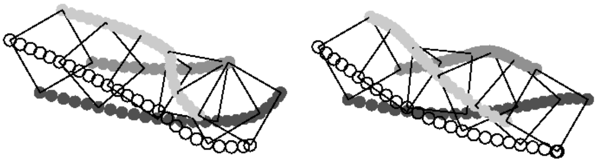


Fig. 20. Motions with four robots by G (left) and $G+$ (right) for instance I_2

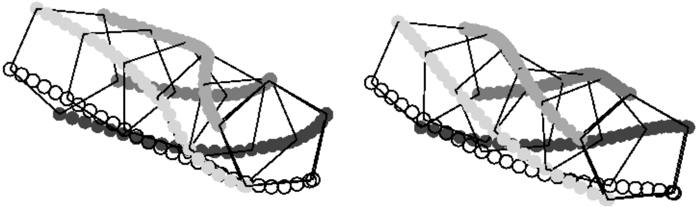


Fig. 21. Motions with five robots by G (left) and $G+$ (right) for instance I_2

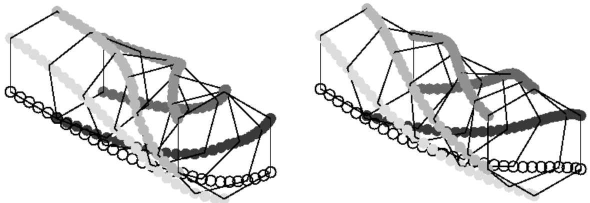


Fig. 22. Motions with six robots by G (left) and $G+$ (right) for instance I_2

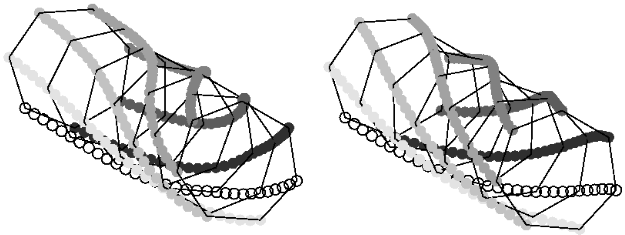


Fig. 23. Motions with seven robots by G (left) and $G+$ (right) for instance I_2

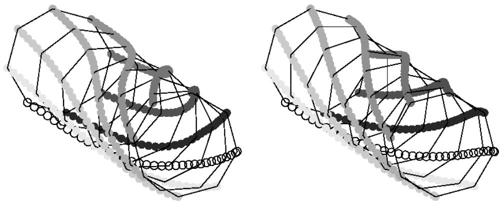


Fig. 24. Motions with eight robots by G (left) and $G+$ (right) for instance I_2

do not show the results for optimal motions. Finish times of G and $G+$ are almost similar, although $G+$ is slightly better (Fig. 15, left). We observed the largest improvement in terms of finish time for the instance I_1 with nine robots: The finish times of G and $G+$ are 835 and 741, respectively, in which the motion by $G+$ is completed 12.2% faster than the motion by G. As for the size of the peak offset vector, the motions by $G+$ are always smoother than those by G

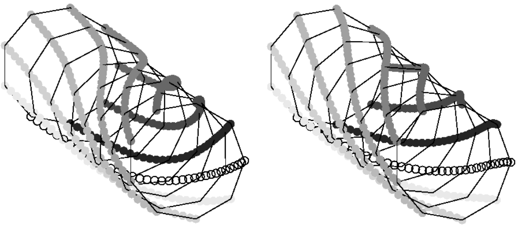


Fig. 25. Motions with nine robots by G (left) and $G+$ (right) for instance I_2

(Fig. 15, right). As an example, for the instance I_1 with nine robots, the size of the peak offset vector by $G+$ is about 63% of that by G; $G+$ improves the smoothness of the motion. In summary, the simulation results indicate that if the number of robots is greater than two, the advantage of algorithm $G+$ over G is in smoother motions rather than smaller finish times. This observation seems to indicate that maintaining a given formation can be a severe constraint when obtaining time-optimal motion with many robots.

We would like to note here that, arguing as in the proof of Theorem 1, we can prove the convergence of Algorithm $G+$, i.e., the robots' positions always converge to their goal positions, for the case of more than two robots in a simple formation such as a regular polygon we examined.

6 Conclusion

In this paper, we have proposed a self-stabilizing marching algorithm in an obstacle-free workplace, where marching means that the robots must move while maintaining a given formation. To this end, for the case of two robots, we have developed an algorithm $G+$ for a group of oblivious robots to transport a ladder, and showed that $G+$ is correct, and self-stabilizing. As promised, $G+$ is simple and constructed from a naive algorithm G that generates only a selfish move, by adding two simple ingredients. Also, the motions obtained by $G+$ are fairly close to the time-optimal motions.

In $G+$, each robot uses the offset vector to figure out (and adjust) its position relative to the object it carries. It is not difficult to extend $G+$ to handle the case in which more than two robots are involved to transport a given object, since the offset vector can be easily calculated from the current positions of the robots. Based on this idea, we have extended $G+$ and demonstrated by computer simulation that $G+$ holds the merits for three or more robots.

We have also observed that $G+$ is indeed used as a marching algorithm for the case of two robots. However, the same observation is obviously possible even for more than two robots, and hence $G+$ can be used as a marching algorithm for more than two robots, as well.

The three parameters s, t, and u used in $G+$ are experimentally determined in this paper based on the limited number of simulations. As for the further deep studies on the algorithm $G+$, if we can develop a method to select their values for every configuration, it must be very useful. The algorithms G and $G+$ are evaluated by delay (finish time) and formation error, which shows that $G+$ is better than G, however it is controversial that how much delay or formation error is accepted for real robots, or whether $G+$ is a best possible algorithm or not. In addition to that, the proof of correctness of $G+$ does not guarantee the maximum finish time or the maximum formation error. The self-stability of $G+$ is demonstrated under an assumption that the robot may move to wrong position because of noise. An alternative interesting situation to be tested is that each robot sometimes misunderstands the others' locations because of sensor errors.

Distributed marching algorithms (i) by robots with different maximum speeds and capabilities, (ii) using many robots, and (iii) in an environment occupied by obstacles, are suggested for future study. Some results on these issues are found in [3,4].

Acknowledgments

This work was partially supported by KAKENHI 18300004 and 18700015.

References

1. Alami, R., Fleury, S., Herrb, M., Ingrand, F., Qutub, S.: Operating a Large Fleet of Mobile Robots Using the Plan-merging Paradigm. In: IEEE Int. Conf. on Robotics and Automation, pp. 2312–2317 (1997)
2. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: A Distributed Memoryless Point Convergence Algorithm for Mobile Robots with Limited Visibility. IEEE Trans. Robotics and Automation 15(5), 818–828 (1999)
3. Asahiro, Y., Chang, E.C., Mali, A., Nagafuji, S., Suzuki, I., Yamashita, M.: Distributed Motion Generation for Two Omni-directional Robots Carrying a ladder. Distributed Autonomous Robotic Systems 4, 427–436 (2000)
4. Asahiro, Y., Chang, E.C., Mali, A., Suzuki, I., Yamashita, M.: A Distributed Ladder Transportation Algorithm for Two Robots in a Corridor. In: IEEE Int. Conf. on Robotics and Automation, pp. 3016–3021 (2001)
5. Asama, H., Sato, M., Bogoni, L., Kaetsu, H., Matsumoto, A., Endo, I.: Development of an Omni-directional Mobile Robot with 3 DOF Decoupling Drive Mechanism. In: IEEE Int. Conf. on Robotics and Automation, pp. 1925–1930 (1995)
6. Balch, T.: Behavior-based Formation Control for Multi-robot Teams. IEEE Trans. Robotics and Automation 14(6), 926–939 (1998)
7. Belta, C., Kumar, V.: Abstraction and Control for Groups of Robots. IEEE Trans. Robotics 20(5), 865–875 (2004)
8. Canepa, D., Gradinariu Potop-Butucaru, M.: Stabilizing Flocking via Leader Election in Robot Networks. In: Int. Symp. Stabilization, Safety, and Security, pp. 52–66 (2007)
9. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the robots gathering problem. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1181–1196. Springer, Heidelberg (2003)

10. Cao, Y.U., Fukunaga, A.S., Kahng, A.B.: Cooperative Mobile Robots: Antecedents and Directions. Autonomous Robots 4, 1–23 (1997)
11. Chen, A., Suzuki, I., Yamashita, M.: Time-optimal Motion of Two Omnidirectional Robots Carrying a Ladder Under a Velocity Constraint. IEEE Trans. Robotics and Automation 13(5), 721–729 (1997)
12. Czyzowicz, J., Gasieniec, L., Pelc, A.: Gathering Few Fat Mobile Robots in the Plane. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 350–364. Springer, Heidelberg (2006)
13. Cohen, R., Peleg, D.: Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. SIAM J. on Computing 34, 1516–1528 (2005)
14. Cohen, R., Peleg, D.: Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements. SIAM J. on Computing 38, 276–302 (2008)
15. Debest, X.A.: Remark about Self-stabilizing Systems. Comm. ACM 38(2), 115–117 (1995)
16. Donald, B.R.: Information Invariants in Robotics: Part I – State, Communication, and Side-effects. In: IEEE Int. Conf. on Robotics and Automation, pp. 276–283 (1993)
17. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots. In: Aggarwal, A.K., Pandu Rangan, C. (eds.) ISAAC 1999. LNCS, vol. 1741, pp. 93–102. Springer, Heidelberg (1999)
18. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Arbitrary Pattern Formation by Asynchronous, Anonymous, Oblivious Robots. Theoretical Computer Science (to appear)
19. Ge, S.S., Lewis, F.L. (eds.): Autonomous Mobile Robots: Sensing, Control Decision Making and Applications. CRC Press, Boca Raton (2006)
20. Gervasi, V., Prencipe, G.: Coordination without Communication: The case of the Flocking Problem. Discrete Applied Mathematics 143, 203–223 (2003)
21. Izumi, T., Katayama, Y., Inuzuka, N., Wada, K.: Gathering Autonomous Mobile Robots with Dynamic Compasses: An Optimal Results. In: Int'l Symp. Distributed Computing, pp. 298–312 (2007)
22. Jadbabaie, A., Lin, J., Morse, A.S.: Coordination of Groups of Mobile Autonomous Agents Using Nearest Neighbor Rules. IEEE Trans. Automatic Control 48(6), 988–1001 (2003)
23. Justh, E.W., Krishnaprasad, P.S.: Equilibria and Steering Laws for Planar Formation. System Control Letters 52(1), 25–38 (2004)
24. Katayama, Y., Tomida, Y., Imazu, H., Inuzuka, N., Wada, K.: Dynamic Compass Models and Gathering Algorithms for Autonomous Mobile Robots. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 274–288. Springer, Heidelberg (2007)
25. Kosuge, K., Oosumi, T.: Decentralized Control of Multiple Robots Handling an Object. In: International Conference on Intelligent Robots and Systems, pp. 318–323 (1996)
26. Martínez, S., Cortés, J., Bullo, F.: Motion Coordination with Distributed Information. IEEE Control Systems Magazine, 75–88 (2007)
27. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
28. Lee, L.F., Krovi, V.: A Standardized Testing-ground for Artificial Potential-field Based Motion Planning for Robot Collectives. In: 2006 Performance Metrics for Intelligent Systems Workshop, pp. 232–239 (2006)

29. Nakamura, Y., Nagai, K., Yoshikawa, T.: Dynamics and Stability in Coordination of Multiple Robotics Mechanisms. Int. J. of Robotics Research 8(2), 44–60 (1989)
30. Olfati-Saber, R.: Flocking for Multi-agent Dynamic Systems: Algorithms and Theory. IEEE Trans. Automatic Control 51(3), 401–420 (2006)
31. Prencipe, G.: CORDA: Distributed Coordination of a Set of Autonomous Mobile Robots. In: ERSADS 2001, pp. 185–190 (2001)
32. Prencipe, G.: On the Feasibility of Gathering by Autonomous Mobile robots. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 246–261. Springer, Heidelberg (2005)
33. Shuneider, F.E., Wildermuth, D., Wolf, H.L.: Motion Coordination in Formations of Multiple Robots Using a Potential Field Approach. Distributed Autonomous Robotic Systems 4, 305–314 (2000)
34. Souissi, S., Défago, X., Yamashita, M.: Gathering Asynchronous Mobile Robots with Inaccurate Compasses. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 333–349. Springer, Heidelberg (2006)
35. Souissi, S., Défago, X., Yamashita, M.: Using Eventually Consistent Compasses to Gather Oblivious Mobile Robots with Limited Visibility. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 471–487. Springer, Heidelberg (2006)
36. Stilwell, D.J., Bay, J.S.: Toward the Development of a Material Transport System Using Swarms of Ant-like Robots. In: IEEE Int. Conf. on Robotics and Automation, pp. 766–771 (1995)
37. Sugihara, K., Suzuki, I.: Distributed Motion Coordination of Multiple Mobile Robots. In: IEEE Int. Symp. on Intelligent Control, pp. 138–143 (1990)
38. Sugihara, K., Suzuki, I.: Distributed Algorithms for Formation of Geometric Patterns with Many Mobile Robots. Journal of Robotic Systems 13(3), 127–139 (1996)
39. Suzuki, I., Yamashita, M.: Formation and Agreement Problems for Anonymous Mobile Robots. In: Annual Allerton Conference on Communication, Control, and Computing, pp. 93–102 (1993)
40. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM J. Computing 28(4), 1347–1363 (1999)
41. Tanner, H., Jadbabaie, A., Pappas, G.J.: Flocking in Fixed and Switching Networks. IEEE Trans. Automatic Control 52(5), 863–868 (2007)
42. Whitcomb, L.L., Koditschek, D.E., Cabrera, J.B.D.: Toward the Automatic Control of Robot Assembly Tasks via Potential Functions: The Case of 2-D Sphere Assemblies. In: IEEE Int. Conf. on Robotics and Automation, pp. 2186–2191 (1992)
43. Yamaguchi, H.: A Distributed Motion Coordination Strategy for Multiple Nonholomic Mobile Robots in Cooperative Hunting Operations. Robotics and Autonomous Systems 43(4), 257–282 (2003)

Fault-Tolerant Flocking in a k-Bounded Asynchronous System*

Samia Souissi[1,2], Yan Yang[1], and Xavier Défago[1]

[1] School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{ssouissi,y.yang,defago}@jaist.ac.jp
[2] Now at Nagoya Institute of Technology,
Department of Computer Science and Engineering
souissi.samia@nitech.ac.jp

Abstract. This paper studies the flocking problem, where mobile robots group to form a desired pattern and move together while maintaining that formation. Unlike previous studies of the problem, we consider a system of mobile robots in which a number of them may possibly fail by crashing. Our algorithm ensures that the crash of faulty robots does not bring the formation to a permanent stop, and that the correct robots are thus eventually allowed to reorganize and continue moving together. Furthermore, the algorithm makes no assumption on the relative speeds at which the robots can move.

The algorithm relies on the assumption that robots' activations follow a k-bounded asynchronous scheduler, in the sense that the beginning and end of activations are not synchronized across robots (asynchronous), and that while the slowest robot is activated once, the fastest robot is activated at most k times (k-bounded).

The proposed algorithm is made of three parts. First, appropriate restrictions on the movements of the robots make it possible to agree on a common ranking of the robots. Second, based on the ranking and the k-bounded scheduler, robots can eventually detect any robot that has crashed, and thus trigger a reorganization of the robots. Finally, the third part of the algorithm ensures that the robots move together while keeping an approximation of a regular polygon, while also ensuring the necessary restrictions on their movement.

1 Introduction

Be it on earth, in space, or on other planets, robots and other kinds of automatic systems provide essential support in otherwise adverse and hazardous environments. For instance, among many other applications, it is becoming increasingly attractive to consider a group of mobile robots as a way to provide support for rescue and relief during or after a natural catastrophe (e.g., earthquake, tsunami, cyclone, volcano eruption). As a result, research on mechanisms for coordination and self-organization of mobile robot systems is beginning to attract considerable attention (e.g, [17,19,20,21]). For

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

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 145–163, 2008.

such operations, relying on a group of simple robots for delicate operations has various advantages over considering a single complex robot. For instance, (1) it is usually more cost-effective to manufacture and deploy a number of cheap robots rather than a single expensive one, (2) higher number yields better potential for a system resilient to individual robot failures, (3) smaller robots have obviously better mobility in tight and confined spaces, and (4) a group can survey a larger area than an individual robot, even if the latter is equipped with better sensors.

Nevertheless, merely bringing robots together is by no means sufficient, and adequate coordination mechanisms must be designed to ensure coherent group behavior. Furthermore, since many applications of cooperative robotics consider cheap robots dwelling in hazardous environments, fault-tolerance is of primary concern.

The problem of reaching agreement among a group of autonomous mobile robots has attracted considerable attention over the last few years. While much formal work focuses on the gathering problem (robots must meet at a point, e.g., [7]) as the embodiment of a *static* notion of agreement, this work studies the problem of flocking (robots must move together), which embodies a *dynamic* notion of agreement, as well as coordination and synchronization. The flocking problem has been studied from various perspectives. Studies can be found in different disciplines, from artificial intelligence to engineering [1,3,5,6]. However, only few works considered the presence of faulty robots [2,4].

Fault-tolerant flocking. Briefly, the main problem studied in this paper, namely the flocking problem, requires that a group of robots move together, staying close to each other, and keeping some desired formation while moving. Numerous definitions of flocking can be found in the literature [3,11,12,14], but few of them define the problem precisely. The rare rigorous definitions of the problem suppose the existence of a leader robot and require that the other robots, called followers, follow the leader in a desired fashion [3,6,10], such as by maintaining an approximation of a regular polygon.

The variant of the problem that we consider in this paper requires that the robots form and move while maintaining an approximation of a regular polygon, in spite of the possible presence of faulty robots—robots may fail by crashing and a crash is permanent. Although we do consider the presence of a leader robot to lead the group, the role of leader is assigned *dynamically* and any of the robots can potentially become a leader. In particular, after the crash of a leader, a new leader must eventually take over that role.

Model. The system is modelled as a system composed of a group of autonomous mobile robots, modelled as points evolving on the plane, and all of which execute the same algorithm independently. Some of the robots may possibly fail by crashing, after which they do not move forever. Although the robots share no common origin, they do share one common direction (as given by a compass), a common unit distance, and the same notion of clockwise direction.

Robots repeatedly go through a succession of activation cycles during which they observe their environment, compute a destination and move. Robots are asynchronous in that one robot may begin an activation cycle while another robot finishes one. While some robots may be activated more often than others, we assume that the scheduler is k-bounded in the sense that, in the interval it takes any correct robot to perform a single activation cycle, no other robot performs more than k activations. The robots can remember only a *limited* number of their past activations.

Contribution. The paper presents a fault-tolerant flocking algorithm for a k-bounded asynchronous robot system. The algorithm is decomposed into three parts. In the first part, the algorithm relies on the k-bounded scheduler to ensure failure detection. In the second part, the algorithm establishes a ranking system for the robots and then ensures that robots agree on the same ranking throughout activations. In the third and last part, the ranking and the failure detector are combined to realize the flocking of the robots by maintaining an approximation of a regular polygon while moving.

Related work. Gervasi and Prencipe [3] have proposed a flocking algorithm for robots based on a leader-followers model, but introduce additional assumptions on the speed of the robots. In particular, they proposed a flocking algorithm for formations that are symmetric with respect to the leader's movement, without agreement on a common coordinate system (except for the unit distance). However, their algorithm requires that the leader is distinguished from the robots followers.

Canepa and Potop-Butucaru [6] proposed a flocking algorithm in an asynchronous system with oblivious robots. First, the robots elect a leader using a probabilistic algorithm. After that, the robots position themselves according to a specific formation. Finally, the formation moves ahead. Their algorithm only lets the formation move straight forward. Although the leader is determined dynamically, once elected it can no longer change. In the absence of faulty robots, this is a reasonable limitation in their model.

To the best of our knowledge, our work is the first to consider flocking of asynchronous (k-bounded) robots in the presence of faulty robots. Also, we want to stress that the above two algorithms do not work properly in the presence of faulty robots, and that their adaptation is not straightforward.

Structure. The remainder of this paper is organized as follows. In Section 2, we present the system model. In Section 3, we define the problem. In Section 4, we propose a failure detection algorithm based on $k-$bounded scheduler. In Section 5, we give an algorithm that provides a ranking mechanism for robots. In Section 6, we propose a dynamic fault tolerant flocking algorithm that maintains an approximation of a regular polygon. Finally, in Section 7, we conclude the paper.

2 System Model and Definitions

2.1 The CORDA Model

In this paper, we consider the CORDA model of Prencipe [8] with k-bounded scheduler. The system consists of a set of autonomous mobile robots $\mathcal{R} = \{r_1, \cdots, r_n\}$. A robot is modelled as a unit having computational capabilities, and which can move freely in the two-dimensional plane. Robots are seen as points on the plane. In addition, they are equipped with sensor capabilities to observe the positions of the other robots, and form a local view of the world.

The local view of each robot includes a unit of length, an origin, and the directions and orientations of the two x and y coordinate axes. In particular, we assume that robots have a partial agreement on the local coordinate system. Specifically, they agree on the orientation and direction of one axis, say y. Also, they agree on the clockwise/counterclokwise direction.

The robots are completely *autonomous*. Moreover, they are *anonymous*, in the sense that they are a priori indistinguishable by appearance. Furthermore, there is no direct means of communication among them.

In the CORDA model, robots are totally *asynchronous*. The cycle of a robot consists of a sequence of events: Wait-Look-Compute-Move.

- *Wait*. A robot is idle. A robot cannot stay permanently idle. At the beginning all robots are in Wait state.
- *Look*. Here, a robot *observes* the world by activating its sensors, which will return a snapshot of the positions of the robots in the system.
- *Compute*. In this event, a robot *performs* a local computation according to its deterministic algorithm. The algorithm is the same for all robots, and the result of the *compute* state is a destination point.
- *Move*. The robot *moves* toward its computed destination. But, the distance it moves is unmeasured; neither infinite, nor infinitesimally small. Hence, the robot can only go towards its goal, but the move can end anywhere before the destination.

In the model, there are two limiting assumptions related to the cycle of a robot.

Assumption 1. *It is assumed that the distance travelled by a robot r in a move is not infinite. Furthermore, it is not infinitesimally small: there exists a constant $\delta_r > 0$, such that, if the target point is closer than δ_r, r will reach it; otherwise, r will move toward it by at least δ_r.*

Assumption 2. *The amount of time required by a robot r to complete a cycle (wait-look-compute-move) is not infinite. Furthermore, it is not infinitesimally small; there exists a constant $\tau_r > 0$, such that the cycle will require at least τ_r time.*

2.2 Assumptions

k-bounded-scheduler. In this paper, we assume the CORDA model with k-bounded scheduler, in order to ensure some fairness of activations among robots. Before we define the k-bounded-scheduler, we give a definition of *full activation cycle* for robots.

Definition 1 (full activation cycle). *A* **full activation cycle** *for any robot r_i is defined as the interval from the event Look (included) to the next instance of the same event Look (excluded).*

Definition 2 (k-bounded-scheduler). *With a* k**-bounded scheduler**, *between two consecutive full activation cycles of the same robot r_i, another robot r_j can execute at most k full activation cycles.*

This allows us to establish the following lemma:

Lemma 1. *If a robot r_i is activated $k+1$ times, then all (correct) robots have completed at least one full activation cycle during the same interval.*

Faults. In this paper, we address ***crash failures***. That is, we consider *initial* crash of robots and also the crash of robots *during execution*. That is, a robot may fail by crashing, after which it executes no actions (no movement). A crash is *permanent* in the sense that a faulty robot never recovers. However, it is still physically present in the system, and it is seen by the other non-crashed robots. A robot that is not faulty is called a *correct* robot.

Before we proceed, we give the following notations that will be used throughout this paper. We denote by $\mathcal{R} = \{r_1, \cdots, r_n\}$ the set of all the robots in the system. Given some robot r_i, $r_i(t)$ is the position of r_i at time t. $y(r_i)$ denotes the y coordinate of robot r_i at some time t. Let A and B be two points, with $\overline{AB}$, we will indicate the segment starting at A and terminating at B, and $dist(A, B)$ is the length of such a segment. Given a region $\mathcal{X}$, we denote by $|\mathcal{X}|$, the number of robots in that region at time t. Finally, let S be a set of robots, then $|S|$ indicates the number of robots in S.

3 Problem Definition

Definition 3 (Formation). *A formation $F = Formation(P_1, P_2, ..., P_n)$ is a configuration, with P_1 the leader of the formation, and the remaining points, the followers of the formation. The leader P_1 is not distinct physically from the robot followers.*

In this paper, we assume that the formation F is a regular polygon. We denote by d the length of the polygon edge (known to the robots), and by $\alpha = (n-2)180°/n$ the angle of the polygon, where n is the number of robots in F.

Definition 4 (Approximate Formation). *We say that robots form an approximation of the formation F if each robot r_i is within ϵ_r from its target P_i in F.*

Definition 5 (The Flocking Problem). *Let r_1,...,r_n be a group of robots, whose positions constitute a formation $F = Formation(P_1, P_2, ..., P_n)$. The robots solve the* **Approximate Flocking Problem** *if, starting from any arbitrary formation at time t_0, $\exists t_1 \geq t_0$ such that, $\forall t \geq t_1$ all robots are at a distance of at most ϵ_r from their respective targets P_i in F, and ϵ_r is a small positive value known to all robots.*

4 Perfect Failure Detection

In this section, we give a simple perfect failure detection algorithm for robots based on a $k-$bounded scheduler in the asynchronous model CORDA. The concept of failure detectors was first introduced by Chandra and Toueg [16] in asynchronous systems with crash faults. A perfect failure detector has two properties: *strong completeness*, and *strong accuracy*. Before we proceed to the description of the algorithm, we make the following assumption, which is necessary for the failure detector mechanism to identify *correct* robots and *crashed* ones.

Assumption 3. *We assume that, at each activation of some robot r_i (correct), r_i computes as destination a position that is different from its current position. Also, a robot*

r_i never visits the same location for the last $k+1$ activations of r_i.[1] *Finally, a robot r_i never visits a location that was visited by any other robot r_j during the last $k+1$ activations of r_j.*

Recall that we only consider *permanent* crash failures of robots, and that crashed robots remain physically in the system. Besides, robots are anonymous. Therefore, the problem is how to distinguish faulty robots from correct ones. Algorithm 1 provides a simple perfect failure detection mechanism for the identification of correct robots. The algorithm is based on the fact that a correct robot must change its current position whenever it is activated (Assumption 3), and also relies on the definition of the $k-$bounded scheduler for the activations of robots. So, a robot r_i considers that some robot r_j is faulty if r_i is activated $k+1$ times, while robot r_j is still in the same position. Algorithm 1 gives as output the set of positions of correct robots $S_{correct}$, and uses the following *variables*:

- $S_{PosPrevObser}$: a global variable representing the set of points of the positions of robots in the system in the previous activation of some robot r_i. These points include the positions of correct and faulty robots. $S_{PosPrevObser}$ is initialized to the empty set during the first activation of robot r_i.
- $S_{PosCurrObser}$: the set of points representing the positions of robots (including faulty ones) in the current activation of some robot r_i.
- c_j: a global variable recording how many times robot r_j did not change its position.

Algorithm 1. Perfect Failure Detection (code executed by robot r_i)

```
Initialization: $S_{PosPrevObser} := \emptyset$; $c_j := 0$
procedure Failure_Detection($S_{PosPrevObser}$,$S_{PosCurrObser}$)
  $S_{correct} := S_{PosCurrObser}$;
  for $\forall\, p_j \in S_{PosCurrObser}$ do
    if ($p_j \in S_{PosPrevObser}$) then                {robot $r_j$ has not moved}
      $c_j := c_j + 1$;
    else
      $c_j := 0$;
    end if
    if ($c_j \geq k$) then
      $S_{correct} = S_{correct} - \{p_j\}$;
    end if
  end for
  return ($S_{correct}$)
end
```

The proposed failure detection algorithm (Algorithm 1) satisfies the two properties of a perfect failure detector: *strong completeness*, and *strong accuracy*. It also satisfies the *eventual agreement* property. These properties are stated respectively in Theorem 1, Theorem 2, and Theorem 3, and their proofs are straightforward (details are in corresponding research report [18]).

[1] That is, r_i never revisits a point location that was within its line of movement for its last $k+1$ total activations.

Theorem 1. ***Strong completeness****: eventually every robot that crashes is permanently suspected by every correct robot.*

Theorem 2. ***Strong accuracy****: there is a finite time after which correct robots are not suspected by any other correct robots.*

Theorem 3. ***Eventual agreement****: there is a finite time after which, all correct robots agree on the same set of correct robots in the system.*

5 Agreed Ranking for Robots

In this section, we provide an algorithm that gives a unique ranking (or identification) to every robot in the system since we assume that robots are anonymous, and do not have any identifier to allow them to distinguish each other. The algorithm allows correct robots to compute and agree on the same ranking. In particular, the ranking mechanism is needed for the election of the leader of the formation. Recall that, a deterministic leader election is impossible without a shared y-axis [9]. Therefore, we assume that robots agree on the y-axis.

We first assume that robots are not located initially at the same point. That is, robots are not in the gathering configuration [7], because it may become impossible to separate them later.[2] The ranking assignment is given in Algorithm 2, which takes as input the set of positions of correct robots in the system $S_{correct}$, and returns as output an ordered set of the positions in $S_{correct}$, called $RankSequence$. The ranking of positions of robots in $S_{correct}$ gives to every robot a unique identification number. The computation of $RankSequence$ is done as follows: $RankSequence = \{S_{correct}, <\}$, where the relation " $<$ "is defined by comparing the y coordinates of the points in $S_{correct}$, and breaking ties from left to right. In other words, the positions of robots in $S_{correct}$ are sorted by decreasing order of $y-$coordinate, such that the robot with greatest y-coordinate is the first in $RankSequence$. When two or more robots share the same y-coordinate, the clockwise direction is used to determine the sequence; a robot r_i that has a robot r_j on its right hand, has a lower rank than r_j in $RankSequence$.

In order for robots to agree on the same $RankSequence$ initially, some restrictions on their movement are required during their first k activations. The movement restriction is given by procedure Lateral_Move_Right(), and it is designed in such a way that all robots compute the same $RankSequence$ during their first k activations. In particular, a robot r_i that does not have robots on $Right(r_i)$ can move by at most the distance $\epsilon_r/(k+1)(k+2)$ along $Right(r_i)$ in order to preserve the same $y-$coordinate. Otherwise, r_i moves by $min(\epsilon_r/(k+1)(k+2), dist(r_i,p)/(k+1)(k+2))$ along $Right(r_i)$, where p is the position of the nearest robot to r_i in $Right(r_i)$. [3] From Algorithm 2, we

[2] Consider two robots that happen to have the same coordinate system and that are always activated together. It is impossible to separate them deterministically. In contrast, it would be trivial to scatter them at distinct positions using randomization (e.g., [15]), but this is ruled out in our model.

[3] Note that, the bounded distance $min(\epsilon_r/(k+1)(k+2), dist(r_i,p)/(k+1)(k+2))$ set on the movement of robots is conservative, and is sufficient to avoid collisions between robots, and to satisfy Assumption 3.

Algorithm 2. Ranking_Correct_Robots (code executed by robot r_i)

```
Input: $S_{correct}$: set of positions of correct robots;
Output: $RankSequence$: Ordered set of positions of correct robots $S_{correct}$;
Initialization: $counter_{act}$ := a global variable recording the number of activations of $r_i$;
procedure Ranking_Correct_Robots($S_{correct}$)
  When $r_i$ is activated
  $counter_{act} := counter_{act} + 1$;
  $Left(r_i)$:= is the ray starting at $r_i$ and perpendicular to its $y-$axis in counter-clockwise direction.
  Sort the $y-$coordinates of robots in $S_{correct}$ in decreasing order.
  if $(\forall r_j, r_k \in S_{correct}, y(r_j) \neq y(r_k))$ then
    $RankSequence :=$ the set $S_{correct}$ in order of decreasing $y-$coordinate;
  else if $y(r_j) = y(r_k)$ then
    if ($r_j$ is on $Left(r_k)$) then
      $RankSequence := r_j < r_k$;
    else
      $RankSequence := r_k < r_j$;
    end if
  end if
  if ($counter_{act} \leq k$) then
    Lateral_Move_Right();
  end if
  Return($RankSequence$);
end
```

Algorithm 3. Procedure Lateral Move Right (code executed by robot r_i).

```
procedure Lateral_Move_Right()
  $Right(r_i)$ := the ray starting at $r_i$ and perpendicular to its $y-$axis in clockwise direction;

  if (If no other robot on $Right(r_i)$) then
    $r_i$ moves by at most $\epsilon_r/(k+1)(k+2)$ to $Right(r_i)$;
  else                                  {some robots are in $Right(r_i)$ including faulty robots}
    $p$ := the position of the nearest robot to $r_i$ in $Right(r_i)$;
    $r_i$ moves by $min(\epsilon_r/(k+1)(k+2), dist(r_i, p)/(k+1)(k+2))$ to $Right(r_i)$;
  end if
end
```

derive the following lemmas. In particular, the algorithm gives a unique ranking to every robot in the system, and also ensures no collisions between robots.

Lemma 2. *Algorithm 2 gives a unique ranking to every correct robot in the system.*

Lemma 3. *By Algorithm 2, there is a finite time after which, all correct robots agree on the same initial sequence of ranking, $RankSequence$.*

Lemma 4. *Algorithm 2 guarantees no collisions between the robots in the system.*

The proofs of the above lemmas are simple (details can be found in corresponding research report [18]).

6 Dynamic Fault-Tolerant Flocking

In this section, we propose a dynamic fault tolerant flocking algorithm, where a group of robots can dynamically generate an approximation of a regular polygon (Definition 4), and maintain it while moving. Our flocking algorithm relies on the existence of two devices, namely a *perfect failure detector* device and a *ranking* device, which were represented respectively in Algorithm 1, and Algorithm 2.

6.1 Algorithm Description

The flocking algorithm is depicted in Algorithm 4, and takes as *input* the length of the polygon edge d, and the *history* of robot r_i, which includes the following variables:

- $S_{PosPrevObser}$: the set of positions of robots in the system during the last previous observation of robot r_i.
- $HistoryMove$: the set of points on the plane visited by robot r_i during its last previous $k+1$ activations.
- nbr_{act}: a counter recording the last previous $k+1$ activations of robot r_i.

The overall idea of the algorithm is as follows. First, when robot r_i gets activated, it executes the following steps:

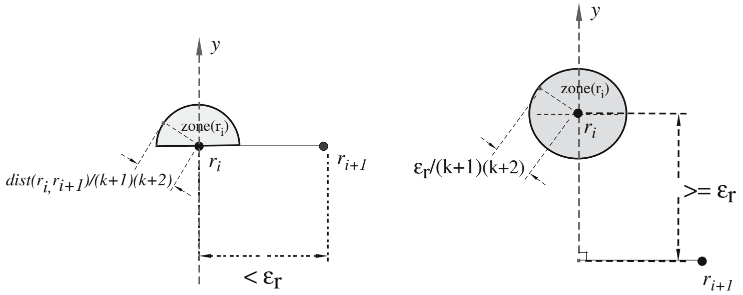
1. Robot r_i takes a snapshot of the current positions $S_{PosCurrObser}$ of robots in the system.
2. Robot r_i calls the failure detection module to get the set of correct robots, $S_{correct}$.
3. Robot r_i calls the ranking module, and gets a total ordering on the set of correct robots $S_{correct}$, called $RankSequence$.
4. Depending on the rank of robot r_i in $RankSequence$, r_i executes the procedure described in Algorithm 5; $\mathsf{Flocking\_Leader(RankSequence, d, nbr_{act}, HistoryMove)}$ if it has the first rank in $RankSequence$ (i.e., the leader). Otherwise, robot r_i is a follower, and it executes the procedure which is described in Algorithm 6, $\mathsf{Flocking\_Follower(RankSequence, d, nbr_{act}, HistoryMove)}$.
5. Robot r_i is a *leader*. First, r_i computes the points of the formation $P_1, ..., P_n$ as in Definition 4, with its location as the first point P_1 in the formation. The targets of the followers are the other points of the formation, and they are assigned to them based on their order in the $RankSequence$. After that, the leader will initiate the movement of the formation, while preserving the same rank sequence, keeping an approximation of the regular polygon, and also avoiding collisions with followers. In order to prevent collisions between robots, the algorithm must guarantee that no two robots ever move to the same location. Therefore, the algorithm defines a movement zone for each robot, within which the robot must move. The zone of the leader, referred to as $Zone(r_i)$, is defined depending on the position of the next robot r_{i+1} in $RankSequence$. Let us denote by $proj_{r_{i+1}}$, the projection of robot r_{i+1} on the $y-$axis of r_i. The movement zone of the leader is defined as follows:
   - r_i and r_{i+1} have the same y coordinate: $Zone(r_i)$ is the half circle with radius $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$, centered at r_i and above r_i (refer to Fig. 1(a)).

Algorithm 4. Dynamic Fault-tolerant Flocking (code executed by robot r_i)

```
Input: Memory($r_i$):$S_{PosPrevObser}$;$HistoryMove$;$nbr_{act}$;
$d$ := the desired distance of the polygon edge;
When $r_i$ is activated
$r_i$ takes a snapshot of the positions $S_{PosCurrObser}$ of robots;
$S_{correct}$ := Failure_Detection($\mathsf{S_{PosPrevObser}}$, $\mathsf{S_{PosCurrObser}}$);
$RankSequence$ := Ranking_Correct_Robots($\mathsf{S_{correct}}$);
$leader$ := first robot in $RankSequence$;
if ($r_i = leader$) then                                        {leader}
    Flocking_Leader(RankSequence, d, $\mathsf{nbr_{act}}$, HistoryMove);
else                                                            {follower}
    Flocking_Follower(RankSequence, d, $\mathsf{nbr_{act}}$, HistoryMove);
end if
```

Algorithm 5. Flocking Leader: Code executed by a robot leader r_i.

```
procedure Flocking_Leader($RankSequence$,$d$,$nbr_{act}$, $HistoryMove$)
    $n := |RankSequence|$;
    $\alpha := (n-2)180°/n$;
    $P$ := Formation($P_1, P_2, ..., P_n$) as in Definition 3;
    $P_1$ := current position of the leader $r_i$;
    $r_{i+1}$:= next robot to $r_i$ in $RankSequence$;
    $proj_{r_{i+1}}$:= the projection of $r_{i+1}$ on $y-$axis of $r_i$;
    if ($proj_{r_{i+1}} = r_i$) then                  {$r_i$ has same $y-$coordinate as $r_{i+1}$}
        $Zone(r_i)$:= half circle with radius $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$, centered at $r_i$ and above $r_i$ (refer to Fig. 1(a));
    else
        $Zone(r_i)$:= the circle centered at $r_i$, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (refer to Fig. 1(b));
    end if
    $S_{CrashInZone}$ := the set of positions of crashed robots in $Zone(r_i)$;
    if ($S_{CrashInZone} \neq \emptyset$) then
        $r_i$ moves to a desired point $Target(r_i)$ within $Zone(r_i)$, excluding the points in $S_{CrashInZone}$, and the points in $HistoryMove$;
    else
        $r_i$ moves to a desired point $Target(r_i)$ within $Zone(r_i)$, excluding the points in $HistoryMove$;
    end if
    $CurrMove$ := the set of points on the segment $\overline{r_i Target(r_i)}$;
    if ($nbr_{act} \leq k+1$) then
        $HistoryMove := HistoryMove \cup CurrMove$;
    else
        $HistoryMove := CurrMove$;
        $nbr_{act} := 1$;
    end if
end
```



(a) r_i and r_{i+1} have the same y-coordinate, and $dist(r_i, r_{i+1}) < \epsilon_r$: $Zone(r_i)$ is the half circle with radius $dist(r_i, r_{i+1})/(k+1)(k+2)$.

(b) r_i and r_{i+1} do not have the same y-coordinate, and $dist(r_i, proj_{r_{i+1}}) \geq \epsilon_r$: $Zone(r_i)$ is the circle with radius $\epsilon_r/(k+1)(k+2)$.

Fig. 1. Zone of movement of the leader

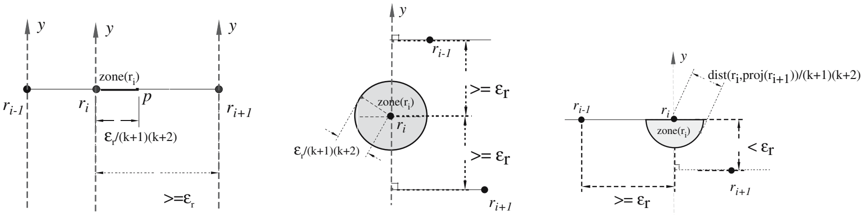
- r_i and r_{i+1} do not have the same y coordinate: $Zone(r_i)$ is the circle, centered at r_i, and with radius $min(dist(r_i, proj_{r_{i+1}})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$ (refer to Fig. 1(b)).

After determining its zone of movement $Zone(r_i)$, robot r_i needs to determine if there are crashed robots within $Zone(r_i)$. If no crashed robots are within its zone, then robot r_i can move to any desired target within $Zone(r_i)$, satisfying Assumption 3. Otherwise, robot r_i can move within $Zone(r_i)$ by excluding the positions of crashed robots, and satisfying Assumption 3.

6. Robot r_i is a *follower*. First, r_i assigns the points of the formation $P_1, ..., P_n$ to the robots in $RankSequence$ based on their order in $RankSequence$. Subsequently, robot r_i determines its target P_i based on the current position of the leader (P_1), and the polygon angle α given in the following equation: $\alpha = (n-2)180°/n$, where n is the number of robots in the formation.

   In order to ensure no collisions between robots, the algorithm also defines a movement zone for each robot follower. The zone of a follower, referred to as $Zone(r_i)$ is defined depending on the position of the previous robot r_{i-1} and the next robot r_{i+1} to r_i in $RankSequence$. Before we proceed, we denote by $proj_{r_{i-1}}$, the projection of robot r_{i-1} on the $y-$axis of robot r_i. Similarly, we denote by $proj_{r_{i+1}}$, the projection of robot r_{i+1} on the $y-$axis of r_i. The zone of movement of a robot follower r_i is defined as follows:

   - r_i, r_{i-1} and r_{i+1} have the same y coordinate, then $Zone(r_i)$ is the segment $\overline{r_i p}$, with p as the point at distance $min(dist(r_i, r_{i+1})/(k+1)(k+2), \epsilon_r/(k+1)(k+2))$ from r_i (Fig. 2(a)).
   - r_i, r_{i-1} and r_{i+1} do not have the same y coordinate, then $Zone(r_i)$ is the circle centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (Fig. 2(b)).
   - r_i and r_{i+1} have the same y coordinate, however r_{i-1} does not, then $Zone(r_i)$ is the half circle above it, centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$.



(a) Aligned. (b) Not aligned. (c) Partly aligned.

Fig. 2. Zone of movement of a follower. There are three cases as follows. The situation (a) in which r_{i-1}, r_i, and r_{i+1} have the same y coordinate. The situation (b) where r_{i-1}, r_i and r_{i+1} do not have the same y coordinate, and $dist(r_i, proj_{r_{i-1}}) \geq \epsilon_r$, and $dist(r_i, proj_{r_{i+1}}) \geq \epsilon_r$. The situation (c) where r_{i-1} and r_i have the same y coordinate, however, r_{i+1} does not. Also, $dist(r_i, r_{i-1}) \geq \epsilon_r$, and $dist(r_i, proj_{r_{i+1}}) < \epsilon_r$.

- r_i and r_{i-1} have the same y coordinate, however r_{i+1} does not, then $Zone(r_i)$ is the half circle below it, centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (Fig. 2(c)).

As we mentioned before, the bound $min(\epsilon_r/(k+1)(k+2), dist(r_i, p)/(k+1)(k+2))$ set on the movement of robots is conservative, and is sufficient to avoid collisions between robots, and to satisfy Assumption 3 (this will be proved later).

For the sake of clarity, we do not describe explicitly in Algorithm 6 the zone of movement of the last robot in the rank sequence. The computation of its zone of movement is similar to that of the other robot followers, with the only difference being that it does not have a next neighbor r_{i+1}. So, if robot r_i has the same $y-$coordinate as its previous neighbor r_{i-1}, then its zone of movement is the half circle with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2))$, centered at r_i and below r_i. Otherwise, the circle centered at r_i, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2))$.

After determining its zone of movement $Zone(r_i)$, robot r_i needs to determine if it can progress toward its target $Target(r_i)$. Note that, $Target(r_i)$ may not necessarily belong to $Zone(r_i)$. To do so, robot r_i computes the intersection of the segment $\overline{r_i Target(r_i)}$ and $Zone(r_i)$, called $Intersect$. If $Intersect$ is equal to the position of r_i, then r_i will move toward its right as given by the procedure Lateral_Move_Right(). Otherwise, r_i moves along the segment $Intersect$ as much as possible, while avoiding to reach the location of a crashed robot in $Intersect$, if any, and satisfying Assumption 3. In any case, if r_i is not able to move to any point in $Intersect$, except its current position, it moves to its right as in the procedure Lateral_Move_Right().

Note that, by the algorithm robot followers can move in any direction by adaptation of their target positions with respect to the new position of the leader. When the leader is idle, robot followers move within the distance $\epsilon_r/(k+1)(k+2)$ or smaller in order to keep an approximation of the formation with respect to the position of the leader, and preserve the rank sequence.

6.2 Correctness of the Algorithm

In this section, we prove the correctness of our flocking algorithm by first showing that correct robots agree on the same ranking during the execution of Algorithm 4 (Theorem 4). Second, we prove that no two correct robots ever move to the same location, and that a correct robot never moves to a location occupied by a faulty robot (Theorem 5). Then, we show that all correct robots dynamically form an approximation of a regular polygon in finite time, and keep this formation while moving (Theorem 6). Finally, we prove that our algorithm tolerates permanent failures of robots (Theorem 7).

Lemma 5. *Algorithm 4 satisfies Assumption 3.*

Proof (Lemma 5). To prove the lemma, we first show that any robot r_i in the system is able to move to a destination that is different from its current location, and robot r_i never visits a point location that was within its line of movement for its last $k+1$ activations. Then, we show that a robot r_i never visits a location that was visited by another robot r_j during the last $k+1$ activations of r_j.

First, assume that robot r_i is the leader. By Algorithm 4, its zone of movement $Zone(r_i)$ is either a circle or a half circle on the plane, excluding the points in its history of moves $HistoryMove$ for the last $k+1$ activations, and the positions of crashed robots. Since, $Zone(r_i)$ is composed of an infinite number of points, the positions of crashed robots are finite, and $HistoryMove$ is a strict subset of $Zone(r_i)$, then robot r_i can always compute and move to a new location that is different from the locations visited by r_i during its last $k+1$ activations.

Now, assume that robot r_i is a follower, and let r_{i-1} and r_{i+1}, be respectively the previous, and next robots to r_i in $RankSequence$. Two cases follow depending on the zone of movement of r_i.

- Consider the case where $Zone(r_i)$ is the *segment* with length $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$, excluding r_i. Since, such case occurs only when r_{i-1}, r_i, and r_{i+1} have the same y coordinate, and robot r_i is only allowed to move to $Right(r_i)$. Then, r_i can always move to a free position in $Right(r_i)$ that does not belong to $HistoryMove$, and that excludes the positions of crashed robots since they are finite and there exists an infinite number of points in $Zone(r_i)$.
- Consider the case where $Zone(r_i)$ is either a circle or a half circle, centered at r_i and with a radius greater than zero, excluding its history of move $HistoryMove$ for the last $k+1$ activations, and the positions of crashed robots. By similar arguments as above, we have $Zone(r_i)$ is composed of an infinite number of points, $HistoryMove$ is a strict subset of $Zone(r_i)$, and the positions of crashed robots are finite. Thus, robot r_i can always compute and move to a new location that is different from the locations visited by r_i during its last $k+1$ activations.

We now show that robot r_i never visits a location that was visited by another robot r_j during the last previous $k+1$ activations of r_j. Without loss of generality, we consider robot r_i and its next neighbor r_{i+1}. The same proof holds for r_i and its previous neighbor r_{i-1}. Observe that if r_i and r_{i+1} are moving away from each other, then neither robots move to a location that was occupied by the other one for its last $k+1$ activations.

Algorithm 6. Flocking Follower: Code executed by a robot follower r_i.

```
procedure Flocking_Follower($RankSequence$,$d$,$nbr_{act}$,$HistoryMove$)
  $n := |RankSequence|$; and $\alpha := (n-2)180°/n$;
  $P :=$ Formation($P_1, P_2, ..., P_n$) as in Definition 3;
  $P_1 :=$ current position of the leader;
  $\forall r_j \in RankSequence, Target(r_j) = P_j \in Formation(P_1, P_2, ..., P_n)$;
  if ($\forall r_j \in RankSequence$, $r_j$ is within $\epsilon_r$ of $P_j$) then          {Formation = True}
    Lateral_Move_Right();
  else                                                         {Flocking and formation generation}
    $r_{i-1}$:= previous robot to $r_i$ in $RankSequence$;
    $proj_{r_{i-1}}$:= the projection of $r_{i-1}$ on $y-$axis of $r_i$;
    $r_{i+1}$ := next robot to $r_i$ in $RankSequence$;
    $proj_{r_{i+1}}$ := the projection of $r_{i+1}$ on $y-$axis of $r_i$;
    if ($proj_{r_{i-1}} = r_i \wedge proj_{r_{i+1}} = r_i$) then          {same y coordinate as neighbors}
      $Zone(r_i)$ := segment with length $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$ starting at $r_i$ to $Right(r_i)$ (Fig. 2(a));
    else if ($proj_{r_{i-1}} \neq r_i$) $\wedge$ ($proj_{r_{i+1}} \neq r_i$) then
      $Zone(r_i)$ := circle centered at $r_i$, with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$ (Fig. 2(b));
    else if ($proj_{r_{i-1}} \neq r_i \wedge proj_{r_{i+1}} = r_i$) then
      $Zone(r_i)$ := half circle centered at $r_i$, above it, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, proj_{r_{i-1}})/(k+1)(k+2), dist(r_i, r_{i+1})/(k+1)(k+2))$;
    else                                                  {$r_i$ has different y coordinate from next robot}
      $Zone(r_i)$ := half circle centered at $r_i$, below it, and with radius $min(\epsilon_r/(k+1)(k+2), dist(r_i, r_{i-1})/(k+1)(k+2), dist(r_i, proj_{r_{i+1}})/(k+1)(k+2))$(Fig. 2(c));
    end if
    $Intersect$ := the intersection of the segment $\overline{r_iTarget(r_i)}$ with $Zone(r_i)$;
    if ($Intersect \neq r_i$) then                               {$r_i$ is able to progress to its target}
      $S_{CrashInLine}$ := the set of crashed robots in the segment $intersect$;
      if ($S_{CrashInLine} = \emptyset$) then
        $r_i$ moves to the last point in $Intersect$, excluding the points in $HistoryMove$;
      else
        $r_c$ := the closest crashed robot to $r_i$ in $Intersect$;
        $r_i$ moves linearly to the last point in the segment $\overline{r_ir_c}$, excluding $r_c$, and the points in $HistoryMove$;
      end if
    else
      Lateral_Move_Right();
    end if
  end if
  $CurrMove$ := the set of points on the segment $\overline{r_iTarget(r_i)}$;
  if ($nbr_{act} \leq k+1$) then
    $HistoryMove := HistoryMove \cup CurrMove$;
  else
    $HistoryMove := CurrMove$; and $nbr_{act} := 1$;
  end if
end
```

Now assume that both robots r_i and r_{i+1} are moving to the same direction, then we will show that r_i never reaches the position of r_{i+1} after $k+1$ activations of r_{i+1}. Assume the worst case where robot r_{i+1} is activated once during each k activations of r_i. Then, after $k+1$ activations of r_{i+1}, r_i will move toward r_{i+1} by a distance of at most $dist(r_i, r_{i+1})(k+1)^2/(k+1)(k+2)$, which is strictly less than $dist(r_i, r_{i+1})$, hence r_i is unable to reach the position of r_{i+1}.

Finally, we assume that both r_i and r_{i+1} are moving toward each other. In this case, we assume the worst case when both robots are always activated together. After $k+1$ activations of either r_i or r_{i+1}, each of them will travel toward the other one by at most the distance $dist(r_i, r_{i+1})(k+1)/(k+1)(k+2)$. Consequently, $2dist(r_i, r_{i+1})/(k+2)$ is always strictly less than $dist(r_i, r_{i+1})$ because $k \geq 1$. Hence, neither r_i or r_{i+1} moves to a location that was occupied by the other during its last $k+1$ activations, and the lemma holds. □

Corollary 1. *By Algorithm 4, at any time t, there is no overlap between the zones of movement of any two correct robots in the system.*

Agreement on Ranking. In this section, we show that correct robots agree always on the same sequence of ranking even in the presence of failure of robots.

Lemma 6. *By Algorithm 4, correct robots always agree on the same $RankSequence$ when there is no crash. Moreover, if some robot r_j crashes, there is a finite time after which, all correct robots exclude r_j from the ordered set $RankSequence$, and keep the same total order in $RankSequence$.*

Proof (Lemma 6). By Lemma 3, all correct robots agree on the same sequence of ranking, $RankSequence$ after the first k activations of any robot in the system. Then, in the following, we first show that the $RankSequence$ is preserved during the execution of Algorithm 4 when there is no crash in the system. Second, we show that if some robot r_j has crashed, there is a finite time after which correct robots agree on the new sequence of ranking, excluding r_j.

- There is *no crash* in the system: we consider three consecutive robots r_a, r_b and r_c in $RankSequence$, such that $r_a < r_b < r_c$. We prove that the movement of r_b does not allow it to swap ranks with r_a or r_c in the three different cases that follow:
  1. r_a, r_b and r_c share the same y coordinate. In this case, r_b moves by $min(\epsilon_r/(k+1)(k+2), dist(r_b, r_c)/(k+1)(k+2))$ along the segment $\overline{r_b r_c}$. Such a move does not change the y coordinate of r_b, and also it does not change its rank with respect to r_a and r_c because it always stays between r_a and r_c, and it never reaches either r_a nor r_b, by the restrictions on the algorithm.
  2. r_a, r_b and r_c do not share the same y coordinate. In this case, the movement of r_b is restricted within a circle $\mathcal{C}$, centered at r_b, and having a radius that does not allow r_b to reach the same y coordinate as either r_a nor r_c. In particular, the radius of $\mathcal{C}$ is equal to $min(\epsilon_r/(k+1)(k+2), dist(r_b, proj_{r_a})/(k+1)(k+2), dist(r_b, proj_{r_c})/(k+1)(k+2))$, which is less than $dist(r_b, proj_{r_a})/k$, and $dist(r_b, proj_{r_c})/k$, where $proj_{r_a}$ and $proj_{r_c}$ are respectively, the projections of robot r_a and r_c on the $y-$axis of r_b. Hence, such a restriction on the movement of r_b does not allow it to swap its rank with either r_a or r_b.

3. Two consecutive robots have the same y coordinate, (say r_a and r_b), however r_c does not. This case is almost similar to the previous one. The movement of r_b is restricted within a half circle, centered at r_b, and below it, and with a radius that does not allow r_b to have less than or equal y coordinate as r_c. In particular, that radius is equal to $min(\epsilon_r/(k+1)(k+2), dist(r_a, r_b)/(k+1)(k+2), dist(r_b, proj_{r_c})/(k+1)(k+2))$, which is less than $dist(r_a, r_b)/k$, and also less than $dist(r_b, proj_{r_c})/k$, where $proj_{r_c}$ is the projection of robot r_c on the $y-$axis of r_b. Hence, the restriction on the movement of r_b does not allow it to swap ranks with either r_a or r_b.

Since, all robots execute the same algorithm, then the proof holds for any two consecutive robots in $RankSequence$. Note that, the same proof applies for both algorithms executed by the leader and the followers because the restrictions made on their movements are the same

- Some robot r_j *crashes*: From what we proved above, we deduce that all robots agree and preserve the same sequence of ranking, $RankSequence$ in the case of no crash. Assume now that a robot r_j crashes. By Theorem 3, we know that there is a finite time after which all correct robots detect the crash of r_j. Hence, there is a finite time after which correct robots exclude robot r_j from the ordered set $RankSequence$.

In conclusion, the total order in $RankSequence$ is preserved for correct robots during the entire execution of Algorithm 4. This terminates the proof. □

The following Theorem is a direct consequence from Lemma 6.

Theorem 4. *By Algorithm 4, all robots agree on the total order of their ranking during the entire execution of the algorithm.*

Collision-Freedom

Lemma 7. *Under Algorithm 4, at any time t, no two correct robots ever move to the same location. Also, no correct robot ever moves to a position occupied by a faulty robot.*

Proof (Lemma 7). To prove that no two correct robots ever move to the same location, we show that any robot r_i always moves to a location within its own zone $Zone(r_i)$, and the rest follows from the fact that the zones of two robots do not intersect (Corollary 1). By restriction on the algorithm, r_i must move to a location $Target(r_i)$, which is within $Zone(r_i)$. Since, r_i belongs to $Zone(r_i)$, $Zone(r_i)$ is a convex form or a line segment, and the movement of r_i is linear, so all points between r_i and $Target(r_i)$ must be in $Zone(r_i)$.

Now we prove that, no correct robot ever moves to a position occupied by a crashed robot. By Theorem 1, robot r_i can compute the positions of crashed robots in finite time. Moreover, by Lemma 5, robot r_i always has free destinations within its zone $Zone(r_i)$, which excludes crashed robots. Finally, Algorithm 4 restricts robots from moving to the locations that are occupied by crashed robots. Thus, robot r_i never moves to a location that is occupied by a crashed robot. □

The following theorem is a direct consequence from Lemma 7.

Theorem 5. *Algorithm 4 is collision free.*

Fault-tolerant Flocking. Before we proceed, we state the following lemma, which sets a bound on the number of faulty robots under which a polygon can be formed.

Lemma 8. *A polygon is generated if and only if the number of faulty robots f is bounded by $f \leq n - 3$, where n is the number of robots in the system, and $n \geq 3$.*

Proof (Lemma 8). The proof is trivial. A polygon requires three or more robots to be formed. Then, the number of robots n in the system should be greater or equal to three. Also, the number of faulty robots f at any time t in the system should be less than or equal to $n - 3$ for the polygon to be formed. This proves the lemma. □

Lemma 9. *Algorithm 4 allows correct robots to form an approximation of a regular polygon in finite time, and to maintain it in movement.*

Proof (Lemma 9). We first show that each robot can be within ϵ_r of its target in the formation $F(P_1, P_2, ..., P_n)$ in a finite number of steps. Second, we show that correct robots maintain an approximation of the formation while moving.

Assume that r_i is a correct robot in the system. If r_i is a leader, then by Algorithm 4, the target of r_i is a point within a circle or half circle, centered at r_i, and with radius less than or equal to ϵ_r satisfying Assumption 3, and excluding the positions of crashed robots. Since, there exists an infinite number of points within $Zone(r_i)$, and by Assumption 2, the cycle of a robot is finite, then r_i can reach its target within $Zone(r_i)$ in a finite number of steps.

Now, consider that r_i is a robot follower. We also show that r_i can reach within ϵ_r of its target P_i in a finite number of steps. We consider two cases:

- Robot r_i can move freely toward its target P_i: every time r_i is activated, it can progress by at most $\epsilon_r/(k+1)(k+2)$. Since, the distance $dist(r_i, P_i)$ is finite, the bound k of the scheduler is also finite, and the cycle of a robot is finite by Assumption 2, then r_i can be within ϵ_r of P_i in a finite number of steps.
- Robot r_i cannot move freely toward its target P_i: first, assume that r_i cannot progress toward its target because of the restriction on $RankSequence$. Since, there exists at least one robot in $RankSequence$ that can move freely toward its target, and this is can be done in finite time. In addition, the number of robots in $RankSequence$ is finite, and by Lemma 5, a robot can always move to a new location satisfying Assumption 3, then, eventually each robot r_i in $RankSequence$ can progress toward its target P_i, and arrive within ϵ_r of it in a finite number of steps.
  Now, assume that r_i cannot progress toward its target P_i because it is blocked by some crashed robots. By Lemma 5, a robot can always move to a new location satisfying Assumption 3. Also, the number of crashed robots is finite, so eventually robot r_i can make progress, and be within ϵ_r of its target in a finite number of steps, by similar arguments.

We now show that correct robots maintain an approximation of the formation while moving. Since, all robots are restricted to move within one cycle by at most

$\epsilon_r/(k+1)(k+2)$, then in every new k activations in the system, each correct robot r_i cannot go farther away than ϵ_r from its position during k activations. Consequently, r_i can always be within ϵ_r of its target P_i as in Definition 4, and the lemma follows. □

Theorem 6. *Algorithm 4 allows correct robots to dynamically form an approximation of a regular polygon, while avoiding collisions.*

Proof (Theorem 6). First, by Theorem 3, there is a finite time after which all correct robots agree on the same set of correct robots. Second, by Theorem 4, all correct robots agree on the total order of their ranking $RankSequence$. Third, By Theorem 5, there is no collision between any two robots in the system, including crashed ones. Finally, by Lemma 9, all correct robots form an approximation of a regular polygon in finite time, and the theorem holds. □

Lemma 10. *Algorithm 4 tolerates permanent crash failures of robots.*

Proof (Lemma 10). By Theorem 1, a crash of a robot is detected in finite time, and by Algorithm 4, a crashed robot is removed from the list of correct robots, although it appears physically in the system. Finally, by Theorem 5, correct robots avoid collisions with crashed robots. Thus, Algorithm 4 tolerates permanent crash failures of robots. □

From Theorem 6, and Lemma 10, we infer the following theorem:

Theorem 7. *Algorithm 4 is a fault tolerant dynamic flocking algorithm that tolerates permanent crash failures of robots.*

7 Conclusion

In this paper, we have proposed a fault-tolerant flocking algorithm that allows a group of asynchronous robots to self organize dynamically, and form an approximation of a regular polygon, while maintaining this formation in movement. The algorithm relies on the assumption that robots' activations follow a k-bounded asynchronous scheduler, and that robots have a limited memory of the past.

Our flocking algorithm allows correct robots to move in any direction, while keeping an approximation of the polygon. Unlike previous works (e.g., [3,6]), our algorithm is fault-tolerant, and tolerates permanent crash failures of robots. The only drawback of our algorithm is the fact that it does not permit the rotation of the polygon by the robots, and this is due to the restrictions made on the algorithm in order to ensure the agreement on the ranking by robots. The existence of such algorithm is left as an open question that we will investigate in our future work.

Finally, our work opens new interesting questions, for instance it would be interesting to investigate how to support flocking in a model in which robots may crash and recover.

Acknowledgments

This work is supported by the JSPS (Japan Society for the Promotion of Science) postdoctoral fellowship for foreign researchers (ID No.P 08046).

References

1. Daigle, M.J., Koutsoukos, X.D., Biswas, G.: Distributed diagnosis in formations of mobile robots. IEEE Transactions on Robotics 23(2), 353–369 (2007)
2. Coble, J., Cook, D.: Fault tolerant coordination of robot teams, citeseer.ist.psu.edu/coble98fault.html
3. Gervasi, V., Prencipe, G.: Coordination without communication: the Case of the Flocking Problem. Discrete Applied Mathematics 143(1-3), 203–223 (2004)
4. Hayes, A.T., Dormiani-Tabatabaei, P.: Self-organized flocking with agent failure: Off-line optimization and demonstration with real robots. In: Proc. IEEE Intl. Conference on Robotics and Automation, vol. 4, pp. 3900–3905 (2002)
5. Saber, R.O., Murray, R.M.: Flocking with Obstacle Avoidance: Cooperation with Limited Communication in Mobile Networks. In: Proc. 42nd IEEE Conference on Decision and Control, pp. 2022–2028 (2003)
6. Canepa, D., Potop-Butucaru, M.G.: Stabilizing flocking via leader election in robot networks. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 52–66. Springer, Heidelberg (2007)
7. Défago, X., Gradinariu, M., Messika, S., Raipin-Parvédy, P.: Fault-tolerant and self-stabilizing mobile robots gathering. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 46–60. Springer, Heidelberg (2006)
8. Prencipe, G.: CORDA: Distributed Coordination of a Set of Autonomous Mobile Robots. In: Proc. European Research Seminar on Advances in Distributed Systems, pp. 185–190 (2001)
9. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Pattern Formation by Autonomous Robots Without Chirality. In: Proc. 8th Intl. Colloquium on Structural Information and Communication Complexity (SIROCCO 2001), pp. 147–162 (2001)
10. Gervasi, V., Prencipe, G.: Flocking by A Set of Autonomous Mobile Robots. Technical Report, Dipartimento di Informatica, Università di Pisa, Italy, TR-01-24 (2001)
11. Reynolds, C.W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. Journal of Computer Graphics 21(1), 79–98 (1987)
12. Brogan, D.C., Hodgins, J.K.: Group Behaviors for Systems with Significant Dynamics. Autonomous Robots Journal 4, 137–153 (1997)
13. John, T., Yuhai, T.: Flocks, Herds, and Schools: A Quantitative Theory of Flocking. Physical Review Journal 58(4), 4828–4858 (1998)
14. Yamaguchi, H., Beni, G.: Distributed Autonomous Formation Control of Mobile Robot Groups by Swarm-based Pattern Generation. In: Proc. 2nd Int. Symp. on Distributed Autonomous Robotic Systems (DARS 1996), pp. 141–155 (1996)
15. Dieudonné, Y., Petit, F.: A Scatter of Weak Robots. Technical Report, LARIA, CNRS, France, RR07-10 (2007)
16. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
17. Schreiner, K.: NASA's JPL Nanorover Outposts Project Develops Colony of Solar-powered Nanorovers. IEEE DS Online 3(2) (2001)
18. Souissi, S., Yang, Y., Défago, X.: Fault-tolerant Flocking in a k-bounded Asynchronous System. Technical Report, JAIST, Japan, IS-RR-2008-004 (2008)
19. Konolige, K., Ortiz, C., Vincent, R., Agno, A., Eriksen, M., Limketkai, B., Lewis, M., Briesemeister, L., Ruspini, E., Fox, O., Stewart, J., Ko, B., Guibas, L.: CENTIBOTS: Large-Scale Robot Teams. Journal of Multi-Robot Systems: From Swarms to Intelligent Autonoma (2003)
20. Bellur, B.R., Lewis, M.G., Templin, F.L.: An Ad-hoc Network for Teams of Autonomous Vehicles. In: Proc. 1st IEEE Symp. on Autonomous Intelligent Networks and Systems (2002)
21. Jennings, J.S., Whelan, G., Evans, W.F.: Cooperative Search and Rescue with a Team of Mobile Robots. In: Proc. 8th Intl. Conference on Advanced Robotics, pp. 193–200 (1997)

Bounds for Deterministic Reliable Geocast in Mobile Ad-Hoc Networks[*]

Antonio Fernández Anta and Alessia Milani

LADyR, GSyC, Universidad Rey Juan Carlos, Spain

Abstract. In this paper we study the impact of the speed of movement of nodes on the solvability of deterministic reliable geocast in mobile ad-hoc networks, where nodes move in a continuous manner with bounded maximum speed. Nodes do not know their position, nor the speed or direction of their movements. Nodes communicate over a radio network, so links may appear and disappear as nodes move in and out of the transmission range of each other. We assume that it takes a given time T for a single-hop communication to reliably complete. The mobility of nodes may be an obstacle for deterministic reliable communication, because the speed of movements may impact on how quickly the communication topology changes.

Assuming the two-dimensional mobility model, the paper presents two tight bounds for the solvability of deterministic geocast. First, we prove that the maximum speed $v_{max} < \frac{\delta}{T}$ is a necessary and sufficient condition to solve the geocast, where δ is a parameter that together with the maximum speed captures the local stability in the communication topology. We also prove that $\Omega(nT)$ is a time complexity lower bound for a geocast algorithm to ensure deterministic reliable delivery, and we provide a distributed solution which is asymptotically optimal in time.

Finally, assuming the one-dimensional mobility model, i.e. nodes moving on a line, we provide a lower bound on the speed of movement necessary to solve the geocast problem, and we give a distributed solution. The algorithm proposed is more efficient in terms of time and message complexity than the algorithm for two dimensions.

Keywords: Mobile ad-hoc network, geocast, speed of movement towards solvability, distributed algorithms.

1 Introduction

A mobile ad-hoc network (MANET) is a set of mobile wireless nodes which dynamically build a network, without relying on a stable infrastructure. Direct communication links are created between pairs of nodes as they come into the transmission range of each other. If two nodes are too far apart to establish a direct wireless link, other nodes act as relays to route messages between them. This self-organizing nature of mobile

[*] This research was supported in part by Comunidad de Madrid grant S-0505/TIC/0285; Spanish MEC grants TIN2005-09198-C02-01 and TIN2008-06735-C02-01.The work of Alessia Milani is funded by a Juan de la Cierva contract.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 164–183, 2008.

ad-hoc networks makes them specially interesting for scenarios where networks have to be built on the fly, e.g., under emergency situation, in military operations, or in environmental data collection and dissemination.

A fundamental communication primitive in certain mobile ad-hoc network is *geocasting* [14]. This is an operation initiated by a node in the system, called the *source*, that disseminates some information to all the nodes in a given geographical area, named the *geocast region*. In this sense, the geocast primitive is a variant of multicasting, where nodes are eligible to deliver the information according to their geographical location. While geocasting in two dimensions is clearly useful, geocasting in one dimension is also a natural operation in some real situations, like announcing an accident to the nearby vehicles in a highway. In mobile ad-hoc environments, geocasting is also a basic building block to provide more complex services. As an example, Dolev et al. [5] use a deterministic reliable geocast service to implement atomic memory in mobile ad-hoc networks. A geocast service is *deterministic* if it ensures deterministic reliable delivery, i.e. all the nodes eligible to deliver the information will surely deliver it.

Designing a geocast primitive in a mobile ad-hoc network forces to deal with the uncertainty due to the dynamics of the network. Since communication links appear and disappear as nodes move in and out of the transmission range of other nodes, there is a (potential) continuous change of the communication topology. In other words, the movement of nodes and their speed of movement usually impacts on the lifetime of radio links. Then, since it takes at least $\Omega(\log n)$ time to ensure a one-hop successful transmission in a network with n nodes [6], mobility may be an obstacle for deterministic reliable communication.

Our contributions. In this paper we study the impact of the maximum speed of movement of nodes on the solvability of deterministic geocast in mobile ad-hoc networks. In our model we assume that a node does not know its position (nodes have no GPS or similar device), and that it knows neither the speed nor the direction of its movement. Additionally, we assume that it takes a given time T for a single-hop communication to succeed. As far as we know, [1] is the only theoretical work that deals with the geocast problem in such a model.

Our results improve and generalize the bounds presented in [1] and, to the best of our knowledge, present the first deterministic reliable geocast solution for two dimensions, i.e. where nodes move in a continuous manner in the plane. In particular, we give bounds on the maximum speed of nodes in order to be able to solve the deterministic reliable geocast problem in one and two dimensions. While the bounds provided in [1] are for a special class of algorithms, our bounds apply to all geocasting algorithms and are tighter.

Then, we present a distributed solution for the two-dimensional mobility model, which is proved to be asymptotically optimal in terms of time complexity. Let n be the number of nodes in the system, it takes $3nT$ time for our solution to ensure that the geocast information is reliably delivered by all the eligible nodes. Additionally, we prove that $\Omega(nT)$ is a time complexity lower bound for a geocast algorithm to ensure deterministic reliable delivery. Finally, we provide a distributed geocast algorithm for the one-dimensional case (i.e. nodes move on a line) and upper bound its message and time complexity. This algorithm is more efficient in terms of message complexity than the algorithms proposed in [1], and (not surprisingly) than the algorithm for two dimension.

Related work. Initially introduced by Imielinski and Navas [14] for the Internet, the geocast problem was then proposed for mobile ad-hoc networks by Ko and Vaidya [7]. The majority of geocast algorithms presented in the literature for mobile ad-hoc networks provide probabilistic guarantees, e.g. [8, 12, 9]. See the review of Jinag and Camp [4] for an overview of the main existing geocast algorithms. As mentioned above, Baldoni et al. [1] provide a deterministic solution for the case where nodes move on a line.

Other deterministic solutions for multicast and broadcast in MANETs have been proposed, but their correctness relies on strong synchronization or stability assumptions. In particular, Mohsin et al. [13] present a deterministic solution to solve broadcast in one-dimensional mobile ad-hoc networks. They assume that nodes move on a linear grid, that nodes know their current position on the grid, and that communication happens in synchronous rounds. Gupta and Srimani [10], and Pagani and Rossi [16] provide two deterministic multicast solutions for MANET, but they require the network topology to globally stabilize for long enough periods to ensure delivery. Moreover, they assume a fixed and finite number of nodes arranged in some logical or physical structure.

Few bounds on deterministic communication in MANETs have been provided [15, 2]. We prove that the lower time complexity bound to complete a geocast in the plane is $\Omega(nT)$. Interestingly, Prakash et al. [15] provide a lower bound of $\Omega(n)$ rounds for the completion time of broadcast in mobile ad hoc networks, where n is the number of nodes in the network. As the authors point out, they consider grid-based networks, but a lower bound proved for this restricted grid mobility model automatically applies to more general mobility models. This latter result improves the $\Omega(D \log n)$ bound provided by Bruschi and Pinto [2], where D is the diameter of the network. These results unveil the fact that, when nodes may move, the dominating factor in the complexity of an algorithm is the number of nodes in the network and not its diameter.

Road map. In Section 2 we present the model for mobile ad hoc network we consider and in Section 3 we revise the problem. Then, in Section 4 we present the results for two dimensions and in Section 5 the results for one dimension. Finally, our conclusions are presented in Section 6.

2 A Mobile Ad-Hoc Network Model

We consider a finite set Π of n (mobile) nodes which move in a continuous manner on a plane (2-dimensional Euclidean space) with bounded maximum speed v_{max}. The nodes in Π do not have access to a global clock, but their local clocks run at the same rate. Additionally, no node in Π fails.

Nodes communicate by exchanging messages over a wireless radio network. All the nodes have the same transmission radius r. At any time, each node is a neighbor of, and can communicate with, all the nodes that are within its transmission range at that time, i.e., the nodes that are *completely inside* a disk, centered at the node's position, of radius r [3]. Formally, let $distance(p, q, t)$ denote the distance between p and q at time t, we say that p and q are *neighbors* at time t if $distance(p, q, t) < r$. Nodes do not know their position, speed, nor direction of movement.

Local broadcast primitive. To directly communicate with their neighbors, nodes are provided with a *local reliable broadcast* primitive. Communication is not instantaneous, it takes some time for a broadcast message to be received. To simplify the presentation we consider as time unit the *time slot*. This is the time the communication of a message takes when accessing the underlying wireless network communication channel without collision. Additionally, we assume that local computation at the nodes takes negligible time (zero time for the purpose of the analyses). Since collisions are a intrinsic characteristic of MANETs, they have to be considered. We assume that the potential collisions due to concurrent broadcasts by neighbors are dealt by a lower level communication layer, and that this layer takes T units of time to (reliably and deterministically) deliver a message to its destination. The value of T could be related to the size of the system and depends on the complexity of the lower level communication protocol. As already stated, [6] shows that it takes at least $\Omega(\log n)$ time to ensure a one-hop successful transmission in a network with n nodes.

Then, if a node p invokes $broadcast(m)$ at time t, then all nodes that remain neighbors of p throughout $[t, t+T]$ receive m by time $t+T$, for some fixed *known* integer $T > 0$. A node that receives a message m generates a $receive(m)$ event. It is possible that some node that has been a neighbor of p at some time in $[t, t+T]$ (but not during the whole period) also receives m, but there is no such guarantee. However, no node receives m after time $t+T$. A node issues a new invocation of the broadcast primitive only after it has completed the previous one (T time later). Then in each time interval of length T a node broadcasts at most one message.

Connectivity. Baldoni *et al.* [1] have proved that traditional connectivity is too weak to implement a deterministic geocasting primitive in the model described. To overcome this impossibility result they have introduced the notion of *strong connectivity*, and assumed it in their model. Like them, we also assume strong connectivity in our model[1]. Let us remark that strong connectivity is only a possible way to overcome the above impossibility. A different approach could be to constrain the mobility pattern of nodes (e.g. [13]) or to assume the global communication topology to be stable long enough to ensure reliable delivery (e.g. [10]). On the other hand, strong connectivity is a local property which helps to formalize the local stability in the communication topology necessary to solve the problem. In the following, we revise the notions of strong neighborhood and strong connectivity.

Definition 1 (Strong neighborhood). *Let $\delta_2 = r$ and δ_1 be fixed positive real numbers such that $\delta_1 < \delta_2$. Two nodes p and p' are* strong neighbors *at some time t if there is a time $t' \leq t$ such that $distance(p, p', t') \leq \delta_1$, and $distance(p, p', t'') < \delta_2$ for all $t'' \in [t', t]$.*

Assumption 1 (Strong Connectivity). *For every pair of nodes p and p', and every time t, there is at least one path of strong neighbors connecting p and p' at t.*

When convenient, we may use that a pair of (strong) neighbors have a (strong) connection, or are (strongly) connected. Observe that once two nodes p and p' become strong

[1] In this paper we do not consider the Delay/Disruption Tolerant Networks model.

neighbors (i.e., they are at distance δ_1 from each other), to get disconnected they must move away from each other so that their distance is at least δ_2. This means that the total distance to be covered in order for p and p' to disconnect is $\delta_2 - \delta_1$. We use the notation $\delta = \frac{\delta_2 - \delta_1}{2}$, where δ denotes the minimum distance that any two nodes that just became strong neighbors have to travel to stop being neighbors when moving in opposite directions. Thus, for a clearer presentation of our results, we express the maximum speed of nodes, denoted v_{max}, as the ratio between δ and the time necessary to travel this space, denoted T'. Formally,

Assumption 2 (Movement Speed). *It takes at least $T' > 0$ time for a node to travel distance $\delta = \frac{\delta_2 - \delta_1}{2}$, i.e. $v_{max} = \frac{\delta_2 - \delta_1}{2T'}$.*

Since nodes move, the topology of the network may continuously change. In this sense, assuming both strong connectivity and an upper bound on the maximum speed of nodes provides some topological stability in the network. In particular, it ensures that there are periods in which the neighborhood of a node remains stable. Formally,

Lemma 1. *If two nodes become strong neighbors at time t, then they are neighbors throughout the interval $(t - T', t + T')$ and remain strong neighbors throughout the interval $[t, t + T')$.*

Proof. If p and p' become strong neighbors at time t, then $distance(p, p', t) = \delta_1$. To be disconnected, they must move away from each other a distance of at least 2δ, so that their distance is at least δ_2. From Assumption 2, this takes at least T' time. Hence, for $\tau \in (t - T', t + T')$, $distance(p, q, \tau) < \delta_1 + 2\delta = \delta_2$, which combined with Definition 1 proves the claims.

3 The Geocast Problem

The geocast is a variant of the conventional multicasting problem, where nodes are eligible to deliver the information if they are located within a specified geographic region. The geocast region we consider is the circular area centered in the location where the source starts the geocasting and whose radius is some given value d. We assume d to be provided as input by the user of the geocast primitive.

The geocast problem is solved by a geocast algorithm run on the mobile nodes, which implements the following geocast primitives: $Geocast(I, d)$ to geocast information I at distance d, and $Deliver(I)$ to deliver information I (previously geo-casted). The geocast algorithm uses the $broadcast(m)$ and $receive(m)$ primitives to achieve communication among neighbors. The geocast information I is initially known by exactly one node, *the source*. If the source invokes $Geocast(I, d)$ at time t, being at location l, then there are three properties that must be satisfied by the geocast service.

Property 1. [Reliable Delivery] There is a positive integer C such that, by time $t + C$, information I is delivered (with $Deliver(I)$) at all nodes that are located at most at distance d away from l throughout $[t, t + C]$.

The following properties rule out solutions which waste resources causing continuous communication or distribution of the information I to the whole Euclidean space.

Property 2. [Termination] If no other node issues a call to the geocast service, then there is a positive integer C' such that after time $t + C'$, no node performs any communication (i.e. a local broadcast) triggered by a geocast.

Property 3. [Integrity] There is a $d' \geq d$ such that, if a node has never been within distance d' from l, it never delivers I.

Observe that these properties are deterministic. This justifies the use of a deterministic reliable local broadcast primitive and the fact that we enforce nodes to be in range less than δ_2 during T steps to complete a successful communication.

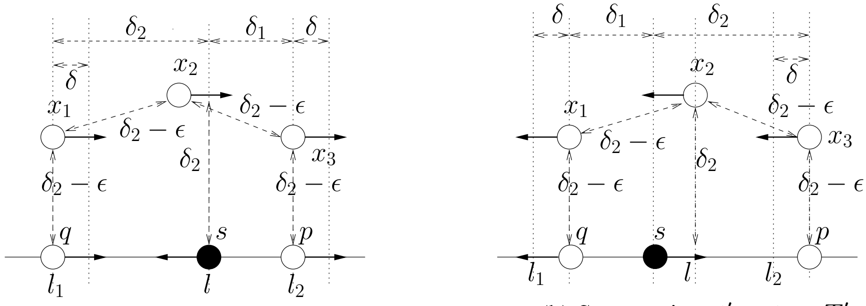
4 Solving the Geocast Problem in Two Dimensions

In this section we first show that in two dimensions T' must be larger than T for the geocast problem to be solvable. Then, we present an algorithm that solves the problem if this condition is met, whose time complexity is $O(nT)$. Finally we prove that this complexity is optimal, since any algorithm has executions in which $\Omega(nt)$ time is required to complete the geocast.

Theorem 3. *No algorithm can solve the geocast problem in two dimension if* $v_{max} \geq \frac{\delta_2 - \delta_1}{2T}$*, i.e. if* $T' \leq T$.

Proof. Consider a $Geocast(I, d)$ primitive invoked at some time t by a source s, with $d \geq \delta_2$. To prove the claim we construct a scenario with 6 nodes, and an execution in it, such that the geocast region contains several nodes permanently, but only s delivers I. Since the reliable delivery property is not satisfied, this proves the claim.

In our scenario, there are 6 nodes, the source s, and nodes p, q, x_1, x_2, and x_3. At the time t of the geocast, we assume that the system is in the state shown in Figure 1.(a). This state can be reached from an initial situation in which the nodes q, x_1, x_2, x_3, p, and s are placed (in this order) on a line, at distance δ_1 each one from the next, and move without breaking the strong connectivity, to the state of Figure 1.(a). Observe that all nodes are strongly connected along the path q, x_1, x_2, x_3, p, s, but that the source is



(a) State at time t and t_0

(b) State at time $t'_0 = t_0 + T'$

Fig. 1. Scenario for the proof of Theorem 3

not a neighbor of neither x_1, x_2, nor x_3. Additionally, both p and q are in the geocast region (since $d \geq \delta_2$). They will be in the region during the whole execution and hence to satisfy reliable delivery they should deliver I.

We consider several possible behaviour of the geocast algorithm. Let us first assume then that, although it invoked $Geocast(I, d)$, s never makes a call to $broadcast(I)$. Then, p and q will never receive nor deliver I and reliable delivery is violated. Otherwise, assume that as a consequence of the $Geocast(I, d)$ invocation, s invokes $broadcast(I)$ at times $t_0, t_1, ...$, with $t_{i+1} \geq t_i + T$. Let us define first the behavior of the nodes in interval $[t_0, t_1]$. At time t_0, the source s and node q start moving towards each other at the maximum speed v_{max}, while nodes p, x_1, x_2, and x_3 start moving in the same direction as q. At time $t'_0 = t_0 + T' \leq t_1$ all nodes have travelled a distance of δ (by definition of T') and the system is in the state depicted in Figure 1.(b). In the interval $[t'_0, t_1]$ no node moves.

Observe that strong connectivity has been preserved during the whole period $[t_0, t'_0]$, since the distances along the path q, x_1, x_2, x_3, p did not change, and the source is a strong neighbor of p for all the period $[t_0, t'_0)$ and at time t'_0 it becomes strong neighbor of q. Note also that neither p nor q have been neighbors of s during the whole period $[t_0, t_0 + T]$, because q is not a neighbor at time t_0 and p is not a neighbor at time $t'_0 \leq t_0 + T$. Hence, in our execution no node delivers I in $[t_0, t_1]$.

The behavior in interval $[t_1, t_2]$ is the same as described for $[t_0, t_1]$, but swapping the directions of movement and the roles of p and q. The initial state at time t_1 is the one show in Figure 1.(b), and the final state reached at time $t'_1 = t_1 + T'$ is the one shown in Figure 1.(a). Again, I is not delivered at p nor q because they have not been neighbors of s in the whole period $[t_1, t_1 + T]$. For any interval $[t_i, t_{i+1}]$ the behavior is the same as in interval $[t_0, t_1]$, if i is even, and the same as in interval $[t_1, t_2]$ if i is odd. Then, in this scenario only s delivers I and the reliable delivery property is not satisfied.

The above theorem gives a lower bound of $T' > T$ to be able to solve the geocast problem. We show now that this bound is tight by presenting an algorithm that always solves the problem if $T' > T$. The algorithm belongs to the class of algorithms presented in Figure 2, which has a configuration parameter M, the bound on time that the algorithm uses to stop the geocast. The algorithm M-$Geocast(I, d)$ works as follows. When the source node invokes a call $Geocast(I, d)$, it immediately delivers the information I (Line 8). Then, it broadcasts I and stores in a local variable TLB the time this first transmission happened (Lines 10-11), in order to retransmit every T units of time (Lines 13-14). When a node p receives for the first time a message with information I, it immediately delivers it and starts rebroadcasting the information periodically (Lines 2-6). With the information I the algorithm broadcasts a value $count_I$, which contains an estimate of the time that has passed since the geocast started. This value combined with the the parameter M is used to terminate the algorithm.

We show now that the algorithm M-$Geocast(I, d)$ solves the geocast problem in two dimensions for an appropriate value of M. Let us denote by S the set of nodes that have already delivered the information I, and $S(t)$ the set S at time t. Let us denote by t_i the time at which the set S increases from size i to $i + 1$. Note that t_0 is the time the geocast starts.

```
Init
    $TLB_I \leftarrow \bot$

upon event $\langle receive(I, c)\rangle$
    if $(TLB_I = \bot)$ then
        trigger $\langle Deliver(I)\rangle$;
        $count_I \leftarrow c + 1$;
        trigger $\langle broadcast(I, count_I)\rangle$
        $TLB_I \leftarrow clock$
    end if

Procedure $M\text{-}Geocast(I, d)$
    trigger $\langle Deliver(I)\rangle$;
    $count_I \leftarrow 0$;
    trigger $\langle broadcast(I, count_I)\rangle$;
    $TLB_I \leftarrow clock$

when $(clock = TLB_I + T)$ and $(count_I < M)$
    $count_I \leftarrow count_I + T$;
    trigger $\langle broadcast(I, count_I)\rangle$
    $TLB_I \leftarrow TLB_I + T$
```

Fig. 2. The code of $M\text{-}Geocast(I, d)$ algorithm class

Lemma 2. *If $T' > T$ and $count_I < M$ at all nodes during the time interval $[t_0, t_{n-1}]$, then $t_{i+1} - t_i \leq 3T$ for every $i \in \{0, \ldots, n-2\}$.*

Proof. Since strong connectivity holds, at any time there must be chains of strong neighbors connecting any two nodes in the system. In particular, at every time $t_0 < t < t_{n-1}$ (i.e., such that $S(t) \neq \Pi$) there must exist at least a pair of neighbors q and p such that $q \in S(t)$ and $p \notin S(t)$. Let $C(t)$ denote the set of all such pairs.

Let us fix an $i \in \{0, \ldots, n-2\}$, and assume, for contradiction, that $t_{i+1} - t_i > 3T$. Consider the case when there is some pair $(q, p) \in C(t_i)$ that belongs to $C(t')$ for all $t' \in [t_i, t_i + 2T]$. In other words, this pair is formed by a node q that has I at t_i, and a node p that does not, neighbors for at least $2T$ time. By the $M\text{-}Geocast(I, d)$ algorithm and the fact that $count_I < M$ during the time interval $[t_0, t_{n-1}]$, a node having the information I will rebroadcast it once every T time. Hence q will rebroadcast the information I at some time $t' \in [t_i, t_i + T]$, and thus p will receive and deliver it by time $t' + T \leq t_i + 2T$.

Otherwise, all the connections in $C(t_i)$, $i \in \{0, \ldots, n-2\}$, have been broken by some time $t' \in (t_i, t_i + 2T]$. But, for strong connectivity to hold, a strong connection has to exist between some node $q \in S(t')$ and a node $p \notin S(t')$, since otherwise these subsets are disconnected at time t'. Let t'', $t_i < t'' \leq t'$, be the time at which q and p become strong neighbors, i.e. they are within distance δ_1 from each other. The claim follows if $q \notin S(t_i)$, since $q \in S(t')$ and $t_i < t'' \leq t' \leq t_i + 2T$. Otherwise, note that by Lemma 1 and the fact that $T' > T$, q and p are neighbors throughout all the period $[t'' - T, t'' + T]$. Then, since $q \in S(t_i)$ and $t_i < t''$, q will broadcast I once in the period $[t'' - T, t'']$, and p will deliver I by time $t'' + T > t' > t_i$. Given that $t'' \leq t_i + 2T$, p will deliver the information I by time $t_i + 3T$ and the claim holds.

Let us now relate the value of the $count_I$ at each node with respect to the time that has passed since $M\text{-}Geocast(I, d)$ was invoked. Let $count_I(q, t)$ be the value of the variable $count_I$ of node q at time t. Let us define a *propagation sequence* as the sequence of nodes $s = p_0, p_1, p_2, ..., p_k = q$ such that the first message received by p_i with information I was sent by p_{i-1}. Node $s = p_0$ is the source of the geocast.

Lemma 3. *Let t_0 be the time at which $M\text{-}Geocast(I, d)$ is invoked at source s. Given a node q with propagation sequence $s = p_0, p_1, p_2, ..., p_k = q$ and a time $t \geq t_0$ at* *which q has delivered I, with $count_I(q,t) \leq M$, then it is satisfied that $((t-t_0) - count_I(q,t)) \in [0, k(T-1)+T]$.*

Proof. We prove by induction on k that at time $t \geq t_0$ it is satisfied that $((t-t_0) - count_I(p_k,t)) \in [0, k(T-1)+T]$, and that if a message is sent at time t it carries a counter $c(p_k,t)$ such that $((t-t_0) - c(p_k,t)) \in [0, k(T-1)]$. The base case is the source node $s = p_0$. At time t_0 the source sets $count_I(s,t_0) = 0$ (Line 9), and then, as long as $count_I < M$, it increments $count_I$ by T every T time (Line 12). Hence, at time $t = t_0 + \alpha$ we have $((t-t_0) - count_I(s,t)) = 0$ if α is a multiple of T, and $((t-t_0) - count_I(s,t)) > 0$ otherwise. Furthermore, the difference $(t-t_0) - count_I$ is always smaller than T. Since messages are broadcast at times $t = t_0 + \alpha$ with α a multiple of T, the values $c(s,t)$ carried by the messages sent by the source satisfy $((t-t_0) - c(s,t)) = 0$.

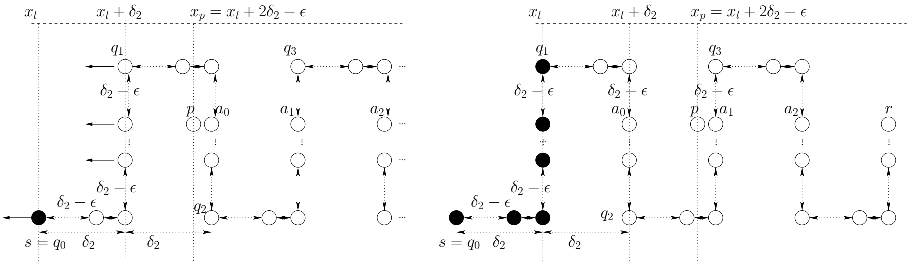
Let us assume now by induction that, if p_{i-1} broadcasts a message at time $t \geq t_0$, this carries a value $c(p_{i-1},t)$ such that $((t-t_0) - c(p_{i-1},t)) \in [0, (i-1)(T-1)]$. If p_i receives I for the first time at t' and the corresponding message was sent by p_{i-1} at time t, p_i sets $count_I(p_i,t') = c(p_{i-1},t) + 1$ (Line 4). This message took between 1 and T time units to be received at time $t' = t + \alpha$. Hence, $\alpha \in [1, T]$. Considering one extreme case, if $((t-t_0) - c(p_{i-1},t)) = 0$ and $\alpha = 1$, then $((t'-t_0) - count_I(p_i,t')) = 0$. In the other extreme, if $((t-t_0) - c(p_{i-1},t)) = (i-1)(T-1)$ and $\alpha = T$, then $((t'-t_0) - count_I(p_i,t')) = i(T-1)$. Therefore, $((t'-t_0) - count_I(p_i,t')) \in [0, i(T-1)]$. Like the source, p_i increments $count_I$ by T every T time as long as $count_I < M$ (Line 12). Hence, at any time $t'' = t' + \alpha$ we have $((t''-t_0) - count_I(p_i,t'')) \in [0, i(T-1)]$ if α is a multiple of T. Otherwise, this difference increases in up to T time, and hence $((t''-t_0) - count_I(p_i,t'')) \in [0, i(T-1)+T]$. Since messages are broadcast by p_i at times $t'' = t' + \alpha$ with α a multiple of T, the values $c(p_i,t'')$ carried by the messages sent by p_i satisfy $((t''-t_0) - c(p_i,t'')) \in [0, i(T-1)]$.

This lemma can be used to prove the following theorem, which shows that the geocast problem can be solved in two dimensions as long as $T' > T$.

Theorem 4. *If $T' > T$, the M-Geocast(I,d) algorithm with $M = 3T(n-1)$ ensures*
(1) the Reliable Delivery Property 1 for $C = 3T(n-1)$,
(2) the Termination Property 2 for $C' = 3T(n-1) + (n-1)(T-1) + T$, and
(3) the Integrity Property 3 for $d' = \max(d, 3T(n-1)v_{max} + (n-1)\delta_2)$.

Proof. The first part of the claim is a direct consequence of Lemma 2, which proves that at most $3T(n-1) \geq t_{n-1} - t_0$ time after *Geocast*(I,d) is invoked, all nodes have delivered the information I. The second part of the claim follows from Lemma 3, using the fact that no propagation sequence has more than n nodes (hence taking $k = n-1$), combined with the first part of the claim. The third claim is also direct consequence of Lemma 2, since the information can be carried by nodes at most distance $3T(n-1)v_{max}$ in time $3T(n-1)$ from the initial location of the source, and travels less than $(n-1)\delta_2$ in the $n-1$ broadcasts that inform new nodes.

Finally we show that the time bound found for the M-*Geocast*(I,d) algorithm with $M = 3T(n-1)$ is in fact asymptotically optimal, since there are cases in which any geocast algorithm requires $\Omega(nT)$ time to complete.



(a) Status at time $t_0 = t_i$

(b) Status at time $t_1 = t_{i+1}$

Fig. 3. Scenario for the proof of Theorem 5

Theorem 5. *Any deterministic Geocast(I, d) algorithm that ensures the Reliable Delivery Property in two dimensions requires time $\Omega(nT)$ to complete, for each $d \geq 3\delta_2^2$.*

Proof. We present a scenario (shown in Figure 3) in which for any $Geocast(I, d)$ call, with $d \geq 3\delta_2^2$, all nodes that are in the geocast region require time $(n-1)T$ to deliver I. Consider the scenario depicted in Figure 3.(a) where $Geocast(I, d)$ is invoked at time t_0 at the source node s while in location $l = (x_l, y_l)$ and with only one neighbor at distance $\delta_2 - \epsilon$. The rest of nodes except p form a chain in which each node is neighbor of its predecessor and successor in the chain. The chain forms a snake shape. Node p is a node that is within distance d of l, at the same level (coordinate y) of the last node in the chain (r in Figure 3.(b)), and connected with some node in the chain as shown. Especially, p is located at some position (x_p, y_p) where x_p is equal to $x_l + 2\delta_2 - \epsilon$ with $\epsilon \ll \delta_2$ and y_p is the same as the coordinate y of node r. Assume that nodes reach this configuration while previously been at distance δ_1 from each other. Thus at time t_0 strong connectivity holds. We usually refer to the location of the node only considering the x coordinate because it is the one of interest.

The number of nodes between any pair of nodes q_i and q_{i+1} as depicted in Figure 3 is fixed to $k = \lfloor \frac{\delta_2}{T v_{max}} \rfloor$. In this execution, at time t_0 all the nodes, except node p, start to move towards the left at the same speed $v = \frac{\delta_2}{T(k+1)} \leq v_{max}$. These values are chosen so that, in the execution we construct, q_i is at position x_l (see Figure 3.(b)) at time $t_0 + iT(k+1)$. When the last node of the chain is at a distance $\delta_2 - \epsilon$ at the left of p, the latter also starts moving to the left at the same speed. In our execution, we assume that a node that receives the information I immediately rebroadcasts it. In any other case the execution can be easily adapted by stopping the movement while I is not rebroadcasted. Then, in the execution, s broadcasts I at time t_0. We assume that each node in the chain receives I from its predecessor in T units of time. Then, q_i receives first I at time $t_0 + iT(k+1)$. Since at that time q_i is at x_l, no progress has been made to the right. Then, the geocast problem will be solved when all nodes in the chain have received I, and p received I from the last node in the chain. Since this implies $n-1$ transmissions and each takes T time, the total time to provide reliable delivery is $(n-1)T$. Finally, for all nodes except p strong connectivity holds throughout all the execution because they do not change their neighbourhood. At time t_0 strong connectivity holds for node

p because it is at distance ϵ from a node a_0 on its right. It is easy to see that strong connectivity holds throughout all the execution, since due to the movement pattern and speed, p will stop to be strong neighbour of node a_i only after it already becomes strong neighbour of node a_{i+1} for $i \geq 0$ (see Figure 3.(a)). The claim holds because p is at most at distance $\sqrt{[\frac{\delta_2}{Tv_{max}}(\delta_2 - \epsilon)]^2 + (2\delta_2 - \epsilon)^2} = \delta_2^2\sqrt{(\frac{1-\frac{\epsilon}{\delta_2}}{Tv_{max}})^2 + \frac{(2\delta_2-\epsilon)^2}{\delta_2^4}} < 3\delta_2^2$ from (x_l, y_l).

The bound of the above theorem depends on n. If n is finite this bound is finite. However, in a system with potentially infinite nodes, the geocast problem may never be solved.

Corollary 1. *No deterministic Geocast(I, d) algorithm will ensure the reliable delivery property in a system with infinite nodes, for $d \geq 3\delta_2^2$.*

5 Solving the Geocast Problem in One Dimension

In this section we explore the geocast problem when all nodes move along the same line. We show first that in order for the problem to be solvable, it is necessary that $T' > T/2$. Then, we present an algorithm that solves the problem efficiently if $T' > T$. These two results leave a gap $(T/2, T]$ of values of T', and hence an interval of maximum movement speed v_{max}, in which it is not known whether the problem can be solved.

Theorem 6. *No algorithm can solve the geocast problem in one dimension if $v_{max} \geq \frac{\delta_2 - \delta_1}{T}$, i.e. if $T' \leq \frac{T}{2}$.*

Proof. Consider a *Geocast*(I, d) primitive invoked at some time t by a source s, with $d \geq \delta_2$. We prove the claim by presenting an scenario in which, independently of the algorithm used, no node except the source delivers information I, while there are other nodes in the geocast region permanently. This violates the reliable delivery property and hence the geocast problem is not solved.

In our scenario there are three nodes, the source s and nodes p and q, that are permanently in the geocast region. Initially, node s is at a position l, from which it will never move. Node p is at position $l_1 = l + \delta_1$ (at distance δ_1 from s), and q is at distance δ_1 from p and at distance $2\delta_1$ from s. Then, q moves to reach the state s_{pq} depicted in Figure 4, which has the following properties: all nodes are located on a single line; the leftmost node is the source s located at position l; a node p is located at position l_1 at distance δ_1 from l; and a node q is located at position l_2 at distance 2δ from l_1. Observe that from the initial configuration up to state s_{pq} strong connectivity holds, and that nodes p and q are always within distance $\delta_2 \leq d$ of l.

If s never broadcasts I then neither p nor q deliver it, and reliable delivery is violated. Otherwise, assume that as a consequence of the *Geocast*(I, d) invocation, s invokes $broadcast(I)$ at times $t_0, t_1, ...$, with $t_{i+1} \geq t_i + T$. Let us define first the behavior of the nodes in interval $[t_0, t_1]$. At time t_0 nodes p and q start moving at their maximum speed v_{max} to exchange their positions. Then, at time $t'_0 = t_0 + 2T'$, p is located at l_2 and q is located at l_1 reaching state s_{qp}. They do not move from that state until t_1.

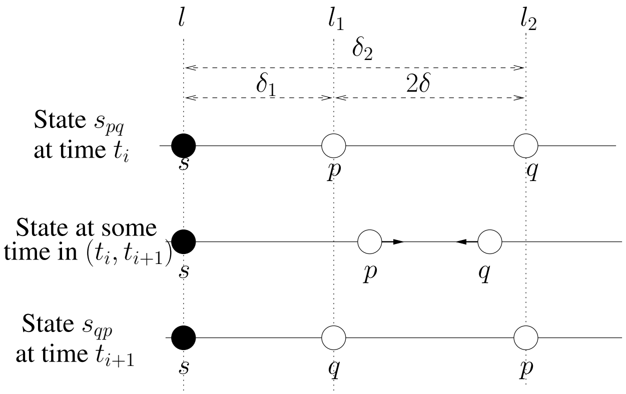

l
l_1
l_2
δ_2
δ_1
2δ
State s_{pq} at time t_i
s
p
q
State at some time in (t_i, t_{i+1})
s
p
q
State s_{qp} at time t_{i+1}
s
q
p

Fig. 4. Scenario for the proof of Theorem 6

Observe that strong connectivity has been preserved during the whole period $[t_0, t'_0]$: p and q never stop being strong neighbors, and the source is strong neighbor of p for all the period $[t_0, t'_0)$ and at time t'_0 it becomes strong neighbor of q. Note that neither p nor q have been neighbors of s during the whole period $[t_0, t_0 + T]$, because q is not a neighbor at time t_0 and p is not a neighbor at time $t'_0 \leq t_0 + T$. Hence, in our execution no node delivers I in $[t_0, t_1]$.

The behavior in interval $[t_1, t_2]$ is the same as described but exchanging the roles of p and q: the initial state is s_{qp}, at time t_1 they start moving to exchange positions, and at time t'_1 they end up at state s_{pq}. Again, I is not delivered at p nor q because they have not been neighbors of s in the whole period $[t_1, t_1 + T]$. For any interval $[t_i, t_{i+1}]$ the behavior is the same as in interval $[t_0, t_1]$, if i is even, and the same as in interval $[t_1, t_2]$ if i is odd. Then, in this scenario of execution only s delivers I and the reliable delivery property is not fulfilled.

This result shows that in order to solve the geocast problem it must hold that $T' > T/2$. In the following we prove that when all nodes move along the same line, the algorithm presented in Figure 2, for an appropriate value of M, efficiently solves the problem as long as $T' > T$ and $\delta_1 > \delta$. Let us first introduce some preliminary Observations and Lemmata which are instrumental for the proof of the main Theorem 13.

Assume that the source $s = q_0$ initiates a call of M-*Geocast*(I, d) at time $t = t_0$ from location $l = l_0$. Next, we prove that I propagates from l_0 towards the right of l_0. (For the left of l_0, the proof is symmetrical.) This happens in steps so that within a small period of time, I moves from a node, q_j to another node q_{j+1} at some large distance away.

Observation 7. *Let p be a node that receives information I at time t, then either p immediately rebroadcasts I or it exists a time $\tau \in [t, t + T]$ such that p broadcasts I both at time $\tau - T$ and at time τ.*

Observation 8. *If $T' > T$, $\frac{\delta T}{T'} < \delta$ is the maximum distance that a node can cover in time T.*

Observation 9. *Let p be a node that receives a message with information I at time t. p has delivered information I by time t.*

Hereafter, we denote $\Delta = \delta_1 + \delta$.

Lemma 4. *Let q_j be a node located at location l_j at time t_j. If $T' > T$ then every node that at time $t_j + T$ is within distance $\Delta = \delta_1 + \delta$ from l_j has been a neighbour of q_j throughout all the period $[t_j, t_j + T]$.*

Proof. At time t_j, q_j is located in l_j and it is a neighbor of all nodes at distance smaller than δ_2 from l_j. Let p be a node that at time $t_j + T$ is located within distance Δ from l_j. Let v be the maximum speed of nodes. Since $T' > T$, in T time a node can travel at most a distance $vT = \frac{\delta}{T'}T < \delta$. Thus at time t_j, p was located at l_p, within distance $\Delta + vT < \delta_2$ from l_j.

To break the connection with p after t_j, q_j has to travel in the opposite direction of p during $[t_j, t_j + T]$. Without loss of generality, assume $l_j \leq l_p$ and consider q_j moving towards the left and p to the right at full speed. At any time $t \in [t_j, t_j + T]$, q_j will be located at $l_j - vt$ and p will be at $l_p + vt$. Let l_p^T denote the location of p at time $t_j + T$, $l_p^T = l_p + vT$. Then, $l_p = l_p^T - vT$ and for all $t \in [t_j, t_j + T]$ $l_p^t = l_p + vt = l_p^T - vT + vt = l_p^T + v(t - T)$. So at time t the distance between q_j and p is $distance(p, q, t) = l_p^t - l_q^t = l_p^T + v(t - T) - (l_j - vt) = l_p^T - l_j + 2vt - vT \leq l_p^T - l_j + vt$. Since $l_p^T \leq l_j + \delta_1 + \delta$, $distance(p, q, t) \leq \delta_1 + \delta + vt < \delta_2$ because $vt < \delta$ for all $t \in [t_j, t_j + T]$.

Lemma 5. *Assume that a node q broadcasts information I at time t, being at location l. If $T' > T$, then every node that at time $t + T$ is within distance Δ from l will deliver the information by time $t + T$.*

Proof. By Lemma 4, every node p that at time $t + T$ is within distance Δ from l is a neighbor of q throughout all the period $[t, t + T]$. Thus if q broadcasts the information I at time t, p will deliver I by time $t + T$.

The following Lemma 6 states that if it exists a node that broadcasts information I at some time t, then by time $t + 3T$ there is another node far away from location l which broadcasts information I. Thus, these two nodes define a non-zero spatial interval and a temporal interval between two successive broadcasts events.

Lemma 6. *Assume that a node q_j broadcasts the information I at time t_j, located at point l_j. Let L_r denote the set of nodes that at time t_j are located on the right of $l_j + \delta_1$. If $\delta_1 \geq \delta$, $T' > T$ and $L_r \neq \emptyset$ then, assuming that $count_I < M$ for all nodes throughout $[t_j, t_{j+1}]$, either all the nodes in L_r deliver information I by time $t_j + T$, or there is a node q_{j+1} which broadcasts I at time t_{j+1} at location l_{j+1} such that:*

1. *$t_{j+1} - t_j \leq 3T$,*
2. *$l_{j+1} - l_j \geq \delta_1 - \frac{\delta T}{T'} > 0$*
3. *let $t = \min(t_j, t_{j+1} - T)$, throughout all the interval $[t, t_{j+1}]$, node q_{j+1} is a neighbor of another node q located on the left of $l_j + \Delta$ and which invoked $broadcast(I)$ at some time in $[t_{j+1} - T, t_{j+1}]$*

Proof. Assume that at time t_j, a node q_j broadcasts the information I being located at l_j. One of the following two cases holds:

- At time $t_j + T$, there is at least a node p located in the interval $[l_j + \delta_1, l_j + \Delta]$. Then, by Lemma 5, p will deliver the information I by time $t_j + T$. By Observation 7, p will broadcast I at some time $t_{j+1} \in [t_j, t_j + T]$. By Observation 8 and its position at time $t_j + T$, p will rebroadcast I at time $t_{j+1} \leq t_j + T$ being at location $l_{j+1} \geq l_j + \delta_1 - \frac{\delta T}{T'}$. The claim holds being $q_{j+1} = p$ and $q = q_j$.
- At time $t_j + T$, no node is located in the interval $[l_j + \delta_1, l_j + \Delta]$. Let L and L' respectively denote the set of nodes that at time $t_j + T$ are located on the left of $l_j + \delta_1$ and the ones that at time $t_j + T$ are located on the right of $l_j + \Delta$.
  If $L' = \emptyset$ then all the nodes that at time t_j were on the right of $l_j + \delta_1$ are within distance Δ from l_j at time $t_j + T$. By Lemma 5, these nodes deliver information I by time $t_j + T$.

  Otherwise, there must exist paths of strong neighbors from nodes in L' to node on the left of $l_j + \delta_1$. In particular, nodes in L' can be connected with nodes in L at most within distance δ on the left of l_j. These latter have delivered the information I by time $t_j + T$. One of the following cases has to hold:

  1. It exists at least a connection between a node p in L' and a node q in L which lasts throughout $[t_j, t_j + 2T]$. Then p will deliver the information I at some time $t \in [t_j, t_j + 2T]$. Note that at time $t_j + T$, p is on the right of $l_j + \Delta$ and, since $T' > T$, it is on the right or on $l_j + \Delta - \frac{\delta T}{T'} > l_j + \delta_1 - \frac{\delta T}{T'}$ throughout all the period $[t_j + T, t_j + 2T]$. Then by Observation 7, p will broadcast information I at some time $t_{j+1} \in [t_j + T, t_j + 2T]$, being located at some position $l_{j+1} > l_j + \delta_1 - \frac{\delta T}{T'}$. The claim holds being $q_{j+1} = p$ and by the fact that p and q are neighbors throughout $[\tau, t_{j+1}] \subset [t_j, t_j + 2T]$, where $\tau = \min\{t_{j+1} - T, t_j\}$.
  2. Each connection between nodes in L' and nodes in L breaks at some time in $[t_j, t_j + 2T]$. Then, a new strong connection has to be created at some time $t \in [t_j, t_j + 2T]$ before all such connections break. Otherwise strong connectivity is violated.

     Let p and q be respectively the node in L and the node in L' that create the new strong connection at time t, i.e. $distance(p, q, t) \leq \delta_1$. By Lemma 4, p and q have been neighbors throughout $[t - T, t + T]$. If $t \in [t_j, t_j + T]$, $[t_j, t_j + T] \subset [t - T, t + T]$ and all such connections have to break at some point in $[t_j + T, t_j + 2T]$, since otherwise it will exist at least a connection between a node in L and a node in L' that lasts throughout all the period $[t_j, t_j + 2T]$ and thus we reach a contradiction.

     Then, a new connection between a node p in L and a node q in L' has to be created at some time $t \in [t_j + T, t_j + 2T]$. At time t, $distance(p, q, t) \leq \delta_1$, and since in $2T$ time a node can travel at most a distance $\frac{2\delta T}{T'}$, at time t_j q was on the right of $l_j - 2\delta$. Thus q delivers information I by time $t_j + T$, and q broadcasts I both at time τ and $\tau' = \tau + T$ with $\tau \in [t_j, t_j + T]$. By Lemma 1, p and q are neighbors throughout all the period $[t - T, t + T]$ with $t \in [t_j + T, t_j + 2T]$. Either τ or τ' is in the interval $[t - T, T]$, then p delivers information I by time $t + T \leq t_j + 3T$. Then, either p immediately broadcasts I or it broadcasted

I at some time in $[t, t+T]$. Either way, p broadcasts information I at time $t_{j+1} \leq t_j + 3T$ being at some location $l_{j+1} > l_j + \delta_1 + \delta - \frac{2\delta T}{T'} > l_j + \delta_1 - \frac{\delta T}{T'}$. The claim holds being $q_{j+1} = p$ and by the fact that throughout $[t_{j+1} - T, t_{j+1}] \subset [t - T, t + T]$ p and q are neighbors.

Observation 10. *Let q, q', and p be three nodes that at time t are respectively located at l_q, $l_{q'}$, and l_p, such that $l_p < l_q$ and $l_p < l_{q'}$. Assume that q delivers information I by time $t + T$ because p invoked a call of broadcast(I) at time t. If q' is between q and p throughout $[t, t + T]$, q' delivers information I by time $t + T$.*

Definition 2. *t_j is a time at which a node q_j invokes the $broadcast(I)$ being located at location l_j such that $t_{j+1} - t_j \leq 3T$ and $l_{j+1} - l_j \geq \delta_1 - \frac{\delta T}{T'}$, for $j \in \{0, 1, \ldots, i\}$.*

The following Lemmas 7, 8 and 9 are instrumental to prove Lemma 10. This latter states that any node that traverses any of the spatial intervals defined by two consecutive broadcast events (the ones defines in Definition 2) during the corresponding broadcast period, deliver the information by a given time.

Lemma 7. *Let $t, t' \in [t_j, t_{j+1}]$ with $t' > t$. Let p be a node that at time t is on the left of l_j and at time t' is located inside the interval $[l_j, l_{j+1}]$. If $\delta_1 > \delta$, p receives a message with information I by time $t_j + T$.*

Proof. By Definition 2, $t_{j+1} - t_j \leq 3T$. Let $t, t' \in [t_j, t_{j+1}]$ with $t' > t$. Let p be a node that at time t is on the left of l_j and at time t' is located inside the interval $[l_j, l_{j+1}]$. If $\delta_1 > \delta$, at time t_j p is at most within distance $3\delta < \delta_2$ on the left of l_j. Then, at time t_j, q_j and p are neighbors and they will remain neighbor at least up to $t_j + T$. This is because in the worst case p reaches position l_j immediately after t_j but then at time $t_j + T$ the distance between p and q_j is at most 2δ. Otherwise they move towards each other getting closer. So p will receive a message with information I by time $t_j + T$.

Lemma 8. *Let $t, t' \in [t_{j+1} - T, t_{j+1}]$ with $t' > t$. Let p be a node that at time t is on the right of location l_{j+1}. If $\delta_1 > \delta$, p receives a message with information I by time $t_{j+1} + T$.*

Proof. Let $t, t' \in [t_{j+1} - T, t_{j+1}]$ with $t' > t$. Let p be a node that at time t is on the right of location l_{j+1}. If at some time $t' \in [t_{j+1} - T, t_{j+1}]$ p is on the left of l_{j+1}, p is neighbor of q_{j+1} throughout all the period $[t_{j+1}, t_{j+1} + T]$. This is because $\delta_1 > \delta$ and at time t_{j+1}, p is on the right of $l_{j+1} - \delta$ and in T times the distance between p and q_{j+1} increases less than 2δ. Then p will receive a message with information I by time $t_{j+1} + T$.

Lemma 9. *Let p be a node that at time $t \in [t_j, t_{j+1}]$ is on the right of l_{j+1}. If at some time $t' \in [t_j, t_{j+1}]$ with $t < t'$, p is located at $l_p \in [l_j, l_{j+1}]$ and it does not exist a time $t'' \in [t_j, t_{j+1}]$ with $t'' > t'$ such that p is not on the right of l_{j+1}, p delivers the information I by time $t_{j+1} + 2T$.*

Proof. Consider a node p that at time $t \in [t_j, t_{j+1}]$ is located at the right of location l_{j+1}. Assume that at time $t' \in [t_j, t_{j+1}]$, with $t' > t$, p is located inside the interval

$[l_j, l_{j+1}]$ and it does not exist $t'' \in [t_j, t_{j+1}]$ with $t'' > t'$ such that p is on the right of l_{j+1} at t''.

If $t' \in [t_{j+1} - T, t_{j+1}]$, the claim follows by Lemma 8 and Observation 9. Then, consider $t' \in [t_j, t_{j+1} - T)$. $[t_j, t_{j+1} - T) \subseteq [t_j, t_j + 2T)$, then by Observation 8, at time $t_{j+1} - T$, p is on the right of $l_{j+1} - 2\delta$. At the same time $t_{j+1} - T$, q_{j+1} is at most within distance $\frac{\delta T}{T'}$ from l_{j+1}, since it has to be located at l_{j+1} at time t_{j+1}. Then, since $3\delta < \delta_2$, at time $t_{j+1} - T$ p and q_{j+1} are neighbors. p and q_{j+1} remain neighbors throughout all the period $[t_{j+1} - T, t_{j+1}]$ because at time t_{j+1}, p is at most within distance 3δ from l_{j+1}, due to Lemma 6.(1) and Observation 8.

By Lemma 6 third bullet, either q_{j+1} receives I at time t_{j+1} because a node q that received the information directly by q_j invoked $broadcast(I)$ at some time $\tau \in [t_{j+1} - T, t_{j+1}]$ or q_{j+1} invoked $broadcast(I)$ also at time $t_{j+1} - T$. In this last case, the claim holds because p and q_{j+1} are neighbors throughout all the period $[t_{j+1} - T, t_{j+1}]$ and because of Observation 9. Then, consider the case where q_{j+1} receives I at time t_{j+1} because a node q invoked $broadcast(I)$ at some time $\tau \in [t_{j+1} - T, t_{j+1}]$.

If at time $t_{j+1} + T$ node p is within distance Δ from l_{j+1} then by Lemma 5 p deliver the message by time $t_{j+1} + T$. Otherwise, the location of p at time $t_{j+1} + T$ is on the left of $l_{j+1} - \Delta$. This implies that the location of p at time t_{j+1} is minor or equal to $l_{j+1} + \Delta - \frac{\delta T}{T'}$. Then, at time t_{j+1}, p and q are neighbors because distance(q, q_{j+1},t_{j+1})$< \delta_2$ and distance(p, q_{j+1},t_{j+1})$\geq \Delta - \frac{\delta T}{T'}$.

Note that at time t_j, p is on the right of l_j. Then, by Observation 10, either p delivers the information by time $t_j + T$ or at some point $t \in [t_j, t_j + T]$ p is located on the right of q. Note that q will broadcast the information once in each time interval $[t_j + kT, t_j + (k+1)T]$ with $k \in \{0, \ldots, 3\}$. So either there is a time in $[t_j, t_{j+1}]$ where p and q are strong neighbors and then p delivers the information by time $t_{j+1} + T$, or at time t_{j+1} q is on the left of p and this latter is on the left of q_{j+1}. Then, p will deliver information I by time $t_{j+1} + 2T$ because of a call of broadcast either at q_{j+1} or at q. This is because either p remains neighbors of q or of q_{j+1} throughout all the interval $[t_{j+1} - T, t_{j+1} + T]$ or at time t_{j+1} p and q are within distance greater than δ_1 from each other and they move towards or in the same direction of q. So they do not disconnect for at least other $2T$.

Lemma 10. *Let p be a node that at some time $t \in [t_j, t_{j+1}]$ is in some location $l_p \in [l_j, l_{j+1}]$. If it does not exist a time $t' \in [t_j, t_{j+1}]$ with $t' > t$ such that p is not on the right of l_{j+1}, p delivers the information I by time $t_{j+1} + 2T$.*

Proof. Let p be a node that at some time $t \in [t_j, t_{j+1}]$ is located at $l_p \in [l_j, l_{j+1}]$. Assume that it does not exist a time $t' \in [t_j, t_{j+1}]$ with $t' > t$ such that p is not on the right of l_{j+1}. Then if at time t_j p is either on the left of l_j or on the right of l_{j+1}, then the claim follows respectively by Lemma 7 and Observation 9, or by Lemma 9. Finally, consider the case where node p is inside the interval $[l_j, l_{j+1}]$ throughout all the interval $[t_j, t_{j+1}]$. We prove that p delivers information I by time $t_{j+1} + T$. If at time $t_j + T$ p is within distance Δ from l_j, p delivers the information I by time $t_j + T$, because of Lemma 11. Then assume that p is located in the interval $[l_j + \Delta, l_{j+1}]$ at time $t_j + T$. At that time q is located on the left of location $l_j + \Delta$. At time $[t_{j+1} - T, t_{j+1}]$ q and q_{j+1} are neighbors because of the third bullet of Lemma 6. At time $t_{j+1} - T$ one of the following cases will happens: (1) p is in between of q and q_{j+1}, (2) p is on the right of both these nodes but on the left of l_{j+1} or (3) p is on the left of both q and q_{j+1}. But this means that p is a neighbor of q throughout $[t_{j+1}, t_{j+1} + 2T]$ or is a neighbor of q_{j+1} throughout $[t_{j+1} - T, t_{j+1}]$. Since q broadcasts I once in each time interval $[t_j + kT, t_j + (k + 1)T]$ with $k \in \{0, \dots, 3\}$ and q_{j+1} broadcasts at time t_{j+1}, p delivers I by time $t_{j+1} + 2T$ and the claim holds.

Now we prove that if a node stays within distance d from the location where the geocast has been invoked, throughout all the geocast period, then it is eventually inside one of the intervals between two consecutive broadcasts at the right time and for long enough to deliver the information I.

Lemma 11. *If a node q stays within distance d from l throughout $[t_0, t_{i+1}]$ for i such that $l + d \in [l_i, l_{i+1}]$, then q delivers the information I by time $t_{i+1} + 2T$.*

Proof. Let t_0 be the time when the source node s performs the first $broadcast(m)$ because of a call of $M\text{-}Geocast(I, d)$. If q is located at $l_0(= l)$ at time t_0 then the lemma holds. Otherwise, without loss of generality, let q be located on the right of s at time t_0. For every time in $[t_0, t_{i+1}]$ q is located either on or on the left of l_{i+1} because $l + d \leq l_{i+1}$.

By induction on j, it is easy to see that it exists a $j \leq i$ such that at time $t \in [t_j, t_{j+1}]$ q is in the interval $[l_j, l_{j+1}]$ and it does not exist a time $t' \in [t_j, t_{j+1}]$ with $t' > t$ such that q is on the right of l_{j+1}. Otherwise at time t_{j+1} q is on the right of l_{j+1}, and for $j = i$ we have that at time t_{i+1} q is on the right of l_{i+1}. This means that at time t_{i+1} q is at distance greater than d from l. By the Lemma 10, q will deliver the information I by time $t_{i+1} + 2T$.

Observation 11. *Let $count_I$ be the counter associated to the communication generated by a call of $M\text{-}Geocast(I, d)$. $count_I$ is set to zero once when the source invokes the first $broadcast(I)$ at time t_0 and it is never reset.*

Observation 12. *Let p be a node different from the source node. p invokes $broadcast(I)$ at some time t only if it has generated a $receive(I)$ event at some time before t.*

Lemma 12. *Let t be the time when a call of $M\text{-}Geocast(I, d)$ is invoked. Every message broadcast or received at some time in $[t, t + k]$ has counter at most equal to k.*

Proof. The proof is by induction on k. For $k = 0$, we have to consider the time t. At that time only the source node invokes a $broadcast(I)$ and the counter of the broadcasted message has value 0 (Line 9 of Figure 2). Then the claim holds. By inductive hypothesis, assume that every message broadcast or received at some time in $[t, t+k]$ has counter at most equal to k. Then, we prove that every message broadcast or received at some time in $[t, t + k + 1]$ has counter at most equal to $k + 1$. We know that this cannot happen by time $t + k$ because of the inductive hypothesis. Then, by contradiction assume that it exists a message that is received at time $t + k + 1$ and whose counter has value greater than $k + 1$. But since it takes at least 1 time unit to receive a message, this means that the message received at time $t+k+1$ was broadcast at the latest at time $t+k$. But then if the message has counter $k+1$ we contradict the inductive hypothesis.

Finally, consider the case where at time $t+k+1$ a message m is broadcast by a node p. p increments its counter possibly each time it receives a message or when it broadcast a message. But by time $t+k$ all the messages received by p have counter smaller or equal to k, and p may have broadcast at most k messages. So at time $t+k$ the counter of p is at most k. Then, when at time $t+k+1$ it broadcasts a message, this message has a counter at most $k+1$. Then the claim follows.

Finally we define the bound for the time to ensure the reliable delivery property and the termination property. From the latter, we obtain the bound for the integrity property.

Theorem 13. *If $T' > T$, the M-Geocast(I,d) algorithm with $M = 3T(i+1)+2T$ and $i = \lfloor \frac{d}{\delta_1 - \frac{\delta T}{T'}} \rfloor$ ensures*
(1) the Reliable Delivery Property 1 for $C = 3T(i+1)+2T$,
(2) the Termination Property 2 for $C' = (3T(i+1)+2T+1)T$, and
(3) the Integrity Property 3 for $d' = (C'+T)(\delta_2 + \frac{\delta}{T'})$.

Proof. Let us first prove (1). From Lemma 6, we know that any $3T$ rounds starting from $t_0 = t$ the information reaches some distance $\delta_1 - \frac{\delta T}{T'}$ farther from l. Formally, $l_i - l \geq i(\delta_1 - \delta T/T')$. Since we want all the nodes that during the geocast interval remain within distance d from l deliver the information I, we need to compute the maximum value that i could take in any execution such that $(l+d) \in [l_i, l_{i+1})$. Then $i \leq \lfloor \frac{l_i - l}{\delta_1 - \frac{\delta T}{T'}} \rfloor$ and because $l_i - l \leq d$, $i \leq \lfloor \frac{d}{\delta_1 - \frac{\delta T}{T'}} \rfloor$.

From Lemma 11, all the nodes that remain within distance d from $l(= l_0)$ throughout $[t_0, t_{i+1}]$ deliver I by time $t_{i+1} + T = t + C$. By Lemma 6, $t_{i+1} - t \leq 3T(i+1)$, and $C = t_{i+1} - t + T \leq 3T(i+1) + 2T$. Then $C \leq 3T(\lfloor \frac{d}{\delta_1 - \frac{\delta T}{T'}} \rfloor + 1) + 2T$.

We have finally to prove that, during $[t, t+C]$, for any node, $count_I < M$, where $M = C$. This follows from Lemma 12.

We prove now (2). Every message received causes rebroadcasting of I in a message with counter at least incremented by one. This will happen at least once every T times. Termination happens after any message received has counter larger than $3T(i+1)+2T$, where $i = \lfloor \frac{d}{\delta_1 - \frac{\delta T}{T'}} \rfloor$. This happens within $(3T(i+1)+2T+1)T + T$ time, because all messages broadcast after time $(3T(i+1)+2T+1)T$ have counters at least equal to $3T(i+1)+2T+1$ and all such messages are received within at most another T times. Note that, in the worst case, each broadcast message is received exactly after T times and then the counter is incremented by one unit, while in reality T steps have passed. Therefore, $C' = (3T(\lfloor \frac{d}{\delta_1 - \frac{\delta T}{T'}} \rfloor + 1) + 2T + 1)T$.

Finally, we prove (3). A broadcast message will be received at least after one time unit during which any node can traverse distance at most $\frac{\delta}{T'}$. Therefore, if a node broadcasts a message from location l' at time t', then its neighbors receive it the earliest at time $t'+1$, when at distance less than $\delta_2 + \frac{\delta}{T'}$ away from l'. Then, if the source starts M-*Geocast*(I,d) at time t from location l, at time $t+m$, the furthest node that delivers I is at distance less than $m(\delta_2 + \frac{\delta}{T'})$ away from l. By (2), after time $t+C'$, no node broadcasts messages with information I. Therefore, no node delivers I after time $t + C' + T$. But at time $t + C' + T$, all nodes that have delivered I are within distance less than $(C' + T)(\delta_2 + \frac{\delta}{T'})$ from l. Therefore, if a node remains further than $d' = (C' + T)(\delta_2 + \frac{\delta}{T'})$ from l, it will never deliver I.

6 Conclusion

We have studied the geocast problem in mobile ad-hoc networks. We have considered a set of n mobile nodes which move in a continuous manner with bounded maximum speed. We have addressed the question of how the speed of movement impacts on providing a deterministic reliable geocast solution, assuming that it takes some time T to ensure a successful one-hop radio communication.

Our results improve and generalize the bounds presented in [1]. For the two-dimensional mobility model, we have presented a tight bound on the maximum speed of movement that keeps the solvability of geocast. We have also proved that $\Omega(nT)$ is a time complexity lower bound for a geocast algorithm to ensure deterministic reliable delivery, and we have provided a distributed solution which is proved to be asymptotically optimal in time. This latter bound confirms the intuition, presented in [15] for the brodcast problem by Prakash et al., that when nodes may move the number of nodes in the system is the impact factor on the reliable communication completion time. In fact, our solution and bounds are also applicable to 3 dimensions, a case that is rarely studied but may be of growing interest.

Finally, assuming the one-dimensional mobility model, i.e. nodes moving on a line, we have proved that $v_{max} < \frac{2\delta}{T}$ is a necessary condition to solve the geocast, where δ is a system parameter, and presented an efficient algorithm when $v_{max} < \frac{\delta}{T}$. This still leaves a gap on the maximum speed to solve the geocast problem in one dimension.

References

1. Baldoni, R., Ioannidou, K., Milani, A.: Mobility Versus the Cost of Geocasting in Mobile Ad-Hoc Networks. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 48–62. Springer, Heidelberg (2007)
2. Bruschi, D., Del Pinto, M.: Lower bounds for the broadcast problem in mobile radio networks. Distributed Computing 10(3), 129–135 (1997)
3. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit Disk Graphs. Discrete Mathematics 86(1-3), 165–177 (1990)
4. Jinag, X., Camp, T.: A review of geocasting protocols for a mobile ad hoc network. In: Proceedings of Grace Hopper Celebration (2002)
5. Dolev, S., Gilbert, S., Lynch, N.A., Shvartsman, A.A., Welch, J.: Geoquorums: Implementing atomic memory in mobile ad hoc networks. Distributed Computing 18(2), 125–155 (2005)
6. Chlebus, B.S., Gasieniec, L., Gibbsons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Distributed Computing 15(1), 27–38 (2002)
7. Ko, Y.-B., Vaidya, N.H.: Geocasting in mobile ad-hoc networks: Location-based multicast algorithms. In: Proceedings of IEEE WMCSA, NewOrleans, LA (1999)
8. Ko, Y., Vaidya, N.H.: Geotora: a protocol for geocasting in mobile ad hoc networks. In: Proceedings of the 8th International Conference on Network Protocols (ICNP), p. 240. IEEE Computer Society, Los Alamitos (2000)

9. Ko, Y., Vaidya, N.H.: Flooding-based geocasting protocols for mobile ad hoc networks. Mobile Network and Application 7(6), 471–480 (2002)
10. Gupta, S.K.S., Srimani, P.K.: An adaptive protocol for reliable multicast in mobile multi-hop radio networks. In: Proceedings of the 2nd Workshop on Mobile Computing Systems and Applications (WMCSA), p. 111. IEEE Computer Society, Los Alamitos (1999)
11. Imielinski, T., Navas, J.C.: Gps-based geographic addressing, routing, and resource discovery. Communication of the ACM 42(4), 86–92 (1999)
12. Liao, W., Tseng, Y., Lo, K., Sheu, J.: Geogrid: A geocasting protocol for mobile ad hoc networks based on grid. Journal of Internet Technology 1(2), 23–32 (2001)
13. Mohsin, M., Cavin, D., Sasson, Y., Prakash, R., Schiper, A.: Reliable broadcast in wireless mobile ad hoc networks. In: Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS), p. 233.1. IEEE Computer Society, Los Alamitos (2006)
14. Navas, J.C., Imielinski, T.: Geocast: geographic addressing and routing. In: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), pp. 66–76. ACM Press, New York (1997)
15. Prakash, R., Schiper, A., Mohsin, M., Cavin, D., Sasson, Y.: A lower bound for broadcasting in mobile ad hoc networks. Ecole Polytechnique Federale de Lausanne, Tech. Rep. IC/2004/37 (2004)
16. Pagani, E., Rossi, G.P.: Reliable broadcast in mobile multihop packet networks. In: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom), pp. 34–42. ACM Press, New York (1997)

Degree 3 Suffices: A Large-Scale Overlay for P2P Networks*

Marcin Bienkowski[1,**], André Brinkmann[2,***], and Miroslaw Korzeniowski[3,†]

[1] University of Wroclaw, Poland
[2] University of Paderborn, Germany
[3] Wroclaw University of Technology, Poland

Abstract. Most peer-to-peer (P2P) networks proposed until now have either logarithmic degree and logarithmic dilation or constant degree and logarithmic dilation. In the latter case (which is optimal up to constant factors), the constant degree is achieved either in expectation or with high probability. We propose the first overlay network, called *SkewCCC*, with a maximum degree of 3 (minimum possible) and logarithmic dilation. Our approach can be viewed as a decentralized and distorted version of a Cube Connected Cycles network. Additionally, basic network operations such as join and leave take logarithmic time and are very simple to implement, which makes our construction viable in fields other than P2P networks. A very good example is scatternet construction for Bluetooth devices, in which case it is crucial to keep the degree at most 7.

1 Introduction

Peer-to-peer networks have become an established paradigm of distributed computing and data storage. One of the main issues tackled in this research area is building an overlay network that provides a sparse set of connections for communication between all node pairs. The aim is to build the network in a way that an underlying routing scheme is able to quickly reach any node from any other, without maintaining a complete graph of connections. In this paper, we investigate such networks suitable not only for peer-to-peer networks but also for Bluetooth scatternet formation and one-hop radio networks.

An important property of the investigated networks is their scalability. We introduce a scalable and dynamic network structure which we call *SkewCCC*. The maximum in- and out-degree of a node inside the SkewCCC network is 3 and routing and lookup times are logarithmic in the current number of nodes. Naturally, it is impossible to decrease the degree to 2 while preserving any network topology besides a ring. Our routing scheme is *name-driven*, i.e. packet

[*] This work has been partially supported by EU Commission COST 295 Action "DYNAMO — Foundations and Algorithms for Dynamic Networks".

[**] Supported by MNiSW grant number N206 001 31/0436, 2006-2008.

[***] Partially supported by the EU within the 6th Framework Programme under contract IST-2005-034891 *Hydra*.

[†] Supported by MNiSW grant number PBZ/MNiSW/07/2006/46.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 184–196, 2008.

routes can be calculated based on their destination address without requiring extensive routing tables [8].

The construction of the SkewCCC network is fully distributed and remains very simple from the computational and communication perspective. This is crucial when designing algorithms for weak devices like small embedded systems, which are often found in sensor networks. To minimize the production costs such devices have very limited computational power and have to reduce their energy usage [6].

It is widely known in the area of P2P networks that if we use a ring-based network such as Chord [12] as a basis for a distributed hash table (DHT), then the load balance is not even, unless special algorithms are employed in order to smoothen the load. In particular, if nodes choose random places for themselves, the expected ratio of the highest loaded to the lowest loaded node is $\Omega(n \log n)$. Our design applied as a topology for a DHT is perfectly balanced, i.e. without applying any additional load balancing schemes each node has an in- and out-degree independent of the network size.

Besides being used as an overlay sensor network or as a peer-to-peer network, the proposed SkewCCC architecture can also be applied for underlay networks. In particular, for the construction of Bluetooth scatternets and one-hop radio networks, it is not sufficient to ensure that the (expected) network degree is bounded by *some* constant, but it is absolutely necessary to keep the degree smaller than 7.

2 Related Work

The current research in the area of P2P networks concentrates on providing dynamic overlay networks with good properties (small diameter, small in- and out-degree). The basic task of peer-to-peer applications, i.e. efficiently locating the peer that stores a sought data item, is performed by means of consistent hashing [7]. The main requirement of this technique is the ability to perform a name-driven routing in the network; the parameters of this routing, like dilation (max. path length) influences directly the performance of the whole system. In this section, we compare our solution with existing dynamic networks. By n we denote the number of nodes currently in the system.

The first proposed distributed hash table (DHT) solutions were ring topologies: Chord [12] and Chord-like structures: Tapestry [13] and Pastry [11]. By introducing shortcuts inside the ring, these DHTs achieve good properties: their dilation is $O(\log n)$ and also the runtime of joining and leaving the network is logarithmic. On the other hand, each node has to keep $\Theta(\log n)$ pointers to other nodes and thus the out-degree is large.

SkipNets [5] and Skip-Graphs [1] as well their deterministic versions such as Hyperrings [2] are based on a hierarchical structure. In this approach the network is organized into levels, where the first level is just a ring of all nodes and higher levels are built of independent rings being refinements of rings from previous levels. The refinement continues until each node itself is a ring. The approach yields good properties concerning storage of sorted or unevenly distributed data but does not improve the degree/distance factor of Chord. In all mentioned designs, the degree of each node is logarithmic (as there are logarithmic number of levels and each node has degree two in each level) and logarithmic dilation (one step has to be done in each ring).

The first overlay network with constant out-degree and logarithmic diameter was Viceroy [9]. The network is based on the randomized approximation of a butterfly network [8] that adds a constant number of outgoing links for long range communications. Furthermore, join or leave operations inside Viceroy require only a constant number of local updates on expectation and a logarithmic number with high probability. Nevertheless, the in-degree of at least one node inside the network becomes with high probability $\Omega(\log n / \log\log n)$ if each node has only a single choice for its address during the join process. The in-degree can also be bounded with high probability if each node has $\Theta(\log n)$ random choices for its address during the join operation. However this leads to $\Theta(\log^2 n)$ updates for each join operation.

The approach of using multiple choices is also used inside the Distance Halving DHT network [10]. The structure is based on a dynamic decomposition of a continuous space into cells, which are assigned to nodes. The underlying deBruijn structure ensures a degree of d for each node and leads to a path length of $O(\log_d n)$ for key lookups.

Besides providing solutions for general networks, there has been some research on overlay networks which consider the special properties of Bluetooth networks. The first scalable overlay for Bluetooth (a network of constant degree and poly-logarithmic diameter) has been presented in [3]. The network is based on a backbone that enables routing based on virtual labeling of nodes without large routing tables or complicated path-discovery methods. The scheme is fully distributed, but still poses high demands on the computational abilities of the underlying devices.

In [4], we have presented an overlay topology with special support for Bluetooth networks which is based on Cube Connected Cycles networks [8]. The resulting network has constant in- and out-degree as well as a dilation of $O(\log n)$. The main drawback of the approach is that the scheme is centralized (in particular, each node has to know the current number of nodes in the system) and hence not scalable. Although the scheme we present in this paper is also based on the CCC network, it is a big step forward as it is completely distributed and self-balancing.

In Fig. 1, we summarize the parameters of different distributed solutions, most of them holding with high probability.

3 SkewCCC

As we base our approach on the hypercube and the CCC (Cube Connected Cycles) networks, we shortly review their construction. These networks were extensively studied and have good properties concerning maintenance, diameter,

Network	out-degree	in-degree	dilation	join runtime
Chord-like networks	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Viceroy (single choice)	$O(1)$	$\Omega(\frac{\log n}{\log\log n})$	$\Theta(\log n)$	$O(\log n)$
Viceroy (multiple choice)	$O(1)$	$O(1)$	$\Theta(\log n)$	$\Omega(\log^2 n)$
Distance Halving (m.c.)	$O(d)$	$O(d)$	$\Theta(\log_d n)$	$\Omega(\log n \cdot \log_d n)$
SkewCCC	3	3	$\Theta(\log n)$	$\Theta(\log n)$

Fig. 1. Summary of old and new results

degree of nodes and routing speed. On the other hand, they are meant exclusively for static networks. For a thorough introduction to these kinds of networks, we refer the reader to [8].

In this paper, we use the following notation. By a string we always mean a binary string, whose bits are numbered from 0. For any two strings a and b, we write $a \sqsubseteq b$ to denote that a is a (not necessarily proper) prefix of b. We denote an empty string by ϵ and use $\cdot$ to denote a concatenation of two strings; $\oplus$ denotes the bitwise xor operation. We also identify strings of fixed lengths with binary numbers they represent.

Definition 1. *The d-dimensional hypercube network has $n = 2^d$ nodes. Each node is represented by a number $0 \leq i < n$. Two nodes i and j are connected if and only if $i \oplus 2^k = j$ for an integer $0 \leq k < d$.*

A d-dimensional Cube Connected Cycles (CCC) network is essentially a d-dimensional hypercube in which each node is replaced with a ring of length d and each of its d connections is assigned to one of the ring nodes. This way the degree of the network is reduced from d to 3, whereas almost all of the network properties (e.g. diameter) are changed only slightly.

Definition 2. *The d-dimensional CCC network has $d \cdot 2^d$ nodes. Each node is represented by a pair (i, j), where $0 \leq i < d$ and $0 \leq j < 2^d$. Each such node is connected to three neighbors: two cycle ones with indices $((i \pm 1) \bmod d, j)$ and a hypercubic one $(i, j \oplus 2^i)$.*

Examples of a 3-dimensional hypercube and CCC network are given in Fig. 2.

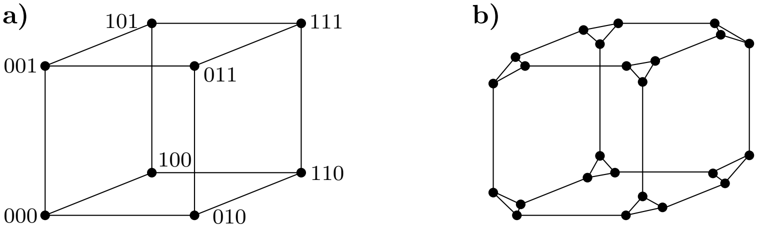


Fig. 2. a) 3-dimensional hypercube, b) 3-dimensional CCC

We present our network in two stages. First, we describe the network and its properties. Second, we describe algorithms, which assure proper structure when nodes join and leave the network.

3.1 Network Structure

When we compare the CCC network with the hypercube, we may think that the d-dimensional hypercube is a *skeleton* for the d-dimensional CCC. In other words, if we replace each node of the hypercube (to which we refer as a *corner*) by a cycle of nodes, then the resulting network is a CCC. In the following description we start from a description of such a skeleton for our network, called *SkewHypercube*, then we show how to replace each of its corners by a ring of real network nodes, finally creating a structure, which we call a *SkewCCC* network.

SkewHypercube. In the following we describe a skeleton network called *skewHypercube*. Each node of this network will correspond to a group of real nodes. To avoid ambiguity, we refer to a skeleton node as a *corner*.

First, we define the set of corners. Each corner i has an identifier, which is a string s_i of length d_i. The number d_i is called the *dimension* of the corner. We require that the set of corner identifiers $\mathcal{C} = \{s_i\}$ is *prefix-free* and *complete*. Prefix-freeness means that for any two strings s_i, s_j, neither $s_i \sqsubseteq s_j$ nor $s_j \sqsubseteq s_i$. Completeness means that for any infinite string s, there exists an identifier $s_i \in \mathcal{C}$, s.t. $s_i \sqsubseteq s$. The description above implies that (i) a single corner with empty name $s = \epsilon$ constitutes a correct set $\mathcal{C}$ and (ii) any correct set of corners $\mathcal{C}$ can be obtained by multiple use of the following operation (starting from the set $\{\epsilon\}$): take a corner i and replace it with two corners j and k with identifiers $s_j = s_i \cdot 0$ and $s_k = s_i \cdot 1$.

Second, we define the set of edges in a SkewHypercube.

Definition 3. *Two SkewHypercube corners i and j are connected iff*

(i) there exists $0 \leq k_i < d_i$, s.t. $s_i \oplus 2^{k_i} \sqsubseteq s_j$ or
(ii) there exists $0 \leq k_j < d_j$, s.t. $s_j \oplus 2^{k_j} \sqsubseteq s_i$.

We note that if all identifiers of corners have the same length d, then our SkewHypercube is just a regular d-dimensional hypercube. On the other hand, the definition above allows the following situation to occur. It may happen that a single corner s has identifier 0 (dimension 1) and there are 2^k corners with dimension $k + 1$ with identifiers starting with 1. This results in corner s having the degree of 2^k. We will explicitly forbid such situations in the construction of our network and require that the dimensions of neighboring corners can differ at most by 1. This ensures that each corner of dimension d has at most $2d$ neighbors. An example of a SkewHypercube is presented in Fig. 3.

Identifiers. To specify which nodes are stored in particular corners of the SkewHypercube, we have to give nodes unique identifiers. These identifiers are infinite strings, chosen randomly upon joining the network, where each bit is

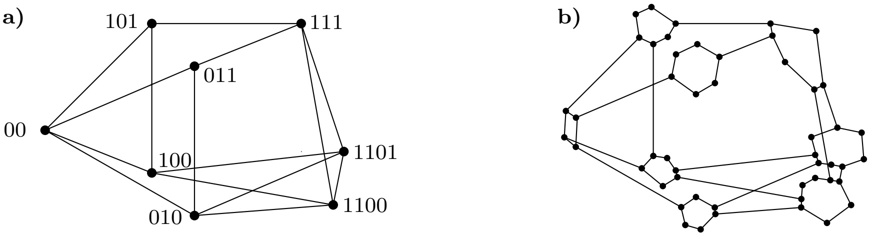


Fig. 3. a) SkewHypercube skeleton, b) SkewCCC, only core nodes are depicted

equiprobably 0 or 1. To avoid the burden of handling infinite strings, one may follow the approach of SkipGraphs [1], i.e. a node chooses for an identifier a string of a fixed length, and when two nodes with the same identifier meet, they choose additional random bits until their identifiers differ on at least one bit. Moreover, it will follow from our construction that such an *identifier conflict* will be detected right after a node joins the network. In practical applications, it is sufficient to choose identifiers of length 160 bits, as in such case the probability of a conflict is overwhelmingly low.

The identifier of a node decides where the node should be placed in the network and constitutes its address. Namely, a node with identifier x is stored in a corner s, s.t. $s \sqsubseteq x$. It remains to show how nodes are connected within a corner and how neighbor relations between corners are represented by real edges between nodes.

From Skeleton to SkewCCC. As mentioned previously, a corner identified by a string s_i contains a group of nodes whose identifiers have prefix s_i. Nodes within a corner are managed in a centralized fashion.

As each skeleton node of degree d can have up to $2d$ neighbors, we require that each corner has at least $2d$ nodes, called *core* ones. These nodes are connected in a ring and each of them is responsible for a (potential) connection to a different corner. For efficiency of routing, we demand that core nodes are sorted in the ring in the same way as in the original CCC network. It means that a node in corner s responsible for a connection to $s \oplus 2^{k+1}$ follows on the ring a node responsible for a connection to $s \oplus 2^k$, whereas a node responsible for a connection to $s \oplus 2^0$ follows the one responsible for a connection to $s \oplus 2^{d-1}$.

It might happen that a corner contains not only its core nodes, but also has to manage additional ones, which are called *spare*. When they join the corner, we put them on the ring between core nodes and we balance the path lengths between two consecutive core nodes. It means that if there are m spare nodes in the corner of dimension d, then there are between $\lfloor m/2d \rfloor$ and $\lceil m/2d \rceil$ spare nodes between any two consecutive core nodes. Due to this construction, each node in the network has degree at most 3. An example of such network is given in Fig. 3.

3.2 Network Maintenance

In this section, we show how to maintain the shape of the network in a distributed way in a dynamic setting, where nodes may join or leave the system. We start with showing a name-driven routing scheme, in which it is sufficient to know the destination identifier to reach the corresponding node in a greedy manner in a logarithmic number of steps (in the total number of nodes in the network).

We introduce a constant $\ell \geq 2$, which is a parameter of our algorithms. The runtime of our operations increase linearly with ℓ, and the probability that the network is not balanced (the dimensions of corners vary) decreases exponentially with ℓ.

Routing and Searching. When we search for a node with name s, we want to find a corner with a name being prefix of s. Routing is performed using the bit-fixing algorithm in which we fix bits of s one by one starting from the lowest bit. When we want to fix the k-th bit of s, and s differs from the current corner name in this bit, we go to a node in the current corner which has a connection to a corner differing exactly on this bit and traverse this connection. When we reach a destination corner (whose name is a prefix of s), we forward the message through all nodes in the corner. The message either reaches its destination or reaches some node for the second time. If the latter happens, this node answers the request with a negative (not found) answer.

Joining. We assume that when a node joins a network, it knows (or can discover and contact) another node which is already a member of the network. After choosing an identifier, the joining node asks its contact to find this identifier in the network. As the identifiers are supposed to be different, the result of the search will be negative, but a node with the longest prefix matching the identifier will be returned as a new contact. The new node joins (as a spare node) the corner to which its new contact belongs. At this point, it is checked if a split of the corner is necessary.

Splitting. A corner s of dimension d could be allowed to split when there are enough resources to form two corners of dimension $d+1$, i.e. if there are at least $2(d+1)$ nodes with prefix $s \cdot 0$ and with prefix $s \cdot 1$. However, for efficiency in a dynamic system, we split a corner when the number of nodes with both prefixes exceeds $12 \cdot \ell \cdot (d+1)$, where $\ell \geq 2$ is a constant parameter described above. Additionally, such a corner is allowed to split only if it has no neighbors of dimension $d-1$.

After splitting, we create two corners: $s \cdot 0$ and $s \cdot 1$ and assign nodes to them according to their names. A connection is established between the two new corners: a core node is responsible for this connection in each of them. Each connection from s is assigned to a proper one of the two new corners. If the connection has been to a corner of dimension d, both corners connect to the neighboring corner and the latter has to assign an additional core node to serve this connection. If the connection has been to two corners of dimension $d+1$, then $s \cdot 0$ connects to the corner with a 0 on position d and $s \cdot 1$ to the corner with a 1 on position d.

Leaving. When a node i wants to leave the network, it searches for a special spare node j in its corner, tells j to take its place, and leaves. The special spare node j is chosen so that after removing i and migrating j into its place, spare nodes are distributed evenly in the corner. After i leaves and j migrates, a check is performed if the number of nodes in the corner has decreased sufficiently to call a merging operation.

Merging. When the number of nodes in a corner s of dimension d drops below $2d$, we have to merge s with its neighbor $s' = s \oplus 2^{d-1}$, i.e. with the one differing from s on the last bit. Actually, we do it already when the number of nodes in such a corner drops to $(6\ell + 7) \cdot d$. First, we send a message to all neighboring corners of dimension $d+1$ (possibly including the neighbors across bit d), telling them to merge first. They merge recursively and after all neighbors of s are of dimension d or $d-1$, we merge the two corners. Naturally, whenever a corner receives a message from one of its neighbors (of lower dimension) telling it to merge, it starts the merge procedure too.

3.3 Analysis

Before we bound the runtime of all operations on the system, we prove that with high probability the system is balanced, i.e. the dimensions of each corner are roughly the same.

To formally define this notion, we introduce (just for the analysis) a parameter d_u, which would be the current dimension of the network if it would be a CCC. This means that if n is the current number of nodes in the network, then $d_u \cdot 2^{d_u} \le n < (d_u + 1) \cdot 2^{d_u+1}$. For $d_u > 2$, it holds that $d_u/2 \le \ln n \le 2d_u$. Additionally, we introduce a parameter d_l differing from d_u by a constant: $d_l := d_u - \log \ell - 5$. For simplicity of notation, we assume that all nodes' identifiers are infinite.

Definition 4. *A skew CCC network is* balanced *if all corners' dimension are between d_l and d_u.*

Now we show that the network is balanced with high probability. We note that even in case of bad luck (happening with polynomially small probability), the system still works — it might just work slower.

Lemma 1. *If the network is stable, i.e. no nodes are currently joining or leaving and no split or merge operations are currently being executed or pending, then with probability $1 - 2 \cdot n^{-\ell}$ the network is balanced.*

Proof. We prove two claims:

(i) the probability that there exists a corner with dimension d_u or greater is at most $n^{-\ell}$;

(ii) the probability that there exists a corner with dimension d_l or smaller is at most $n^{-\ell}$.

For proving (i), we take a closer look at the set $\mathcal{S}$ of all node identifiers. For any string s, let $\mathcal{S}_s = \{s_i \in \mathcal{S} : s \sqsubseteq s_i\}$, i.e. S_s consists of all identifiers starting with s. We say that $\mathcal{S}$ is *well separated* if for each d_u-bit string s, $|\mathcal{S}_s| \le (6\ell + 7) \cdot d_u$.

First, we observe that if $\mathcal{S}$ is well separated, then there is no corner of dimension d_u or higher. Assume the contrary and choose the corner s_i with highest dimension (at least d_u). Then there are at most $(6\ell + 7) \cdot d_u$ identifiers starting with s_i, i.e. remaining in this corner. As all the neighbors of corner s_i have smaller or equal dimension, this corner should be merged, which contradicts the assumption that the network is stable.

Second, we show that the probability that $\mathcal{S}$ is not well separated is at most $n^{-\ell}$. Fix any string s of length d_u. For each node i with identifier s_i let an indicator random variable X_i^s be equal to 1 if $s \sqsubseteq s_i$. Since s is a d_u-bit string, we have $\mathrm{E}[X_i^s] = 2^{-d_u}$. Let $X^s = \sum_{i=1}^n X_i^s$; by the linearity of expectation, $\mathrm{E}[X^s] = \sum_{i=1}^n \mathrm{E}[X_i^s] = n \cdot 2^{-d_u} \geq d_u$. Using the Chernoff bound, we obtain that

$$\begin{aligned}
\Pr\left[X^s \geq (6\ell+7)d_u\right] &\leq \Pr\left[X^s - \mathrm{E}[X^s] \geq (6\ell+6)\cdot \mathrm{E}[X^s]\right] \\
&\leq \exp\left(-\frac{(6\ell+6)\cdot \mathrm{E}[X^s]}{3}\right) \\
&\leq \mathrm{e}^{-(2\ell+2)\cdot d_u} \\
&\leq n^{-(\ell+1)} \quad .
\end{aligned}$$

There are $2^{d_u} \leq n$ possible d_u-bit strings, and thus (by the sum argument) the probability that $\mathcal{S}$ is not well separated is at most $n \cdot n^{-(\ell+1)} = n^{-\ell}$.

Proving (ii) is analogous. We say that $\mathcal{S}$ is *well glued* if for each d_l-bit string s, $|\mathcal{S}_s| \geq 12\ell \cdot d_l$.

Again, we prove that if $\mathcal{S}$ is well glued, then there is no corner of dimension d_l or lower. Assume the contrary and let s_i be the corner with lowest dimension. There are at least $12\ell \cdot d_l$ nodes in corner s_i. As all the neighbors of corner s_i have greater or equal dimension, this corner should be splitted, which contradicts the assumption that the network is stable.

Again, we show that the probability that $\mathcal{S}$ is not well glued is at most $n^{-\ell}$. Let t be any d_l-bit string, indicator random variables X_i^t denote if $t \sqsubseteq s_i$ and $X^t = \sum_{i=1}^n X_i^t$. Then $\mathrm{E}[X^t] = n \cdot 2^{-(d_u - 5 - \log \ell)} \geq 32\ell \cdot d_u$. Using the Chernoff bound, we get that

$$\begin{aligned}
\Pr\left[X^t \leq 12\ell \cdot d_l\right] &\leq \Pr\left[X^t \leq (1-1/2)\cdot \mathrm{E}[X^t]\right] \\
&\leq \mathrm{e}^{-\mathrm{E}[X^t]/8} \\
&\leq \mathrm{e}^{-4\ell \cdot d_u} \\
&\leq n^{-2\ell} \quad .
\end{aligned}$$

There are $2^{d_l} \leq n$ possible d_l-bit strings, and thus the probability that $\mathcal{S}$ is not well glued is at most $n \cdot n^{-2\ell} \leq n^{-\ell}$. □

According to Lemma 1, the system is balanced with high probability. Now, we will show that all basic operations are performed in a time that is logarithmic in the current number of nodes. In the following, we assume that the high probability event of the network being balanced actually happens.

Lemma 2. *If the network is balanced, then each search operation is performed in logarithmic time.*

Proof. Since each corner is of dimension $\Theta(\log n)$, we have to fix $\Theta(\log n)$ bits in order to reach the destination corner. As the number of nodes in each corner is within a constant factor from its dimension, there are $\Theta(\log n)$ nodes in each corner. In particular, there is a constant number of spare nodes between any two consecutive core nodes. Thus, in order to fix the i-th bit after fixing the $(i-1)$-st bit we have to traverse only a constant number of edges. In order to fix the first bit and to reach the destination node after reaching the destination corner, we have to travel at most through all the nodes of two corners. Hence, the total number of traversed edges is $O(\log n)$. □

Before we prove upper bounds for join and leave operations we bound the time of split and merge.

Lemma 3. *If the network is balanced, then each split operation is performed in logarithmic time.*

Proof. When we split a corner s of dimension d into corners $s \cdot 0$ and $s \cdot 1$ of dimension $d + 1$, then there are more than sufficient nodes for each corner and we know that each neighboring corner is of dimension d or $d + 1$.

Since the corner s currently has to split and $\ell \geq 1$, there are at least $12(d+1)$ nodes of each type in s. Starting in any node, we traverse the ring a constant number of times and do the following. In the first pass, we make sure that there are two connections to neighbor corners across every bit. If there are two connections already, there is nothing to do and if there is only one, we take any spare node in the corner we are currently splitting and make a connection to a spare node in the neighboring corner. From now on, these two spare nodes (one in each corner) are core nodes. Finally, we add two core nodes without an outside connection to s; they will be responsible for connecting the two corners into which we split s.

In the second pass, we use all spare nodes to create two additional rings: one built of nodes with the d-th bit equal to 0 and the other built of nodes with the d-th bit equal to 1. Since each ring has at most $2(d + 1)$ core nodes and at least $12(d + 1)$ nodes in total, each of the newly created rings has at least $10(d + 1)$ nodes. In the next pass, we can go along each of the three rings in parallel and pass the responsibility for a connection to another corner from the old ring to one of the new ones. This means that the ring with nodes with d-th bit equal to 0 takes responsibility for the connections to corners $(s \oplus 2^k) \cdot 0$ (analogously if the last bit is equal to 1).

In the last traversal of all rings, we delete nodes from the old ring and make them join one of the new rings as spare nodes. Again, we move nodes with the d-th bit equal to 0 to the newly created corner $s \cdot 0$ and nodes with the d-th bit equal to 1 to the newly created corner $s \cdot 1$.

As we have used only a constant number of traversals of rings of length $O(\log n)$, the whole split operation needs time $O(\log n)$. □

We note that in case of search and split operations, the system can be balanced in a weaker sense than we defined, i.e. we just need that the corner dimension is $O(\log n)$. The merge operation is the only operation which depends on the property that the difference between dimension of the corners is constant. It is also the most time and work consuming function, as a single merge operation might need executing other merge operations before it can start.

Lemma 4. *If the network is balanced, then a merge operation is performed in time $O(\text{polylog}(n))$, including recursive execution of other needed merge operations, whereas the amortized cost per merge operation is $O(\log n)$.*

Proof. Since the total number of different dimensions of all corners in the network is bounded by a constant ($\log \ell + 5$), the recursive execution of other possibly necessary merge operations for neighboring corners of higher dimensions has only constant depth. As on each level a corner can have $d + O(1) = O(\log n)$ neighbors, the total number of involved corners is $\log^{O(1)} n$. Below we prove that a single merge operation of corners $s \cdot 0$ and $s \cdot 1$ of dimension $d+1$ into a corner s of dimension d has cost $O(d) = O(\log n)$, if all neighbors have been reduced to equal or lower dimension.

Similarly to the split operation, we can traverse in parallel both rings $s \cdot 0$ and $s \cdot 1$ which we want to merge. We denote their dimension by d. As no neighbor of $s \cdot 0$ and $s \cdot 1$ is of dimension $d + 1$, only d connections of $2d$ core nodes are actually used in each of them. We first build a ring composed of these used core nodes of both old rings, whereas we zip them into one ring interleaving the core nodes of $s \cdot 0$ and the core nodes of $s \cdot 1$. When we remove core nodes from the old rings, we glue the holes so that we get two rings composed of spare nodes. Next, we remove two core nodes which have been responsible for the connection across bit $d - 1$ (they connected $s \cdot 0$ to $s \cdot 1$) and move them to one of the old rings as spare nodes. In the next traversal of the old rings we calculate how many spare nodes they contain and then, in the last traversal, we evenly distribute the spare nodes in the newly created corner s.

Notice that there is no need to add any connections to neighboring corners — all necessary connections already exist. On the other hand, if our new $(d - 1)$-dimensional corner is a neighbor to another $(d - 1)$-dimensional one, we have a double connection with this corner. We should remove one of the connections, namely the one which originates from $s \cdot 1$.

Since the cost of merging $s \cdot 0$ and $s \cdot 1$ into s (not including recursive merging of neighbors) is $\Theta(d)$, and the cost of splitting s into $s \cdot 0$ and $s \cdot 1$ has also been $\Theta(d)$, we can amortize the cost of merging into s against the cost of splitting s. This shows that the amortized cost of a merge operation together with its symmetric split operation of a corner of dimension d is $\Theta(d) = \Theta(\log n_m) = \Theta(\log n_s)$, where n_m is the number of nodes in the system at the moment when we perform the merge operation and n_s is the number of nodes at the moment when we have performed the split operation. □

Lemma 5. *The join operation can be performed in logarithmic time.*

Proof. Each time a node joins the network, it has to search its position inside the network and to take its position inside the ring. Based on the previous lemmas, the search operation can be performed in logarithmic time and the update of the ring structure involves the creation of two new edges and the removal of one existing edge. Furthermore, it might happen that the corner has to split its dimension, resulting in additional $O(\log n)$ operations. □

Finally, in the following lemma, we analyze the cost of a leave operation.

Lemma 6. *The leave operation can be performed in polylogarithmic and amortized logarithmic time.*

Proof. A typical leave operation only triggers a few connection updates. If the node has been a spare node in its corner, the leave operation involves two connection updates, if the node has been a core node, one additional update has to be performed to re-connect to the neighboring corner.

Besides the connection updates, a node leaving the network might also trigger a merge operation of the corner. Based on the previous lemmas, each merge operation costs at most polylogarithmic time (and logarithmic amortized time) and majorizes the cost of a leave operation. □

4 Conclusion and Outlook

We have shown a fully distributed but simple scheme which joins a potentially very large set of computationally weak nodes into an organized network with minimal possible degree of 3, logarithmic dilation and name-driven routing.

Based on the properties and the structure of the SkewCCC network, it is possible to further investigate aspects of heterogeneity and locality. The former means allowing the existence of network nodes which can have a higher degree and potentially also greater computational power. The latter aspect would incorporate distances of the underlying network.

References

1. Aspnes, J., Shah, G.: Skip graphs. ACM Transactions on Algorithms 3(4) (2007); Also appeared in: Proc. of the 14th SODA, pp. 384–393 (2003)
2. Awerbuch, B., Scheideler, C.: The hyperring: a low-congestion deterministic data structure for distributed environments. In: Proc. of the 15th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 318–327 (2004)
3. Barrière, L., Fraigniaud, P., Narayanan, L., Opatrny, J.: Dynamic construction of bluetooth scatternets of fixed degree and low diameter. In: Proc. of the 14th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 781–790 (2003)
4. Bienkowski, M., Brinkmann, A., Korzeniowski, M., Orhan, O.: Cube connected cycles based bluetooth scatternet formation. In: Proc. of the 4th International Conference on Networking, pp. 413–420 (2005)
5. Harvey, N.J.A., Jones, M.B., Saroiu, S., Theimer, M., Wolman, A.: Skipnet: a scalable overlay network with practical locality properties. In: Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (2003)

6. Jiang, X., Polastre, J., Culler, D.: Perpetual environmentally powered sensor networks. In: Proc. of the 4th Int. Symp. on Information Processing in Sensor Networks (IPSN), pp. 463–468 (2005)
7. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proc. of the 29th ACM Symp. on Theory of Computing (STOC), pp. 654–663 (1997)
8. Leighton, F.T.: Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers, San Francisco (1992)
9. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proc. of the 21st ACM Symp. on Principles of Distributed Computing (PODC), pp. 183–192 (2002)
10. Naor, M., Wieder, U.: Novel architectures for P2P applications: The continuous-discrete approach. ACM Transactions on Algorithms 3(3) (2007); Also appeared in: Proc. of the 15th SPAA, pp 50–59 (2003)
11. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
12. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking 11(1), 17–32 (2003); In: Proc. of the ACM SIGCOMM, pp. 149–160 (2001)
13. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications 22(1), 41–53 (2004)

On the Time-Complexity of Robust and Amnesic Storage[⋆]

Dan Dobre, Matthias Majuntke, and Neeraj Suri

TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
{dan,majuntke,suri}@cs.tu-darmstadt.de

Abstract. We consider wait-free implementations of a regular read/write register for unauthenticated data using a collection of $3t + k$ base objects, t of which can be subject to Byzantine failures. We focus on *amnesic* algorithms that store only a limited number of values in the base objects. In contrast, non-amnesic algorithms store an unbounded number of values, which can eventually lead to problems of space exhaustion. Lower bounds on the time-complexity of read and write operations are currently met only by non-amnesic algorithms. In this paper, we show for the first time that amnesic algorithms can also meet these lower bounds. We do this by giving two amnesic constructions: for $k = 1$, we show that the lower bound of *two* communication rounds is also sufficient for *every* read operation to complete and for $k = t + 1$ we show that the lower bound of *one* round is also sufficient for *every* operation to complete.

Keywords: distributed storage, Byzantine failures, wait-free algorithms.

1 Introduction

Motivated by recent advances in the Storage-Area Network (SAN) technology, and also by the availability of cheap commodity disks, distributed storage has become a popular method to provide increased storage space, high availability and disaster tolerance. We address the problem of implementing a reliable read/write distributed storage service from unreliable storage units (e.g. disks), a threshold of which might fail in a malicious manner. Fault-tolerant access to replicated remote data can easily become a performance bottleneck, especially for data-centric applications usually requiring frequent data access. Therefore, minimizing the time-complexity of read and write operations is essential. In this paper, we show how optimal time-complexity can be achieved using algorithms that are also space-efficient.

An essential building block of a distributed storage system is the abstraction of a read/write register, which provides two primitives: a *write* operation, which writes a value into the register, and a *read* operation which returns a value previously written [1]. Much recent work, and this paper as well, focuses on *regular*

[⋆] Research funded in part by DFG GRK 1362 (TUD GKmM), EC NoE ReSIST and Microsoft Research via the European PhD Fellowship.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 197–216, 2008.

registers where read operations never return outdated values. A regular register is deemed to return the last value written before the read was invoked, or one written concurrently with the read (see [1] for a formal definition). Regular registers are attractive because even under concurrency, they never return spurious values as sometimes done by the weaker class of *safe* registers [1]. Furthermore, they can be used, for instance, together with a failure detector to implement consensus [2].

The abstraction of a reliable storage is typically built by replicating the data over multiple unreliable distributed storage units called *base objects*. These can range from simple (low-level) read/write registers to more powerful base objects like *active disks* [3] that can perform some more sophisticated operations (e.g. an atomic read-modify-write). Taken to the extreme, base objects can also be implemented by full-fledged servers that execute more complex protocols and actively push data [4]. We consider Byzantine-fault tolerant register constructions where a threshold $t < n/3$ of the base objects can fail by being *non-responsive* or by returning *arbitrary* values, a failure model called NR-arbitrary [5]. Furthermore, we consider *wait-free* implementations where concurrent access to the base objects and client failures must not hamper the liveness of the algorithm. Wait-freedom is the strongest possible liveness property, stating that each client completes its operations independent of the progress and activity of other clients [6]. Algorithms that wait-free implement a regular register from Byzantine components are called *robust* [7]. An implementation of a reliable register requires the (client) processes accessing the register via a high-level operation to invoke multiple low-level operations on the base objects. In a distributed setting, each invocation of a low-level operation results in one *round* of communication from the client to the base object and back. The number of rounds needed to complete the high-level operation is used as a measure for the time-complexity of the algorithm.

Robust algorithms are particularly difficult to design when the base objects store only a limited number of written values. Algorithms that satisfy this property are called *amnesic*. With amnesic algorithms, values previously stored are not permanently kept in storage but are eventually erased by a sequence of values written after them. Amnesic algorithms eliminate the problem of space exhaustion raised by (existing) non-amnesic algorithms, which take the approach of storing the entire version history. Therefore, the amnesic property captures an important aspect of the space requirements of a distributed storage implementation. The notion of amnesic storage was introduced in [7] and defined in terms of *write-reachable configurations*. A configuration captures the state of the correct base objects. Starting from an initial configuration, any low-level read/write operation (i.e., one changing the state of a base object) leads the system to a new configuration. A configuration C' is write-reachable from a configuration C when there is a sequence consisting only of (high-level) write operations that starting from C, leads the system to C'. Intuitively, a storage algorithm is amnesic if, except a finite number of configurations, all configurations reached by the algorithm are eventually *erased* by a sufficient number of values written after them. Erasing a configuration C', which itself was obtained from a configuration C, means to reach a configuration C'' that could have been obtained directly from C without going through C'. This means that once in C'', the system cannot tell whether it has ever been in configuration C'. For instace, an algorithm that stores the entire history of written values in the base objects is not amnesic. In contrast, an algorithm that stores in the base objects only the last l written values is amnesic because after writing the $l + 1^{\text{st}}$ value, the algorithm cannot recall the first written value anymore.

1.1 Previous and Related Work

Despite the importance of *amnesic and robust* distributed storage, most implementations to date are either *not* robust or *not* amnesic. While some relax wait-freedom and provide weaker termination guarantees instead [2,8], others relax consistency and implement only the weaker safe semantics [5,9,2,10]. Generally, when it comes to robustly accessing (unauthenticated) data, most algorithms store an unlimited number of values in the base objects [11,10,12]. Also in systems where base objects push messages to subscribed clients [4,13,14], the servers store every update until the corresponding message has been received by every non-faulty subscriber. Therefore, when the system is asynchronous, the servers might store an unbounded number of updates. A different approach is to assume a stronger model where data is self-verifying [9,15,16], typically based on digital signatures. For unauthenticated data, the only existing robust and amnesic storage algorithms [17,18] do not achieve the same time-complexity as non-amnesic ones. Time-complexity lower bounds have shown that protocols using the optimal number of $3t + 1$ base objects [4] require at least *two rounds* to implement both read/write operations [10,2]. So far these bounds are met only by non-amnesic algorithms [12]. In fact, the only robust and amnesic algorithm with optimal resilience [17] requires an unbounded number of read rounds in the worst case. For the $4t + 1$ case, the trivial lower bound of *one round* for both operations is *not* reached by the only other existing amnesic implementation [18] that albeit elegant, requires at least *three* rounds for reading and *two* for writing.

1.2 Paper Contributions

Current state of the art protocols leave the following question open: *Do amnesic algorithms inherently have a non-optimal time complexity?* This paper addresses this question and shows, for the first time, that amnesic algorithms can achieve optimal time complexity in both the $3t + 1$ and $4t + 1$ cases. Justified by the impossibility of amnesic and robust register constructions when readers do not write [7], one of the key principles shared by our algorithms is having the readers change the base objects' state. The developed algorithms are based on a novel concurrency detection mechanism and a helping procedure, by which a writer detects overlapping reads and helps them to complete. Specifically, the paper makes the following two main contributions:

- A first algorithm, termed DMS, which uses $4t + 1$ base objects, described in Section 3. With DMS, *every* (high-level) read and write operation is *fast*, i.e., it completes after only *one round* of communication with the base objects. This is the first robust and amnesic register construction (for unauthenticated data) with optimal time-complexity.

- A second algorithm, termed DMS3, which uses the optimal number of $3t+1$ base objects, presented in Section 4. With DMS3, every (high-level) read operation completes after only *two rounds*, while *write* operations complete after *three rounds*. This is the first amnesic and robust register construction (for unauthenticated data) with optimal read complexity. Note also that, compared to the optimal write complexity, it needs only one additional communication round.

Table 1 summarizes our contributions and compares DMS and DMS3 with recent distributed storage solutions for unauthenticated data.

Table 1. Distributed storage for unauthenticated data

		Worst-Case Time-complexity			
Protocol	Resilience	Read	Write	Amnesic	Robust
Abraham et al. [18]	$4t+1$	3	2	√	√
DMS	$4t+1$	1	1	√	√
Guerraoui and Vukolić [10]	$3t+1$	2	2	×	√
Byzantine Disk Paxos [2]	$3t+1$	$t+1$	2	√	×
Guerraoui et al. [17]	$3t+1$	unbounded	3	√	√
DMS3	$3t+1$	2	3	√	√

2 System Model and Preliminaries

2.1 System Model

We consider an asynchronous shared memory system consisting of a collection of processes interacting with a finite collection of n base objects. Up to t out of n base objects can suffer NR-arbitrary failures [5] and any number of processes may fail by crashing. Each object implements one or more *registers*. A register is an object type with value domains *Val*, an initial value v_0 and two invocations: *read*, whose response is $v \in Vals$ and *write*(v), $v \in Vals$, whose response is *ack*. A read/write register is single-reader single-writer (SRSW) if only one process can read it and only one can write to it; a register is multi-reader single-writer (MRSW) if multiple processes can read it. Sometimes processes need to perform two operations on the same base object, a write (of a register) followed by a read (of a different register). To reduce the number of rounds, we collapse consecutive write/read operations accessing the same base object to a single low-level operation called *write&read*. The *write&read* operation can be implemented in a single round, for instance using active disks [3] as base objects[1].

[1] Note that since *write&read* is not an atomic operation, it can be implemented from simple read/write registers and thus the model is not strengthened.

2.2 Preliminaries

In order to distinguish between the target register's interface and that of the base registers, throughout the paper we denote the high-level read (resp. write) operation as READ (resp. WRITE). Each of the developed protocols uses an underlying layer that invokes operations on different base objects in separate threads in parallel. We use the notation from [2] and write **invoke** $write(X_i,v)$ (resp. **invoke** $x[i] \leftarrow read(X_i)$) to denote that a write(v) operation on register X_i (resp. a read of register X_i whose response will be stored in a local variable $x[i]$) is invoked in a separate thread by the underlying layer. The notation **invoke** $x[i] \leftarrow write\&read(\langle Y_i, v\rangle, X_i)$ denotes the invocation of an operation *write&read* on base object i, consisting of a write(v) on register Y_i followed by a read of register X_i (whose response will be stored in $x[i]$).

As base objects may be non-responsive, high-level operations can return while there are still pending invocations to the base objects. The underlying layer keeps track of which invocations are pending to ensure well-formedness, i.e., that a process does not invoke an operation on a base object while invocations of the same process and on the same base object are pending. Instead, the operation is denoted enabled. If an operation is enabled when a pending one responds, the response is discarded and the enabled operation is invoked. See e.g. [2] for a detailed implementation of such layers.

We say that an operation *op* is *complete* in a run if the run contains a response step for *op*. For any two operations op_1 and op_2, when the response step of op_1 precedes the invocation step of op_2, we say op_1 *precedes* op_2. If neither $op1$ nor $op2$ precedes the other then the two operations are said to be *concurrent*.

In order to better convey the insight behind the protocols, we simplify the presentation in two ways. We introduce a shared object termed *safe counter* and describe both algorithms in terms of this abstraction. Although easy to follow, the resulting implementations require more rounds than the optimal number. Thus, for each of the protocols we explain how with small changes these rather didactic versions can be "condensed" to achieve the announced time-complexity. The full details of the optimizations can be found in our publicly available technical report [19]. Secondly, for presentation simplicity we implement a SRSW register. Conceptually, a MRSW register for m readers can be constructed using m copies of this register, one for each reader. In a distributed storage setting, the writer accesses all m copies in parallel, whereas the reader accesses a single copy. It is worth noting that this approach is heavy and that in practice, cheaper solutions are needed to reduce the communication complexity and the amount of memory needed in the base objects.

We now introduce the *safe counter* abstraction used in our algorithms. A safe counter has two *wait-free operations* INC and GET. INC modifies the counter by incrementing its value (initially 0) and returns the new value. Specifically, the k^{th} INC operation denoted INC^k returns k. GET returns the current value of the counter without modifying it. The counter provides the following guarantees:

Validity: If GET returns k then GET does not precede INC^k.
Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Note that under concurrency, a safe counter might return an outdated value, but never a forged value. In the absence of concurrency, the newest value is returned.

We now explain the intuition behind our algorithms. Both algorithms use the safe counter introduced above to arbitrate between writer and reader. During each READ (resp. WRITE) operation, the reader (resp. writer) executes INC to advance the counter (resp. GET to read the counter). The values returned by the counter's operations are termed *views*. By incrementing its current view, a READ announces its intent to read from the base objects. A subsequent invocation of GET by the writer returns the updated view. When the writer detects a concurrent READ, indicated by a view change, it *freezes* the most recent value previously written. Freezing a value v means that v may be overwritten *only if* the READ operation that attempts to read v has completed. We note that the READ operation that caused a value v to be frozen does not violate regularity by returning v because all newer values were written concurrently with the READ. However, READs must not return old values previously frozen. This is necessary to ensure regularity and it is done by freezing a value v together with the view of the READ due to which v is frozen. A READ whose view is higher than the one associated with v knows that it must pick a newer value. A READ operation completes when it finds a value v to return such that (a) v is reported by a correct base object and (b) v is not older than the latest value written before the READ is invoked.

3 A Fast Robust and Amnesic Algorithm

We start by describing an initial version of protocol DMS that uses the safe counter abstraction. It is worth noting that the algorithm requires more rounds than the optimum, but it conveys the main idea. Next, we explain the changes applied to DMS to obtain an algorithm with optimal time-complexity.

3.1 Protocol Description

We present a robust and amnesic SRSW register construction using a safe counter and $4t + 1$ regular base registers, out of which t can incur NR-arbitrary failures. Figure 1 illustrates a simple construction of the safe counter used. The description of the counter is omitted for the sake of brevity. The shared objects used by DMS are detailed in Figure 2 and the algorithm appears in Figure 3.

The WRITE performs in two phases, (1) a write phase where it first writes a timestamp-value pair to $n - t$ registers and (2) a subsequent read phase, where it executes GET to read the current view. In case a view change occurs between two successive WRITEs, the value of the first WRITE is frozen. Recall that once frozen, a value is not erased before the next view change. Similarly, the READ

```
Predicates:
    safe($c$) $\triangleq$
    $|\{i : c' \in y[i] \wedge c' \geq c\}| \geq t+1$

GET()
    for $1 \leq i \leq n$ do $y[i] \leftarrow \bot$
    for $1 \leq i \leq n$ do
        invoke $y[i] \leftarrow read(Y_i)$
    wait for $n-t$ responses
    return $\max\{c \in Integers : \mathsf{safe}(c)\}$
```

```
Local variables:
    $y[1 \ldots n] \in Integers$
    $k \in Integers$, initially 0

INC()
    $k \leftarrow k+1$
    for $1 \leq i \leq n$ do
        invoke $write(Y_i, k)$
    wait for $n-t$ responses
    return $k$
```

Fig. 1. Safe counter from $4t+1$ safe registers $Y_i \in$ *Integers*

consists of (1) a write phase, where it first executes INC to increment the current view and (2) a subsequent read phase, where it reads at least $n-t$ registers. To ensure that READ never returns a corrupted value, the returned value must be read from $t+1$ registers, a condition captured by the predicate safe. Moreover, to ensure regularity, READ must not return old values written before the last WRITE preceding the READ. This condition is captured by the predicate highestCand.

We now give a more detailed description of the algorithm. As depicted in Figure 2, each base register consists of three value fields *current*, *prev* and *frozen* holding timestamp-value pairs, and an integer field *view*. The writer holds a variable x of the same type and uses x to overwrite the base registers. Each WRITE operation saves the timestamp-value pair previously written in *x.prev*. Then, it chooses an increasing timestamp, stores the value together with the timestamp in *x.curr* and overwrites $n-t$ registers with x. Subsequently, the writer executes GET. If the view returned by GET is higher than the current view (indicating a concurrent READ), then $x.view$ is updated and the most recent value previously written is frozen, i.e., the content of *x.prev* is stored in *x.frozen* (line 14, Figure 3). Finally, WRITE returns *ack* and completes. It is important to note that the algorithm is amnesic because each correct base object stores at most three values (*curr*, *prev* and *frozen*).

The READ first executes INC to increment the current view, and then it reads at least $n-t$ registers into the array $x[1...n]$, where element i stores the content of register X_i. If necessary, it waits for additional responses until there is a *candidate* for returning, i.e., a read timestamp-value pair that satisfies both predicates safe

Types:

TSVals $\triangleq$ *Integers* $\times$ *Vals*, with selectors *ts* and *val*

Shared objects:

- regular registers $X_i \in TSVals^3 \times Integers$ with selectors *curr*, *prev*, *frozen* and *view*, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$
- safe counter object $Y \in Integers$, initially $Y = 0$

Fig. 2. Shared objects used by DMS

```
Predicates (reader):
  $\mathsf{readFrom}(c, i) \triangleq (c = x[i].curr \wedge x[i].view < view) \vee$
                      $(c = x[i].frozen \wedge x[i].view = view)$
  $\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].curr, x[i].prev, x[i].frozen\}\}| \geq t+1$
  $\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leq c.ts\}| \geq 2t+1$

Local variables (reader):
  $view \in Integers$, initially 0
  $x[1 \dots n] \in TSVals^3 \times Integers$

READ()
  for $1 \leq i \leq n$ do $x[i] \leftarrow \bot$
  $view \leftarrow \textsc{inc}(Y)$
  for $1 \leq i \leq n$ do invoke $x[i] \leftarrow read(X_i)$
  wait until $n - t$ responded $\wedge \exists c \in TSVals$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$
  return $c.val$

Local variables (writer):
  $newView$, $ts \in Integers$, initially 0
  $x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE($v$)
  $ts \leftarrow ts+1$
  $x.prev \leftarrow x.curr$
  $x.curr \leftarrow \langle ts, v\rangle$
  for $1 \leq i \leq n$ do invoke $write(X_i, x)$
  wait for $n - t$ responses
  $newView \leftarrow \textsc{get}(Y)$
  if $newView > x.view$ then
    $x.view \leftarrow newView$
    $x.frozen \leftarrow x.prev$
  return $ack$
```

Fig. 3. Robust and amnesic storage algorithm DMS $(4t+1)$

and highestCand. A timestamp-value pair c is safe when it appears in some field *curr*, *prev* or *frozen* of $t+1$ elements of x, ensuring that c was reported by at least one correct register. Enforcing regularity is more subtle. Simply waiting until the highest timestamped value read becomes safe might violate liveness because it may be reported by a faulty register. To solve this problem, we introduce the predicate highestCand. A value c is highestCand when $2t+1$ base registers report values that were written *not after* c, which implies that newer values are missing from $t+1$ correct registers. As any complete WRITE skips at most t correct registers, all values newer than c were written *not* before READ is invoked and consequently, they can be discarded from the set of possible return candidates.

We now explain with help of Figure 4 why READs are wait-free. We consider the critical situation when multiple WRITEs are concurrent with a READ. Specifically, we consider the k^{th} READ (henceforth READ^k), whose INC results in k

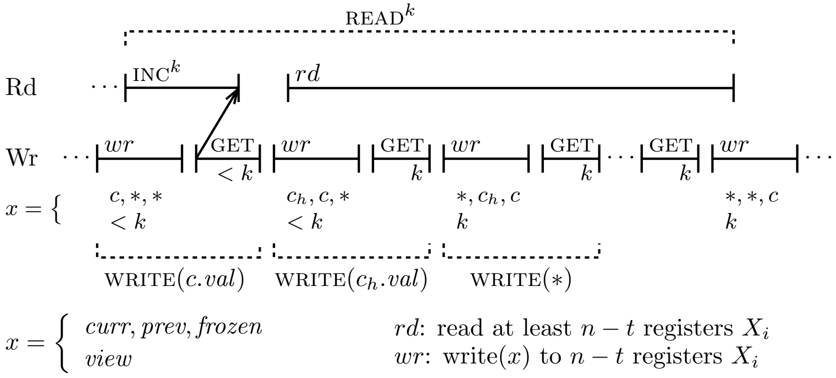


Fig. 4. Correctness argument of the READ operation in DMS

(henceforth INC^k), and the *last* WRITE that still reads a view lower than k, i.e., the corresponding GET returns a view lower than k. Note that by the safety property of the counter, INC^k does not precede GET and thus c is stored in $2t+1$ correct registers *before* any of them is read. A key aspect of the algorithm is to ensure that no matter how many WRITEs are subsequently invoked, c never disappears from all fields of those $2t + 1$ correct registers, as long as READ^k is still in progress. Essentially this holds because the subsequent WRITE re-writes c to all registers and it also freezes c to ensure that future WRITEs do the same. In this process, c migrates from *curr* to *prev* and from *prev* to *frozen* where it stays until the next view change. Therefore, c eventually becomes safe. But what if c is not highestCand? In this situation, at least $t + 1$ correct registers report timestamp-value pairs higher than c. We note that if any of them had stored c in its *frozen* field, then it would report c. This implies that none of these registers has stored c in its *frozen* field and thus, also none of these registers has stored a timestamp-value pair higher than c_h in its *curr* field. Therefore, c_h is reported by $t + 1$ correct registers, and hence it is safe. Note that c_h is also highestCand because only faulty registers report values with higher timestamps.

We now explain how the fast algorithm is derived from DMS. The principle underlying the optimization is to condense one round of write to the base objects and a subsequent round of read of the base objects into a single round of *write&read*. For this purpose we disregard the safe counter abstraction and directly weave INC and GET (Fig. 1) into READ and WRITE (Fig. 3) respectively. As a result, the reader advances the view *and* reads the base registers in one round. Likewise, the writer stores a value in the base registers *and* reads the view in a single round. The reader code (Fig. 3) is modified as follows: variable *view* is incremented locally, and line 3 is replaced with the statement **for** $1 \leq i \leq n$ **do invoke** $x[i] \leftarrow write\&read(\langle Y_i, view\rangle, X_i)$. Similarly, in the writer code (Fig. 3), line 9 is replaced with the statement **for** $1 \leq i \leq n$ **do invoke** $y[i] \leftarrow write\&read(\langle X_i, x\rangle, Y_i)$. Additionally in line 11, instead of executing GET, the writer picks the $t + 1^{\text{st}}$ highest element of y.

We now informally argue that the optimization is correctness preserving. As in the above example, we consider READ^k and the last WRITE that reads a view lower than k. Recall that the WRITE operation stores c in $2t + 1$ correct base objects and each of them responds with the current view it has stored. The writer then picks the $t + 1^{\text{st}}$ highest view reported. We argue that $t + 1$ correct base objects have stored c before any of them respond to READ^k. This would imply that c is safe. As the WRITE operation reads a view lower than k, out of the $2t + 1$ correct base objects accessed by it, at most t report k. Thus, the remaining $t + 1$ objects are accessed by READ^k only after c was written to them. Applying the above arguments, it is not difficult to see that c is never erased from $t + 1$ correct registers before READ^k completes, and thus it eventually becomes safe. Regarding regularity, again, arguments similar to above can be used. A formal proof of the optimized algorithm can be found in the full paper [19]. The remainder of this section is concerned with the correctness of DMS.

3.2 Protocol Correctness

Lemma 1 (Regularity). *Algorithm DMS in Figure 3 implements a regular register.*

Proof. We show that the READ operation always returns the value of the latest WRITE preceding the READ, or a newer written value. Suppose that $c.val$ is the value returned by READ^k. We assume by contradiction that there exists a value $c_h.val$ such that $c_h.ts > c.ts$ and WRITE($c_h.val$) precedes READ^k. As WRITE($c_h.val$) is complete, $n - 2t$ correct registers have stored c_h or a higher timestamp-value pair before any of them is read. The fact that $c.val$ is returned implies that c is highestCand. Thus, there are at least $2t + 1$ registers X_i and values c' with timestamp $c'.ts \leq c.ts$ such that readFrom(c',i) is true. Note that one of them is a correct register X_i updated with c_h. As values are written with monotonically increasing timestamps, by definition of readFrom, necessarily c' is read from $x[i].frozen$ and $x[i].view = k$. However, because the counter is valid, the first time a WRITE operation reads view k is only after the WRITE of $c_h.val$. Thus, in view k only timestamp-value pairs c_h or higher are frozen, a contradiction. □

Lemma 2 (Wait-freedom). *Algorithm DMS in Figure 3 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n - t$ responses. Showing that READs are also live is more subtle. To derive a contradiction, we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $3t + 1$) have responded. We choose c as the $2t + 1^{\text{st}}$ lowest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leq c.ts$ are readFrom $2t + 1$ correct registers (set L). Also, we note that values with timestamps $\geq c.ts$ are readFrom $t + 1$ correct registers (set R). In the following, we distinguish the cases where the WRITE of *c.val* reads a view equal to k (case 1), or lower than k (case 2). Note that by the validity of the counter, only views $\leq k$ are returned. Case 1 implies that (a) only timestamp-value pairs lower than c are frozen, and (b) c is the highest timestamp-value pair readFrom the *curr* field of a correct register. Together (a) and (b) imply that c is the highest timestamp-value pair readFrom a correct register. Thus, for all registers $X_i \in R$ ($\geq t+1$), readFrom(c',i) implies that $c' = c$ and hence, c is safe. We now consider case 2 where WRITE($c.val$) reads a view lower than k. This implies that c or a higher timestamp-value pair is frozen in view k. If $t+1$ registers in L were updated with c before they are read, then they would report c either from their *curr* or their *frozen* field, and clearly c would be safe. Therefore, c is missing from $t+1$ correct registers. Thus, WRITE($c.val$)'s write phase (lines 9–10) does not precede READ^k's read phase (lines 3–4). By the transitivity of the precedence relation, INC^k (line 2) precedes GET (line 11). By the safety of the counter, WRITE($c.val$) reads view k, a contradiction. □

Theorem 1 (Robustness). *The algorithm in Figure 3 wait-free implements a regular register.*

Proof. Immediately follows from Lemma 1 and 2.

4 A Robust and Amnesic Algorithm with Optimal READ-Complexity and Resilience

Similar to the previous section, we describe an initial version of DMS3 that uses a safe counter. The algorithm requires more rounds than the optimum but it is easier to understand because most of its complexity is hidden in the counter implementation. Then, we overview the changes necessary to obtain the optimal algorithm. The full details of the optimized DMS3 such as the pseudocode and proofs can be found in our technical report [19]. We proceed in a bottom-up fashion and describe the counter implementation first.

4.1 A Safe Counter with Optimal Resilience

We present a safe counter with operations INC and GET using $3t+1$ base objects $i \in \{1 \ldots n\}$, where t base objects can be subject to NR-arbitrary failures. The types and shared objects used by the counter are depicted in Figure 5 and the algorithm appears in Figure 6. Each base object i implements two regular registers: a register T_i holding a timestamp written by GET and read by INC, and a second register Y_i consisting of two fields *pw* and *w*, modified by INC and read by GET. While the *pw* field stores only the counter value, the *w* field stores the counter value together with a *high-resolution timestamp* [20]. A high-resolution timestamp is a timestamp-array with n entries, one for each base object.

The GET operation performs in two phases. The first phase reads from the base objects until $n-t$ registers Y_i have responded and all responses are *non-conflicting*. This condition is captured by the predicate conflict. When two base

Additional Types:

TSs $\triangleq$ *Integers* array of size n, *Integers*$[n]$

TSsInt $\triangleq$ *TSs* $\times$ *Integers* with selectors *hrts* (high-resolution timestamp) and *cnt*

Shared objects:

- regular registers $Y_i \in$ *Integers* $\times$ *TSsInt* with selectors *pw* and *w*, initially $Y_i = \langle 0, \langle [0, \ldots, 0], 0 \rangle \rangle$
- regular registers $T_i \in$ *Integers*, initially 0

Fig. 5. Shared objects used by the safe counter $(3t + 1)$

objects i and j are in conflict, then at least one of them is malicious. In this situation, the GET operation can wait for more than $n - t$ responses without blocking, effectively filtering out responses from malicious base objects. Next, the GET operation uses the responses to build a candidate set from values appearing in the w field of Y_i. In the second phase, the GET operation chooses an increasing timestamp ts and overwrites $n-t$ registers T_i with ts; at the same time it re-reads the registers Y_i until $n - t$ of them have responded and there exists a *candidate* to return. This condition is captured by the predicates safe and highCand. If no candidate can be returned (because of overlapping INC operations), GET returns the initial counter value 0.

Similarly, the INC operation performs in two phases, a pre-write and a write phase. The pre-write phase accesses $n-t$ base objects i, overwriting the pw field of Y_i with an increasing counter value and reading the individual timestamps stored in T_i into a single high-resolution timestamp. Subsequently, in the write phase, INC stores the counter value together with the high-resolution timestamp in the w field of $n - t$ registers Y_i and returns.

We now show that the algorithm in Figure 6 wait-free implements a safe counter. We do this by showing that the two following properties are satisfied:

Validity: If GET returns k then GET does not precede INC^k.

Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Lemma 3 (Validity). *The counter object implemented in Figure 6 is valid.*

Proof. If the initial value is returned then we are done. Else only a value $c.cnt = k$ is returned such that c is safe. This implies that $t + 1$ base objects report values k or higher either from their pw or w fields. As not all of them are faulty, there exists a correct object Y_i and a value $l \geq k$ such that l was indeed written to Y_i. As INC^k precedes INC^l (or it is the same operation) and GET does not precede INC^l, it follows that GET does not precede INC^k. □

Lemma 4 (Safety). *The counter object implemented in Figure 6 is safe.*

Proof. Let INC^k be the last operation preceding the invocation of GET. Furthermore, for all $l > k$, GET precedes INC^l. By assumption, $c.cnt = k$ was written to

```
Local variables (INC):
    $y \in \mathit{Integers} \times \mathit{TSsInt}$, initially $\langle 0, \langle [0, \ldots, 0], 0 \rangle \rangle$
    $\mathit{cnt} \in \mathit{Integers}$, initially 0   //counter value
    $\mathit{hrts}[1 \ldots n] \in \mathit{Integers}$, initially $[0, \ldots, 0]$   //high-resolution timestamp
INC()
    $\mathit{cnt} \leftarrow \mathit{cnt} + 1$
    $y.pw \leftarrow \mathit{cnt}$
    for $1 \leq i \leq n$ do invoke $\mathit{hrts}[i] \leftarrow \mathit{write\&read}(\langle Y_i, y \rangle,\ T_i)$
    wait for $n - t$ responses
    $y.w.\mathit{hrts} \leftarrow \mathit{hrts}$
    $y.w.\mathit{cnt} \leftarrow \mathit{cnt}$
    for $1 \leq i \leq n$ do invoke $\mathit{write}(Y_i, y)$
    wait for $n - t$ responses
    return ack

Predicates (GET):
    $\mathsf{conflict}(i, j) \triangleq y[i].w.\mathit{hrts}[j] \geq \mathit{ts}$
    $\mathsf{safe}(c) \triangleq |\{i : \max\{PW[i]\} \geq c.\mathit{cnt} \lor (\exists c' \in W[i] \land c'.\mathit{cnt} \geq c.\mathit{cnt})\}| > t$
    $\mathsf{highCand}(c) \triangleq c \in C \land (c.\mathit{cnt} = \max\{c'.\mathit{cnt} : c' \in C\})$
Local variables (GET):
    $PW[1 \ldots n] \in 2^{\mathit{Integers}}$, $W[1 \ldots n] \in 2^{\mathit{TSsInt}}$, $C \in 2^{\mathit{TSsInt}}$
    $y[1 \ldots n] \in \mathit{Integers} \times \mathit{TSsInt} \cup \{\bot\}$
    $\mathit{ts} \in \mathit{Integers}$, initially 0
GET()
    for $1 \leq i \leq n$ do $y[i] \leftarrow \bot$; $PW[i] \leftarrow W[i] \leftarrow \emptyset$
    $C \leftarrow \emptyset$
    $\mathit{ts} \leftarrow \mathit{ts} + 1$
    for $1 \leq i \leq n$ do invoke $y[i] \leftarrow \mathit{read}(Y_i)$
    repeat
        CHECK
    until a set $S$ of $n - t$ objects responded $\land\ \forall i, j \in S : \neg\mathsf{conflict}(i, j)$
    $C \leftarrow \{y[i].w : |\{j : y[j].w \neq y[i].w\}| \leq 2t\}$
    for $1 \leq i \leq n$ do invoke $y[i] \leftarrow \mathit{write\&read}(\langle T_i,\ \mathit{ts} \rangle,\ Y_i)$
    repeat
        CHECK
        $C \leftarrow C \setminus \{c \in C : |\{i : \exists c' \in W[i] \land c' \neq c\}| \geq 2t + 1\}$
    until $n - t$ responded $\land\ \exists c \in C$: $(\mathsf{safe}(c) \land \mathsf{highCand}(c)) \lor C = \emptyset$
    if $C \neq \emptyset$ then return $c.\mathit{cnt}$ else return 0

CHECK
    if $Y_i$ responded then
        $PW[i] \leftarrow PW[i] \cup \{y[i].pw\}$
        $W[i] \leftarrow W[i] \cup \{y[i].w\}$
```

Fig. 6. Safe counter algorithm $(3t + 1)$

the w field of $t+1$ correct objects before GET is invoked. Therefore, c is added to the candidate set C (line 16) and because at most $2t$ objects respond without c, it is never removed. Furthermore, $t+1$ correct objects eventually report c in the second GET round and c becomes safe. As there are no concurrent INC operations, eventually $2t+1$ correct objects report values k or lower from their w field and hence all c_h where $c_h.cnt > k$ are removed from C. Thus, c eventually becomes both safe and highCand and $c.cnt = k$ is returned. □

Lemma 5 (Wait-freedom). *The counter object implemented in Figure 6 is wait-free.*

Proof. As the INC operation never waits for more than $n-t$ responses, clearly it never blocks. In the following we prove that the GET operation does not block (1) at line 15 and (2) at line 21. We assume by contradition that the GET operation blocks. Case (1): as the GET operation never updates a correct base object with ts before the second round, correct base objects are never in conflict with each other and thus the GET operation does not block at line 15. Case (2): The GET operation blocks at line 21. Therefore, there exists $c \in C$ and c is not safe. Let $c.cnt = k$. If some correct base object has reported c in its w field in the first round of GET, then $t+1$ correct base objects report k or higher in their pw field in the second round and thus c is safe. Therefore, we assume that no correct base object reports c in w in the first round. If no correct object reports c in w in the second round, then $2t+1$ correct base objects respond with $c' \neq c$ in their w field and c is removed from C. In the following we assume that some correct object reports c in w in the second round. Let F ($|F| > 0$) denote the set of faulty objects that report c in their w field in the first round. Let X ($|X| \geq 0$) be the set of correct base objects i such that Y_i reports to the second GET round a value lower than k in both fields pw and w. This implies that the pre-write phase of INC at Y_i does not precede the second GET round reading Y_i (see Fig. 7 (a)). By the semantics of *write&read*, the second GET round has updated T_i with *ts before* reading Y_i (line 17). Similarly, the first round of INC has pre-written k to Y_i *before* reading T_i (line 3). By transitivity, the second GET round has completed the update of T_i *before* the first INC round has read T_i, and thus T_i reports ts (Fig. 7 (a)). Let $X' = \{j \in X : c.hrts[j] = ts\}$, that is, the objects in X that have actually responded to the first INC round. Note that for all $i \in F$ and for all $j \in X'$, conflict(i, j) is true. Hence, the $2t+1-|F|$ objects that have responded without c in their w field in the first round of GET do not include any object in X'. Overall, after the second GET round, $2t+1-|F|+|X'|$ base objects have responded without c in their w field. If $|F| \leq |X'|$ then c is removed from the set of candidates C (line 20), a contradiction. Therefore, we consider the case $|F| > |X'|$. Out of the $t+1$ correct base objects updated by the pre-write phase of INC, $t+1-|X'|$ respond with a timestamp lower than ts. Consequently, for every such base object i, GET has completed updating T_i with *ts not before* INC reads T_i (see Figure 7 (b)). By the semantics of *write&read* and by the transitivity of the precedence relation, register Y_i has stored k in its pw field before the second GET round reads Y_i. Hence, at least $t+1-|X'|+|F|$

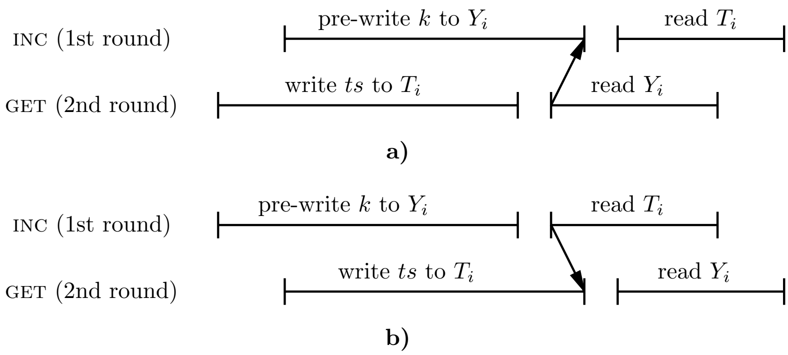


Fig. 7. Safe counter correctness argument

base objects report values k or higher. As $|F| > |X'|$, $t+1$ base objects report k or a higher value, and thus c is safe, a contradiction. □

Theorem 2. *The Algorithm in Figure 6 wait-free implements a safe counter.*

Proof. Follows directly from Lemma 3, 4 and 5. □

4.2 The DMS3 Protocol

Protocol Description

In this section we present a robust and amnesic SRSW register construction from a safe counter and $3t+1$ regular base registers, out of which t can be subject to NR-arbitrary failures. We now describe the WRITE and READ operations of the DMS3 algorithm illustrated in Figure 8.

The WRITE operation performs in three phases, (1) a pre-write phase (lines 7–9) where it stores a timestamp-value pair c in the *pw* field of $n-t$ registers, (2) a read phase (line 10), where it calls GET to read the current view and (3) a write phase (lines 14–16), where it overwrites the w field of $n-t$ registers with c. If the read phase results in a view change, the most recent value previously written is frozen together with the new view. This is done by updating the *view* field and copying the value stored in w to the *frozen* field (lines 11–13). The reader performs exactly the same steps as in DMS (see Section 3).

We now explain with help of Figure 9 why READs are wait free. Similar to the description of DMS in Section 3, we consider READ^k and the last WRITE that reads a view lower than k. Note that INC^k does not precede GET and thus, c is stored in the *pw* field of $t+1$ correct registers before they are read. Also, the w field of $t+1$ correct registers is updated with c. As the subsequent WRITE encounters a view change, c is written to the *frozen* field of $t+1$ correct registers, where it stays until READ^k completes. Hence, c is sampled from $t+1$ correct registers' *pw, w* or *frozen* field and thus it is safe. Note that c is also highestCand because only faulty registers report newer values.

```
Shared objects:
    regular registers $X_i \in TSVals^3 \times Integers$, with selectors $pw$, $w$, $frozen$ and
    $view$, initially $X_i = \langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$
Predicates (reader):
    $\mathsf{readFrom}(c, i) \triangleq (c = x[i].w \wedge x[i].view < view) \vee$
                          $(c = x[i].frozen \wedge x[i].view = view)$
    $\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].pw, x[i].w, x[i].frozen\}\}| \geq t + 1$
    $\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.ts \leq c.ts\}| \geq 2t + 1$
Local variables (reader):
    $view \in Integers$, initially 0
    $x[1 \ldots n] \in TSVals^3 \times Integers \cup \{\bot\}$
READ()
    for $1 \leq i \leq n$ do $x[i] \leftarrow \bot$
    $view \leftarrow$ INC$(Y)$
    for $1 \leq i \leq n$ do invoke $x[i] \leftarrow read(X_i)$
    wait until $n - t$ responded $\wedge\ \exists c \in TSVals$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$
    return $c.val$

Local variables (writer):
    $ts$, $newView \in Integers$, initially 0
    $x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$
WRITE$(v)$
    $ts \leftarrow ts+1$
    $x.pw \leftarrow \langle ts, v\rangle$
    for $1 \leq i \leq n$ do invoke $write(X_i, x)$
    wait for $n - t$ responses
    $newView \leftarrow$ GET$(Y)$
    if $newView > x.view$ then
        $x.view \leftarrow newView$
        $x.frozen \leftarrow x.w$
    $x.w \leftarrow \langle ts, v\rangle$
    for $1 \leq i \leq n$ do invoke $write(X_i, x)$
    wait for $n - t$ responses
    return $ack$
```

Fig. 8. Robust and amnesic storage algorithm DMS3 ($3t + 1$)

With DMS3, the high-level operations have a non-optimal time-complexity. We now explain how the optimized version is obtained by collapsing individual low-level operations. More precisely, a write operation and a consecutive read operation are merged together to a *write&read* operation. The safe counter abstraction is disregarded and the counter operations INC and GET are weaved into READ and WRITE respectively. Recall that the counter operations consist of two rounds each. In the WRITE implementation, the pre-write phase and the first round of GET are collapsed. Note that the three-phase structure of the WRITE is preserved in that the writer reads the current view *before* it moves to the write phase. Similarly, in the READ implementation, the second INC round and

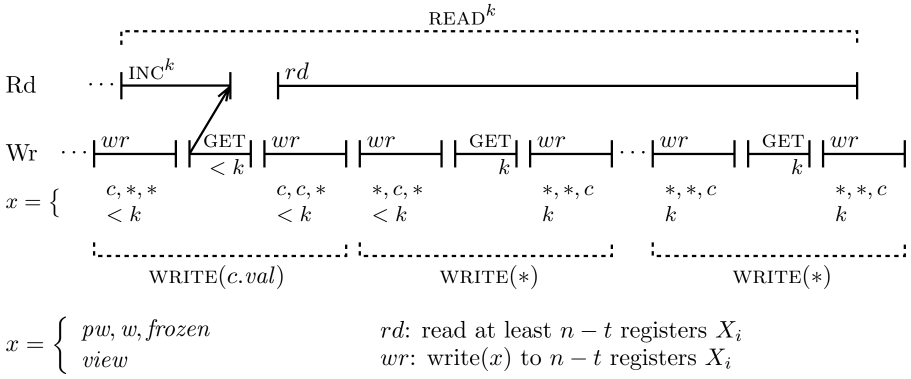


Fig. 9. Correctness argument of the READ operation in DMS3

the read phase are merged together. Overall, this results in a time-complexity of *three* rounds for the WRITE and *two* rounds for the READ.

We now informally argue that the optimization is correctness preserving. As above, we consider READ^k and the last WRITE that reads a view lower than k. We argue that $t + 1$ correct base registers have stored c in their pw field before any of them is read. This would imply that c is safe. The fact that the WRITE of $c.val$ reads a view lower than k implies that k is missing from at least $2t+1$ base objects. We know from the safe counter algorithm in the previous section that if only $2t$ base objects respond without k, then k is never removed from the set of candidates. As the safe counter implementation is wait-free, k is eventually read, contradicting the initial assumption. Therefore, $2t+1$ base objects respond without k, and thus there are $t + 1$ correct base objects among them that are accessed by (the read phase of) READ^k only after c was pre-written to them. By applying similar arguments as above, it is not difficult to see that c does not disappear from any of the $t + 1$ correct base objects before READ^k completes. This would imply that c eventually becomes safe. For a formal treatment we refer the interested reader to our full paper [19]. The remainder of this section is concerned with the correctness of DMS3.

Protocol Correctness

Lemma 6 (Regularity). *Algorithm DMS3 in Figure 8 implements a regular register.*

Proof. Identical to the proof of Lemma 1. □

Lemma 7 (Wait-freedom). *Algorithm DMS3 in Figure 8 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n-t$ responses. To derive a contradiction we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $2t+1$) have responded. We choose c as the highest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leq c.ts$ are readFrom $2t+1$ correct registers. In the following, we distinguish the cases where the view read by the WRITE of $c.val$ is equal to k (case 1) or it is lower than k (case 2). Note that by the validity of the counter, only views $\leq k$ are returned. Case 1: Let X_i be a correct register such that readFrom(c, i). Since by assumption $x[i].view = k$, c is readFrom the *frozen* field of X_i. However, in view k only timestamp-value pairs lower than c are frozen, a contradiction. Now we consider case 2, where the WRITE$(c.val)$ reads a view lower than k. This implies that INC^k does not precede GET. As the pre-write phase (lines 8–9) precedes GET (line 10), and INC^k (line 2) precedes the read phase (lines 3–4), by transitivity, the pre-write phase also precedes the read phase (see Figure 9). Thus, $t+1$ correct registers have stored c in their *pw* field *before* they are read. What is left to show is that no subsequent WRITE erases c from all fields of those $t+1$ correct registers. Note that in view k, only timestamp-value pairs c or higher a frozen. Thus, if c was stored in the w field of $t+1$ correct registers before they are read, then c would be safe. Hence, c is missing from $t+1$ correct registers' w field. Consequently, WRITE$(c.val)$'s write phase (lines 15–16) does not precede READ^k's read phase (lines 3–4). By transitivity, the subsequent WRITE reads view k and freezes c. Note that c is erased from *pw* only after c was previously stored in w (line 14). Furthermore, c is erased from w only after it was stored in *frozen* (line 13). As k is the last view, by the validity of the safe counter, c is never erased from *frozen*. □

Theorem 3 (Robustness). *Algorithm DMS3 in Figure 8 implements a robust register.*

Proof. Immediately follows from Lemma 6 and 7.

5 Concluding Remarks

We have presented amnesic algorithms that robustly implement a shared register from a collection of n base objects, of which up to $t < n/3$ can be subject to NR-arbitrary failures. For $n \geq 3t+1$ we have shown that *two* rounds of communication with the base objects are sufficient for every READ operation to complete. This is the first robust and amnesic register construction that matches the *two*-round lower bound proved in [10]. For the $n \geq 4t+1$ case, we have presented the first robust and amnesic register construction that matches the (trivial) *one*-round lower bound for *every* operation. Note that our construction is tight because with less than $4t+1$ base objects, both the READ and the WRITE operations require at least *two* communication rounds [2,10].

The main result of this paper, that robust access to amnesic storage is possible in optimal time is somewhat surprising given the large body of literature on non-amnesic [11,10,12,4,14,13] and non-robust [8,9,18,5] algorithms. Moreover, our result is counter-intuitive because so far, only non-amnesic algorithms match the time-complexity lower bounds. As a corollary, our result suggests that the intuition of amnesic algorithms being inherently less efficient than non-amnesic ones is largely unjustified.

Some of the prior amnesic (but not robust) register implementations assume that the readers cannot modify the base objects (see e.g. [2]). This assumption in fact results in implementations that possess several properties that could be valuable in practice, for instance the ability to tolerate any number of malicious readers while using only $\mathcal{O}(1)$ memory at the base objects. We are not aware of any robust implementation supporting that as well, and in fact, our algorithms are not an exception. We leave as an open problem the question whether robust and amnesic register implementations exist, that would support any number of readers while using only $\mathcal{O}(1)$ memory at the base objects.

Acknowledgments

We thank Gregory Chockler, Felix Freiling, Marco Serafini and Jay Wylie for many useful comments on an earlier version of this paper.

References

1. Lamport, L.: On interprocess communication. part II: Algorithms. Distributed Computing 1(2), 86–101 (1986)
2. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing 18(5), 387–408 (2006)
3. Chockler, G., Malkhi, D.: Active disk paxos with infinitely many processes. Distributed Computing 18(1), 73–84 (2005)
4. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal Byzantine Storage. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 311–325. Springer, Heidelberg (2002)
5. Jayanti, P., Chandra, T.D., Toueg, S.: Fault-tolerant wait-free shared objects. J. ACM 45(3), 451–500 (1998)
6. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
7. Chockler, G., Guerraoui, R., Keidar, I.: Amnesic Distributed Storage. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 139–151. Springer, Heidelberg (2007)
8. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead byzantine fault-tolerant storage. In: SOSP 2007: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 73–86. ACM, New York (2007)
9. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distrib. Comput. 11(4), 203–213 (1998)
10. Guerraoui, R., Vukolić, M.: How fast can a very robust read be? In: PODC 2006: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, pp. 248–257. ACM, New York (2006)
11. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient byzantine-tolerant erasure-coded storage. In: DSN 2004: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN 2004), Washington, DC, USA, pp. 135–144. IEEE Computer Society, Los Alamitos (2004)

12. Guerraoui, R., Vukolić, M.: Refined quorum systems. In: PODC 2007: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pp. 119–128 (2007)
13. Bazzi, R.A., Ding, Y.: Non-skipping timestamps for byzantine data storage systems. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 405–419. Springer, Heidelberg (2004)
14. Aiyer, A., Alvisi, L., Bazzi, R.A.: Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 7–19. Springer, Heidelberg (2007)
15. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded byzantine distributed storage. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), Washington, DC, USA, pp. 115–124. IEEE Computer Society, Los Alamitos (2006)
16. Liskov, B., Rodrigues, R.: Tolerating byzantine faulty clients in a quorum system. In: ICDCS 2006: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 34–43. IEEE Computer Society, Los Alamitos (2006)
17. Guerraoui, R., Levy, R.R., Vukolić, M.: Lucky read/write access to robust atomic storage. In: DSN 2006: Proceedings of the International Conference on Dependable Systems and Networks (DSN 2006), pp. 125–136 (2006)
18. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Wait-free regular storage from byzantine components. Inf. Process. Lett. 101(2) (2007)
19. Dobre, D., Majuntke, M., Suri, N.: On the time-complexity of robust and amnesic storage. Technical Report TR-TUD-DEEDS-04-01-2008, Technische Universität Darmstadt (2008), http://www.deeds.informatik.tu-darmstadt.de/dan/amnesicTR.pdf
20. Chockler, G., Rachid Guerraoui, I.K., Vukolic, M.: Reliable distributed storage. IEEE Computer (to appear, 2008)

Graph Augmentation via Metric Embedding

Emmanuelle Lebhar[1] and Nicolas Schabanel[2]

[1] CNRS and CMM-Univesidad de Chile, Chile
elebhar@dim.uchile.cl
[2] CNRS and Université Paris Diderot, France
Nicolas.Schabanel@liafa.jussieu.fr

Abstract. Kleinberg [17] proposed in 2000 the first random graph model achieving to reproduce small world navigability, i.e. the ability to greedily discover polylogarithmic routes between any pair of nodes in a graph, with only a partial knowledge of distances. Following this seminal work, a major challenge was to extend this model to larger classes of graphs than regular meshes, introducing the concept of *augmented graphs navigability*. In this paper, we propose an original method of augmentation, based on metrics embeddings. Precisely, we prove that, for any $\varepsilon > 0$, any graph G such that its shortest paths metric admits an embedding of distorsion γ into $\mathbb{R}^d$ can be augmented by one link per node such that greedy routing computes paths of expected length $O(\frac{1}{\varepsilon}\gamma^d \log^{2+\varepsilon} n)$ between any pair of nodes with the only knowledge of G. Our method isolates all the structural constraints in the existence of a good quality embedding and therefore enables to enlarge the characterization of augmentable graphs.

Keywords: Small world, metrics embedding, greedy routing.

1 Introduction

The small world effect, or "six degrees of separation", is the well known property observed in social networks [9,21] that any pair of nodes in these networks is connected by a very short chain of acquaintances (typically polylogarithmic in the size of the network), that, moreover, can be discovered locally. In the literature, a *small world graph* can either refer to this property or to a graph with polylogarithmic diameter and high clustering (see e.g. [23]). In this paper, a small world graph refers to a graph of polylogarithmic diameter *and* whose short paths can be discovered locally, i.e. which is *navigable*. This surprising property has gained a lot of interest recently since Kleinberg [17] introduced the first analytical graph model for navigability, and because of its potential in the design of large decentralized networks with efficient routing schemes. The model proposed by Kleinberg in 2000 consists in a d-dimensional mesh augmented by one extra random link in each node, distributed according to the d-harmonic distribution. The local search is then modeled by greedy routing, which is the simple algorithm that, at each node, forwards the message to the neighbor that is the

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 217–225, 2008.

closest to the destination *in the mesh*. Kleinberg demonstrates that greedy routing computes paths of expected length $\Theta(\log^2 n)$ between any pair of nodes in his model, with the only knowledge of the distances in the mesh: the augmented mesh is *navigable*

Following this seminal work, a major challenge was to extend this model to larger classes of graphs than regular meshes, i.e. to determine which n-node graphs G admit an augmentation with one link in each node such that greedy routing with the only of G computes polylog(n)-length paths between any pair in the augmented graph. Kleinberg [18] and Duchon et al. [7] showed that this is possible for all graphs of bounded growth, i.e. where, for any node u and radius $r \geq 1$, the $2r$-neighborhood of u is of size at most a constant times its r-neighborhood. Fraigniaud [10] demonstrates that any bounded treewidth graph can also be augmented by one link per node to become navigable, and Abraham and Gavoille [4] showed that, more generally, this is possible for all graphs excluding a fixed minor. The definition of the problem can directly be extended to metric spaces by asking which n-points metric spaces[1] $M = (V, \delta)$ can be augmented by $O(\log n)$ links such that, in the resulting graph, greedy routing computes polylog(n) routes between any pair with the only knowledge of M. In this framework, Slivkins [22] showed that any doubling metric can be augmented to become navigable. A doubling metric is a metric where, for all $r \geq 1$, any ball of radius $2r$ can be covered by at most C balls of radius r, for some constant C.

However, it was recently proven by Fraigniaud et al. [13] that such an augmentation is not possible for all graphs: there exist an infinite familiy of n-node graphs on which any distribution of augmented links will leave the greedy paths of expected length $\Omega(n^{1/\sqrt{\log n}})$ for some pairs. The best upper bound valid for arbitrary graphs up to our knowledge is an expected length $\tilde{O}(n^{1/3})$ between any pair, due to Fraigniaud et al. [12], with some specific link augmentation. The remaining gap between these two bounds is today still open and leaves a question mark on the limiting characteristics of a metric for the navigability augmentation.

Orthogonally to the navigability question, studies on embeddings of metric spaces have known huge developments this last decade (cf. Chapter 15 of [20] for a review), due in particular to their applications in approximation algorithms [15] and more recently in handling efficiently large decentralized networks [6]. An embedding σ of a metric $M = (V, \delta)$ into a metric $M' = (V', \delta')$ is an injective function σ on V into V'. Its quality is characterized by the distorsion it induces on the distances. For the sake of simplicity, we consider only non-contracting embeddings, we then say that σ has distorsion γ if and only iff for any $u, v \in V$, $\delta'(\sigma(u), \sigma(v)) \leq \gamma \cdot \delta(u, v)$. Crucial networking problems like routing, resource location or nearest neighbor search are easy to handle on a low dimensional euclidean space. However, large real networks like the Internet do not present

[1] A metric space $M = (V, \delta)$ is a set of points V associated with a distance function δ. Therefore, any weighted graph naturally defines a metric M on its set of nodes V with the distance function δ being the length of a shortest path between two nodes.

such a simple structure. The increasing interest for metrics embeddings comes therefore partially from the fact that, if the embedding is of good quality, it can provide a way to develop efficient algorithms on complex, or even arbitrary, metric spaces, by solving them on a simple metric space that approximates them well (cf. e.g. [14,15]). In addition, many good quality embeddings are computed with randomized local algorithms that only require a distance oracle, making them particularly appropriate to the large decentralized networks setting (cf. e.g. [5] for a seminal example).

In this paper, we propose a new way to tackle the augmented graphs navigability problem through the metric embedding setting.

1.1 Our Contribution

We introduce a generalized augmentation process. The main feature of our augmentation process is to use an embedding of the input graph shortest paths metric into a metric that is easy to augment into a navigable graph. This distinction between the augmentation process in itself (handled on the "easy" metric) and the structural characteristics of the input (captured by the embedding quality) provides a new way to characterize the classes of navigable graphs. We consider embedding into $(\mathbb{R}^d, \ell_p)$ which is the d-dimensional euclidean space associated to the ℓ_p norm, for $d, p \geq 1$: for any $u = (u_1, \ldots, u_d)$ and $v = (v_1, \ldots, v_d)$, we have $||u - v||_p = (\sum_{i=1}^{d} |u_i - v_i|^p)^{1/p}$. We prove the following theorem:

Theorem 1. *Let $p, n, \gamma, d \geq 1$. For any $\varepsilon > 0$, any n-node graph G whose shortest path metric $M = (V, \delta)$ admits an embedding of distorsion γ into $(\mathbb{R}^d, \ell_p)$ can be augmented with one link per node such that greedy routing in the resulting graph computes paths of expected length $O(\frac{1}{\varepsilon}\gamma^d \log^{2+\varepsilon} n)$ between any pair with the only knowledge of M.*

For instance, using the recent embedding result of Abraham et al. [3], we get as an immediate corollary that, for any $0 < \varepsilon \leq 1$ and any $n \geq 1$, any n-node graph G of doubling dimension D (cf. [14]) can be augmented so that the expected lengths of all greedy paths is $O((\log^{(1+\varepsilon)} n)^{O(D/\varepsilon)} \log^2 n) = O((\log n)^{O(D)})$ with the only knowledge of G. This provides a more direct proof to the fact that bounded doubling dimension graphs are navigable (proved in [22]).

Intuitively, if the metric considered is not too far from a metric M which can be easily augmented, we use a low distorsion embedding of the metric into M, draw the random links in M, and then map back appropriately these links to the original metric so that they will still be useful shortcuts for greedy routing.

Moreover, the design of the augmented links in our process can be done in a fully decentralized way and only requires to know the embedding. In the case where the chosen embedding is itself local (like e.g. the seminal Bourgain embedding [5] if a distance oracle is available), we thus provide an algorithm which locally adds one address to each routing table in a network and guarantees a small number of hops decentralized routing between any pair for a large class of input graphs.

2 A Universal Augmentation Process via Metric Embedding

In this section, we present our augmentation process that adds one directed link per node. This process is universal in the sense that it only requires as an input the base graph (arbitrary) and an embedding function of this graph into $\mathbb{R}^d_{\ell_p}$, for some $p, d \geq 1$. Such a function exists for any graph and therefore the algorithm is not restricted to a specific graph class. However, as we will see in the next section, the analysis of greedy routing might give a poor routing time result if the embedding is not of good quality. There exists lower bound results on arbitrary metric embedding quality. A typical example is that embedding some n-node constant degree expander graph into $\mathbb{R}^d_{\ell_p}$ requires distortion $\Omega(\log n)$ [20] and dimension $d = \Omega(\log n)$ [2]. Nevertheless, expander graphs are always navigable without any augmentation given their polylogarithmic diameter.

The augmentation algorithm is based on the well known augmentation of d-dimensional meshes of the Kleinberg model, where the shortcuts are distributed according to the d-harmonic distribution. The idea is to map back these links to the original set of nodes. Given that not all the extremities of the shortcuts added in ℓ_p^d are images of the original nodes, this requires some careful rewiring.

```
Augmentation Process $\mathcal{AP}$
Input: An $n$-node graph $G = (V, E)$, an embedding $\sigma$ of its shortest path metric $M = (V, \delta)$ into $\ell_p^d$, and a constant $\varepsilon > 0$;
Output: $G$ augmented with one directed link in each node.
Begin
   For each $u \in V$ do
   Pick a point $\tau_u \in \mathbb{R}^d_{\ell_p}$ with probability density:
      $\frac{1}{Z} \frac{1}{\left(||\sigma(u) - \tau||_p\right)^d \ln^{1+\varepsilon}(||\sigma(u) - \tau||_p + e)}$,
   over all $\tau \in \mathbb{R}^d_{\ell_p}$.
    Add a directed link from $u$ to $v \in V$ where $v$ is the node such that $\sigma(v)$ is the closest point to $\tau_u$ in $\sigma(V)$.
End.
```

Note: e stands here for exp(1) and is only used to allow distance to be zero in the formula. Z is the normalizing factor of the probability density described: $Z = \int_{t>0}^{\infty} \frac{S(t)}{t^d \ln^{1+\varepsilon}(t+e)} dt$, where $S(t)$ is the surface of an hypersphere of raius t in $\mathbb{R}^d$. Figure 1 illustrates the process $\mathcal{AP}$.

3 Navigability of Graphs Augmented with $\mathcal{AP}$

In this section, we demonstrate our main result. The *intrinsic dimension* [3], or *doubling dimension* [1] of a graph G characterizes its geometric property, this is

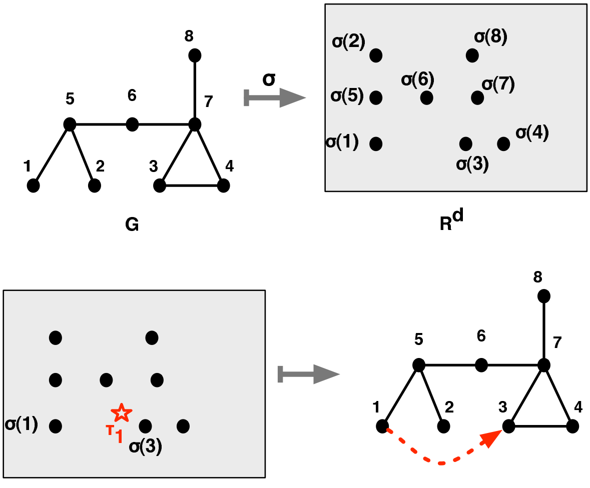


Fig. 1. Illustration of the augmented link from vertex 1 to 3 with process $\mathcal{AP}$

the minimum constant α such that any ball in G can be covered by at most 2^{α} balls of half the radius. We show that, if a graph has low intrinsic dimension, $\mathcal{AP}$ process provides augmented shortcuts that enables navigability. We have the following theorem:

Theorem 2. *Let $p, n, \gamma, d \geq 1, \varepsilon > 0$, G an n-node graph and σ an embedding of distorsion γ of the shortest path metric M of G into $(\mathbb{R}^d, \ell_p)$. Then, greedy routing in $\mathcal{AP}(G, \sigma, \varepsilon)$ computes paths of expected length at most $O(\frac{1}{\varepsilon}\gamma^d \log^{2+\varepsilon} n)$ between any pair, with the only information of the distances in M.*

Proof. In order to analyze greedy routing performances in $\mathcal{AP}(G, \sigma, \varepsilon)$, we begin by analyzing some technical properties of the probability distribution of the chosen points τ in $(\mathbb{R}^d, \ell_p)$. For any $u \in G$, we say that τ_u, as defined in algorithm $\mathcal{AP}$, is the *contact point* of u.

Let Z be the normalizing factor of the contact points distribution. We have:

$$Z = \int_{t>0}^{\infty} \frac{S(t)}{t^d \ln^{1+\varepsilon}(t+e)} dt,$$

where $S(t)$ stands for the surface of a sphere of radius t in $\mathbb{R}^d_{\ell_p}$. This surface is at most $c_p \cdot \big(2^d/(d-1)!\big) \cdot t^{d-1}$, where $c_p > 1$ is a constant depending on p. It follows:

$$Z \leq c_p \cdot \frac{2^d}{(d-1)!} \int_{t>1}^{\infty} \frac{dt}{t \ln^{1+\varepsilon}(t+e)} \leq c_p \cdot \frac{(1+e)}{\varepsilon} \cdot \frac{2^d}{(d-1)!}.$$

Let s and $t \in G$ be the source and the target of greedy routing in $\mathcal{AP}(G, \sigma, \varepsilon)$. Let $M = (V, \delta)$ be the shortest paths metric of G. Let v be the current node of greedy routing, and let $1 \leq i \leq \lceil \log \delta(s,t) \rceil$ such that $\delta(v,t) \in [2^{i-1}, 2^i)$.

Since σ has distorsion γ, we have:

$$\delta(v,t) \leq ||\sigma(v) - \sigma(t)||_p \leq \gamma \cdot \delta(v,t).$$

Let $X = ||\sigma(v) - \sigma(t)||_p$, and let $\mathcal{E}$ be the event: "$||\tau_v - \sigma(t)||_p \leq X/(4\gamma)$". Let $L(v)$ be the contact of v (i.e. the closest point to τ_v in $\sigma(V)$.

Claim. If $\mathcal{E}$ occurs, then $\delta(L(v),t) \leq \delta(v,t)/2$.

Indeed, assume that $\mathcal{E}$ occurs. From the triangle inequality, we have:

$$||\sigma(L(v)) - \sigma(t)||_p \leq ||\sigma(L(v)) - \tau_v||_p + ||\tau_v - \sigma(t)||_p.$$

And since $\sigma(L(v)) = \tau$ is closer to τ_v than $\sigma(t)$ by definition of $\mathcal{AP}$, we get:

$$||\sigma(L(v)) - \sigma(t)||_p \leq 2||\tau_v - \sigma(t)||_p \leq X/(2\gamma).$$

Finally:

$$\delta(L(v),t) \leq ||\sigma(L(v)) - \sigma(t)||_p \leq X/(2\gamma) \leq \delta(v,t)/2. \qquad \diamond$$

Claim. The probability that $\mathcal{E}$ occurs is greater than

$$C\,\frac{\varepsilon}{d5^d\gamma^d}\,\frac{1}{\ln^{1+\varepsilon}(2\gamma\delta(v,t)+e)},$$

for some constant $C > 0$.

Proof of the claim. Let P be the probability that $\mathcal{E}$ occurs. P is the probability that τ_v belongs to the ball of radius $X/(4\gamma)$ centered at $\sigma(t)$ in $\mathbb{R}^d_{\ell_p}$. Let $\mathcal{B}$ be this ball. We have, by definition of $\mathcal{AP}$:

$$\begin{aligned} P &= \frac{1}{Z}\int_{\tau\in\mathcal{B}} \frac{1}{(||\sigma(v)-\tau||_p)^d \ln^{1+\varepsilon}(||\sigma(v)-\tau||_p + e)} \\ &\geq \frac{1}{Z}\int_{\tau\in\mathcal{B}} \frac{1}{\big((1+1/(4\gamma))X\big)^d \ln^{1+\varepsilon}((1+1/(4\gamma))X+e)}, \end{aligned}$$

since $(1+1/(4\gamma))X$ is the largest distance from $\sigma(v)$ to any point in $\mathcal{B}$.

On the other hand, the volume of $\mathcal{B}$ is at least $c'_p \cdot \frac{2^d}{d!}(X/(4\gamma))^d$, for some constant $c'_p > 0$. We get:

$$\begin{aligned} P &\geq \frac{1}{Z}\cdot\frac{c'_p 2^d (X/4\gamma)^d}{d!(1+\frac{1}{4\gamma})^d X^d}\cdot\frac{1}{\ln^{1+\varepsilon}((1+\frac{1}{4\gamma})X+e)} \\ &\geq \frac{c'_p}{c_p(1+e)}\cdot\frac{\varepsilon}{d5^d}\cdot\frac{1}{\gamma^d}\cdot\frac{1}{\ln^{1+\varepsilon}((1+\frac{1}{4\gamma})X+e)} \\ &\geq C\,\frac{\varepsilon}{d5^d\gamma^d}\,\frac{1}{\ln^{1+\varepsilon}(2\gamma\delta(v,t)+e)}. \qquad \diamond \end{aligned}$$

Claim. If the current node v of greedy routing satisfies $\delta(v,t) \in [2^{i-1}, 2^i)$ for some $1 \leq i \leq \lceil \log\delta(s,t)\rceil$, then after $O(\frac{1}{\varepsilon}\gamma^d(i-1)^{1+\varepsilon})$ steps on expectation, greedy routing is at distance less than 2^{i-1} from t.

Proof of the claim. Combining the claims, we get that, with probability $\Omega([\frac{1}{\varepsilon}\gamma^d \ln^{1+\varepsilon}(\gamma\delta(v,t))]^{-1})$ (where the Ω notation hides a linear factor in ε), the contact $L(v)$ of v is at distance at most 2^{i-1} to t. If this does not occur, greedy routing moves to a neighbor v' at distance strictly less than $\delta(v,t)$ to t and strictly greater than 2^{i-1} and we can repeat the same argument. Therefore, after $O(\frac{1}{\varepsilon}\gamma^d \ln^{1+\varepsilon}(\gamma\delta(v,t))) = O(\frac{1}{\varepsilon}\gamma^d (i-1)^{1+\varepsilon})$ steps, greedy routing is at distance less than 2^{i-1} to t with constant probability. ◇

Finally, from this last claim, the expected number of steps of greedy routing from s to t is at most:

$$\sum_{i=1}^{\log(\delta(s,t))} O(\gamma^d (i-1)^{1+\varepsilon}) = O(\frac{1}{\varepsilon}\gamma^d \log^{2+\varepsilon} n).$$

From this theorem, results giving new insights on the navigability problem can be derived from the very recent advances in metric embeddings theory. In particular, graphs of bounded doubling dimension, that subsumes graphs of bounded growth, received an increasing interest recently. They are of particular interest for scalable and distributed network applications since it is possible to decompose them greedily into clusters of exponentially decreasing diameter.

Corollary 1. *For any $\varepsilon > 0$, any n-node graph G of bounded doubling dimension α can be augmented with one link per node so that greedy routing compute paths of expected length $O(\frac{1}{\varepsilon}\log^{(2+\varepsilon+2\alpha)} n)$ between any pair of vertices with the only knowledge of G.*

Indeed, from Theorem 1.1 of [3], it is known that, for every n-point metric space M of doubling dimension α and every $\theta \in (0,1]$, there exists an embedding of M into $\mathbb{R}^d_{\ell_p}$ with distorsion $O(\log^{1+\theta} n)$ and dimension $O(\alpha/\theta)$. Taking $\theta = 1$ gives the corollary. This result was previously proved in [22] by another method of augmentation, using "rings of neighbor". The originality of our method is that it is not specific to a given graph or metric class, this dependency lying only in the embedding function. Therefore, it enables to get more direct proofs that a graph is augmentable into a navigable small world than previous ones.

This new kind of augmentation process via embedding is also promising to derive lower bounds on metrics embedding quality. Indeed, since not all graphs can be augmented to become navigable, necessarily, if there exists a positive result on small world augmentation via some embedding, then this embedding cannot keep the same quality for all graphs. For the particular case of Theorem 2, we derive that any injective function σ that embeds any arbitrary metric into $\mathbb{R}^d_{\ell_p}$ with distorsion γ has to satisfy $\gamma^d = \tilde{\Omega}(n^{1/\sqrt{\log n}})$. This lower bound is however subsumed by the bound provided by the Johnson-Lindenstrauss flattening lemma [16]: $\gamma^d = O((1+\varepsilon)^{\log n/\varepsilon^2}) = O(n^{(1+\varepsilon)/\varepsilon^2})$ for any $0 < \varepsilon < 1$, which is essentially tight (cf. e.g. [20]).

It is worth to note that Fraigniaud and Gavoille [11] recently tackled the question of navigating in a graph that has been augmented using the distances in a spanner[2] of this graph. They remarked that greedy routing usually requires to know the spanner map of distances in order to achieve an efficient routing. On the contrary, our augmentation process does not requires greedy routing to be aware of distances in $\mathbb{R}^d$. This is due to the geography of the spaces considered: an embedding of a graph in $\mathbb{R}^d$ preserves geographical neighboring regions.

4 Discussion

The result presented in this paper gives new perspectives in the understanding of networks small world augmentations. Indeed, the augmentation process $\mathcal{AP}$ isolates all the dependencies on the graph structure in the embedding function.

On the other hand, such an augmentation process focuses on the geography of the graph and cannot capture the augmentation processes that are based on graph separator decomposition. It can be distinguished two main kinds of augmentation processes in the navigable networks literature. One kind of augmentation relies on the graph density and its similarity with a mesh (like augmentations in [7,17,18,22]), while the other kind relies on the existence of good separators in the graph (like augmentations in [4,10]). Augmentation via embedding cannot be directly extended to augmentations using separators because of the difficulty to handle the distortion in the analysis of greedy routing. Finally, the extension of $\mathcal{AP}$ to graphs that are close to a tree metric (using embeddings into tree metrics) could open the path to the exhaustive characterization of graph classes that can be augmented to become navigable, as well as provide new lower and upper bounds on embeddings as side results. More generally, the exhaustive characterization of the graphs that can be augmented to become navigable is still an important open problem, as well as the design of good quality embeddings into low dimensional spaces.

References

1. Assouad, P.: Plongements lipschitzien dans $\mathbb{R}^n$. Bull. Soc. Math. France 111(4), 429–448 (1983)
2. Abraham, I., Bartal, Y., Neiman, O.: Advances in metric embedding theory. In: Proceeeding of the the 38th annual ACM symposium on Theory of Computing (STOC), pp. 271–286 (2006)
3. Abraham, I., Bartal, Y., Neiman, O.: Embedding Metric Spaces in their Intrinsic Dimension. In: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA), pp. 363–372 (2008)
4. Abraham, I., Gavoille, C.: Object location using path separators. In: Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 188–197 (2006)
5. Bourgain, J.: On Lipschitz embedding of finite metric spaces in Hilbert space. Israel Journal of Mathematics 52, 46–52 (1985)

[2] A k-spanner of a graph G is a subgraph G' such that for any $u, v \in G$, $\mathrm{dist}_{G'}(u,v) \leq k\mathrm{dist}_G(u,v)$.

6. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: A decentralized network coordinate system. In: ACM SIGCOMM (2004)
7. Duchon, P., Hanusse, N., Lebhar, E., Schabanel, N.: Could any graph be turned into a small-world? Theoretical Computer Science 355(1), 96–103 (2006)
8. Duchon, P., Hanusse, N., Lebhar, E., Schabanel, N.: Towards small world emergence. In: 18th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA), pp. 225–232 (2006)
9. Dodds, P.S., Muhamad, R., Watts, D.J.: An experimental study of search in global social networks. Science 301, 827–829 (2003)
10. Fraigniaud, P.: Greedy routing in tree-decomposed graphs: a new perspective on the small-world phenomenon. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 791–802. Springer, Heidelberg (2005)
11. Fraigniaud, P., Gavoille, C.: Polylogarithmic network navigability using compact metrics with small stretch. In: 20th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA), pp. 62–69 (2008)
12. Fraigniaud, P., Gavoille, C., Kosowski, A., Lebhar, E., Lotker, Z.: Universal Augmentation Schemes for Network Navigability: Overcoming the $\sqrt{n}$-Barrier. In: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA), pp. 1–7 (2007)
13. Fraigniaud, P., Lebhar, E., Lotker, Z.: A Doubling Dimension Threshold $\Theta(\log\log n)$ for Augmented Graph Navigability. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 376–386. Springer, Heidelberg (2006)
14. Gupta, A., Krauthgamer, R., Lee, J.R.: Bounded geometries, fractals, and low-distortion embeddings. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 534–543 (2003)
15. Indyk, P.: Algorithmic aspects of geometric embeddings. In: Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science, FOCS (2001)
16. Johnson, W.B., Lindenstrauss, J.: Extensions of Lipschitz maps into a Hilbert space. Contemporary mathematics 26, 189–206 (1984)
17. Kleinberg, J.: The Small-World Phenomenon: An Algorithmic Perspective. In: 32nd ACM Symp. on Theo. of Comp. (STOC), pp. 163–170 (2000)
18. Kleinberg, J.: Small-World Phenomena and the Dynamics of Information. Advances in Neural Information Processing Systems (NIPS) 14 (2001)
19. Kleinberg, J.: Complex networks and decentralized search algorithm. In: Intl. Congress of Math, ICM (2006)
20. Matousek, J.: Lectures on Discrete Geometry. Graduate Texts in Mathematics, vol. 212. Springer, Heidelberg (2002)
21. Milgram, S.: The Small-World Problem. Psychology Today, 60–67 (1967)
22. Slivkins, A.: Distance estimation and object location via rings of neighbors. In: 24th Annual ACM Symp. on Princ. of Distr. Comp. (PODC), pp. 41–50 (2005)
23. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature 393, 440–442 (1998)

A Lock-Based STM Protocol That Satisfies Opacity and Progressiveness

Damien Imbs and Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France
{damien.imbs,raynal}@irisa.fr

Abstract. The aim of a software transactional memory (STM) system is to facilitate the delicate problem of low-level concurrency management, i.e. the design of programs made up of processes/threads that concurrently access shared objects. To that end, a STM system allows a programmer to write transactions accessing shared objects, without having to take care of the fact that these objects are concurrently accessed: the programmer is discharged from the delicate problem of concurrency management. Given a transaction, the STM system commits or aborts it. Ideally, it has to be efficient (this is measured by the number of transactions committed per time unit), while ensuring that as few transactions as possible are aborted. From a safety point of view (the one addressed in this paper), a STM system has to ensure that, whatever its fate (commit or abort), each transaction always operates on a consistent state.

STM systems have recently received a lot of attention. Among the proposed solutions, lock-based systems and clock-based systems have been particularly investigated. Their design is mainly efficiency-oriented, the properties they satisfy are not always clearly stated, and few of them are formally proved. This paper presents a lock-based STM system designed from simple basic principles. Its main features are the following: it (1) uses visible reads, (2) does not require the shared memory to manage several versions of each object, (3) uses neither timestamps, nor version numbers, (4) satisfies the opacity safety property, (5) aborts a transaction only when it conflicts with some other live transaction (progressiveness property), (6) never aborts a write only transaction, (7) employs only bounded control variables, (8) has no centralized contention point, and (9) is formally proved correct.

Keywords: Atomic operation, Commit/abort, Concurrency control, Consistent global state, Lock, Opacity, Progressiveness, Shared object, Software transactional memory, Transaction.

1 Introduction

Software transactional memory. Recent advances in technology, and more particularly in multicore processors, have given rise to a new momentum to practical and theoretical research in concurrency and synchronization. Software transactional memory (STM) constitutes one of the most visible domains impacted by these advances. Given that concurrent processes (or threads) that share data structures (base objects) have to synchronize, the transactional memory concept originates from the observation that traditional

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 226–245, 2008.

lock-based solutions have inherent drawbacks. On one side, if the set of data whose accesses are controlled by a single lock is too large (large grain), the parallelism can be drastically reduced, while, on another side, the solutions where a lock is associated with each datum (fine grain), are difficult to master, error-prone, and difficult to prove correct.

The software transactional memory (STM) approach has been proposed in [23]. Considering a set of sequential processes that accesses shared objects, it consists in decomposing each process into (a sequence of) transactions (plus possibly some parts of code not embedded in transactions). This is the job of the programmer. The job of the STM system is then to ensure that the transactions are executed as if each was an atomic operation (it would make little sense to move the complexity of concurrent programming from the fine management of locks to intricate decompositions into transactions). So, basically, the STM approach is a structuring approach. (STM borrows ideas from database transactions; there are nevertheless fundamental differences with database transactions that are examined below [10].)

Of course, as in database transactions, the fate of a transaction is to abort or commit. (According to its aim, it is then up to the issuing process to restart -or not- an aborted transaction.) The great challenge any STM system has to take up is consequently to be efficient (the more transactions are executed per time unit, the better), while ensuring that few transactions are aborted. This is a fundamental issue each STM system has to address. Moreover, in the case where a transaction is executed alone (no concurrency) or in the absence of conflicting transactions, it should not be aborted. Two transactions conflict if they access the same object and one of them modifies that object.

Consistency of a STM. In the past recent years, several STM concepts have been proposed, and numerous STM systems have been designed and analyzed. On the correctness side (safety), an important notion that has been introduced very recently is the concept of opacity. That concept, introduced and formalized by Guerraoui and Kapałka [12], is a consistency criterion suited to STM executions. Its aim is to render aborted transactions harmless.

The classical consistency criterion for database transactions is serializability [19] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that are committed. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. In a STM system, the code encapsulated in a transaction can be any piece of code and consequently a transaction has to always operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all the consistent states. If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction raises an exception that has to be handled by the process that invoked the corresponding transaction[1]. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to

[1] Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops.

always see a consistent state of the data it accesses. The important point is here that a transaction can (a priori) be any piece of code (involving shared data), it is not restricted to predefined patterns. This also motivates the design of STM protocols that reduce the number of aborts (even if this entails a slightly lower throughput for short transactions). Roughly speaking, opacity extends serializability to all the transactions (regardless of whether they are committed or aborted). Of course, a committed transaction is considered entirely. Differently, only an appropriately defined subset of an aborted transaction has to be considered.

Opacity (like serializability) states only what is a correct execution, it is a safety property. It does not state when a transaction has to commit, i.e., it is not a liveness property. Several types of liveness properties are investigated in [22].

Context of the work. Among the numerous STM systems that have been designed in the past years, only four of them are considered here, namely, JVSTM [8], TL2 [9], LSA-RT [21], and RSTM [16]. This choice is motivated by (1) the fact that (differently from a lot of other STM systems) they all satisfy the opacity property, and (2) additional properties that can be associated with STM systems.

Before introducing these properties, we first consider underlying mechanisms on which the design of STM systems is based.

- From an operational point of view, *locks* and (physical or logical) *clocks* constitute base synchronization mechanisms used in a lot of STM systems. Locks allow mutex-based solutions. Clocks allow to benefit from the progress of the (physical or logical) time in order to facilitate the validation test when the system has to decide the fate (commit or abort) of a transaction. As a clock can always increase, clock-based systems require appropriate management of the clock values.

An important design principle that differentiates STM systems is the way they implement base objects. More specifically we have the following.

- Two types of implementation of base objects can be distinguished, namely, the *single version* implementations, and the *multi-version* implementations. The aim of the latter is to allow the commit of more (mainly read only) transactions, but requires to pay a higher price from the shared memory occupation point of view.

An STM implementation can also be characterized by the fact it satisfies or not important additional properties. We consider here the progressiveness property.

- The *progressiveness* notion, introduced in [12], is a safety property from the commit/abort termination point of view: it defines an execution pattern that forces a transaction not to abort another one.

  As already indicated, two transactions conflict if they access the same base object and one of them updates it. The STM system satisfies the progressiveness property, if it "forcefully aborts T_1 only when there is a time t at which T_1 conflicts with another concurrent transaction (say T_2) that is not committed or aborted by time t" [12]. This means that, in all the other cases, T_1 cannot be aborted due to T_2. As an example, let us consider Figure 1 where two patterns are depicted. Both involve the same conflicting concurrent transactions T_1 that reads X, and T_2 that writes X (each transaction execution is encapsulated in a rectangle). On the left

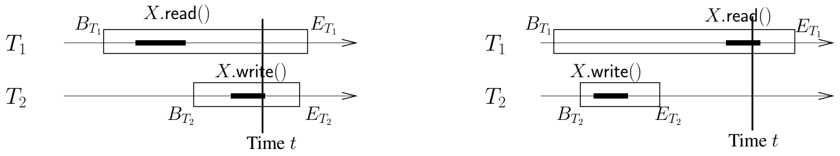


Fig. 1. The progressiveness property

side, T_2 has not yet terminated when T_1 reads X. In that case, an STM system that aborts T_1, due to its its conflict with T_2, does not violate the progressiveness property. Differently, when we consider the right side, T_2 has terminated when T_1 reads X. In that case, an STM system that guarantees the progressiveness property, cannot abort T_1 due to T_2.

Finally, a last criterion to compare STM systems lies in the way they cope with lower bound results related to the cost of read and write operations.

- Let k be the number of objects shared by a set of transactions. A theorem proved in [12] states the following. For any STM protocol that (1) ensures the opacity consistency criterion, (2) is based on single version objects, (3) implements invisible read operations, and (4) ensures the progressiveness property, each read/write operation issued by a transaction requires $\Omega(k)$ computation steps in the worst case. This theorem shows an inescapable cost associated with the implementation of invisible read operations as soon as we want single version objects and abort only due to conflict with a live transaction.

Considering the previous list of items (base mechanisms, number of versions, additional properties, lower bound), Table 1 indicates how each of the TL2, LSA-RT, JVSTM, and RSTM behaves. While traditional comparisons of STM systems are based on efficiency measurements (usually from benchmark-based simulations), this table provides a different view to compare STM systems. A read operation issued by a transaction is *invisible* if its implementation does not entail updates of the underlying control variables (kept in shared memory). Otherwise, the read is visible.

Content of the paper. The $\Omega(k)$ lower bound states an inherent cost for the STM systems that want to ensure invisible read operations and progressiveness while using a single version per object. When looking at Table 1, we see that, while both TL2 and JVSTM implement invisible read operations, each circumvents the $\Omega(k)$ lower bound in its own way. JVSTM uses several copies of each object and does not ensure the progressiveness property. TL2 does not ensure the progressiveness property either (it has even scenarios in which a transaction is directed to abort despite the fact that it has read consistent values.)

Progressiveness is a noteworthy safety property. As already indicated, it states circumstances where transactions must commit[2]. Considering consequently progressiveness as a first class property, this paper presents a new STM system that circumvents the $\Omega(k)$ lower bound and satisfies the progressiveness property. To that end it employs

[2] This can be particularly attractive when there are long-lived read-only transactions.

Table 1. Properties ensured by protocols (that satisfy the opacity property)

System	TL2 [9]	LSA-RT [20]	JVSTM [8]	RSTM [16]	This paper
Clock-free	no	no	no	yes	yes
Lock-based	yes	no	yes	no	yes
Single version	yes	no	no	yes	yes
Invisible read operations	yes	yes	yes	no	no
Progressiveness	no	yes	no	yes	yes
Circumvent the $\Omega(k)$ lower bound	yes	no	yes	no	yes

a single version per object and implements visible read operations. Moreover, differently from nearly all the STM systems proposed so far, whose designs have been driven mainly by implementation concerns and efficiency, the paper strives for a protocol with powerful properties that can be formally proved. Its formal proof gives us a deeper understanding on how the protocol works and why it works. Combined with existing protocols, it consequently enriches our understanding of STM systems.

Finally, let us notice that the proposed protocol exhibits an interesting property related to contention management. The shared control variables associated with each object X (it is their very existence that makes the read operations visible) can be used by an underlying contention manager [11,24]. If the contention manager is called when a transaction is about to commit, it can benefit from the content of these variables to decide whether to accept the commit or to abort the transaction in case this abort would entail more transactions to commit.

Roadmap. The paper is made up of 6 sections. Section 2 describes the computation model, and the safety property we are interested in (opacity, [12]). The proposed protocol is presented incrementally. A base protocol is first presented in Section 3. This STM protocol (also called STM system in the following) associates a lock and two atomic control variables (sets) with each object X. It also uses a global control variable (a set denoted OW) that is accessed by all the update transactions (when they try to commit). Section 4 presents a formal proof of the protocol. Then, Section 5 presents the final version of the protocol. The resulting STM system has the following noteworthy features. It (1) does not require the shared memory to manage several versions of each object, (2) does not use timestamps, (3) satisfies the opacity and progressiveness properties, (4) never aborts a write only transaction, (5) employs only bounded control variables, and (6) has no centralized contention point. The design of provable STM protocols is an important issue for researchers interested in the foundations of STM systems [3]. Finally, Section 6 concludes the paper.

2 Computation Model and Problem Specification

2.1 Computation Model

Base computation model: processes, base objects, locks and atomic registers The base system (on top of which one has to build a STM system) is made up of n asynchronous sequential processes (also called threads) denoted $p_1, \ldots, p_n$ (a process is also sometimes denoted p) that cooperate through base read/write atomic registers and locks. The shared objects are denoted with upper case letters (e.g., the base object X). A lock, with its classical mutex semantics, is associated with each base object X.

Each process p has a local memory (a memory that can be accessed only by p). Variables in local memory are denoted with lower case letters indexed by the process id (e.g., lrs_i is a local variable of p_i).

High (user) abstraction level: transactions From a structural point of view, at the user abstraction level, each process is made up of a sequence of transactions (plus some code managing these transactions). A transaction is a sequential piece of code (computation unit) that reads/writes base objects and does local computations. At the abstraction level at which the transactions are defined, a transaction sees only base objects, it sees neither the atomic registers nor the locks. (Atomic registers and locks are used by the STM system to correctly implement transactions on top of the base model).

A transaction can be a *read-only* transaction (it then only reads base objects), or an *update* transaction (it then modifies at least one base object). A *write-only* transaction is an update transaction that does not read base objects. A transaction is assumed to be executed entirely (commit) or not at all (abort). If a transaction is aborted, it is up to the invoking process to re-issue it (as a new transaction) or not. Each transaction has its own identifier, and the set of transactions can be infinite.

2.2 Problem Specification

Intuitively, the STM problem consists in designing (on top of the base computation model) protocols that ensure that, whatever the base objects they access, the transactions are correctly executed. The following property formulates precisely what "correctly executed" means in this paper.

Safety property. Given a run of a STM system, let $\mathcal{C}$ be the set of transactions that commit, and $\mathcal{A}$ the set of transactions that abort. Let us assume that any transaction T starts with an invocation event (B_T) and terminates with an end event (E_T).

Given $T \in \mathcal{A}$, let $T' = \rho(T)$ be the transaction built from T as follows (ρ stands for "reduced"). As T has been aborted, there is a read or a write on a base object that entailed that abortion. Let $prefix(T)$ be the prefix of T that includes all the read and write operations on the base objects accessed by T until (but excluding) the read or write that provoked the abort of T. $T' = \rho(T)$ is obtained from $prefix(T)$ by replacing its write operations on base objects and all the subsequent read operations on these objects, by corresponding write and read operations on a copy in local memory. The idea here is that only an appropriate prefix of an aborted transaction is considered: its write operations on base objects (and the subsequent read operations) are made fictitious in $T' = \rho(T)$. Finally, let $\mathcal{A}' = \{T' \mid T' = \rho(T) \ \wedge \ T \in \mathcal{A}\}$.

As announced in the Introduction, the safety property considered in this paper is *opacity* (introduced in [12] with a different formalism). It expresses the fact that a transaction never sees an inconsistent state of the base objects. With the previous notation, it can be stated as follows:

- $\mathsf{Opacity}$. The transactions in $\mathcal{C} \cup \mathcal{A}'$ are linearizable (i.e., can be totally ordered according to their real-time order [13]).

This means that the transactions in $\mathcal{C} \cup \mathcal{A}'$ appear as if they have been executed one after the other, each one being executed at a single point of the time line between its invocation event and its end event.

3 A Lock-Based STM System: Base Version

This section presents a base protocol that builds a STM system on top of the base system described in Section 2.1. Without ambiguity, the same identifier T is used to denote both a transaction itself and its unique name.

3.1 The STM System Interface

The STM system provides the transactions with three operations denoted $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction, and X a base object.

- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted.
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation never forces a transaction to immediately abort.
- If a transaction attains its last statement (as defined by the user) it executes the operation $\mathsf{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice, a transaction T that invokes $\mathsf{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\mathsf{read}_T()$.)

3.2 The STM System Variables

To implement the previous STM operations, the STM system uses a lock per base object X, and the following atomic control variables that are sets (all initialized to $\emptyset$).

- A read set RS_X is associated with each object X. This set contains the id of the transactions that have read X since its last update. A transaction adds its id to RS_X to indicate a possibility of conflict.
- A set OW, whose meaning is the following: $T \in OW$ means that the transaction T has read an object Y and, since this reading, Y has been updated (so, there is a conflict).
- A set FBD_X per base object X (FBD_X stand for For$B$iDden). $T \in FBD_X$ means that the transaction T has read an object Y that since then has been overwritten (hence $T \in OW$), and the overwriting of Y is such that any future read of X by T will be invalid (i.e., the value obtained by T from Y and any value it will obtain from X in the future cannot be mutually consistent): reading X from the shared memory is forbidden to the transactions in FBD_X.

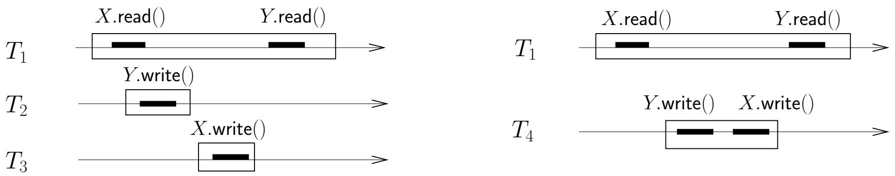


Fig. 2. Meaning of the set FBD_X

An example explaining the meaning of FBD_X is described in Figure 2. On the left side, the execution of three transactions are depicted (as before, each rectangle encapsulates a transaction execution). T_1 starts by reading X, executes local computation, and then reads Y. The execution of T_1 overlaps with two transactions, T_2 that is a simple write of Y, followed by T_3 that is a simple write of X. It is easy to see that the execution of these three transactions can be linearized: first T_2, then T_1 and finally T_3. In this execution, FBD_X does not include T_1.

In the execution on the right side, T_2 and T_3 are combined to form a single transaction T_4. It is easy to see that this concurrent execution of T_1 and T_4 cannot be linearized. Due to its access to X, the STM system (as we will see) will force T_4 to add T_1 to FBD_Y, entailing the abort of T_1 when T_1 will access Y (if T_1 would not access Y, it would not be aborted). Let us observe that the same thing occurs if, instead of T_4, we have (with the same timing) a transaction made up of X.write() followed by another transaction including Y.write().

The STM system also uses the following local variables (kept in the local memory of the process that invoked the corresponding transaction). lrs_T is a local set where T keeps the ids of all the objects it reads. Similarly, lws_T is a local set where T keeps the ids of all the objects it writes. Finally, $read\_only_T$ is a boolean variable initialized to $true$.

The previous shared sets can be efficiently implemented using Bloom filters (e.g., [2,7,17]). In a very interesting way, the small probability of false positive on membership queries does not make the protocol incorrect (it can only affect its efficiency by entailing non-necessary aborts).

Let us recall that a process is sequential and consequently executes transactions one after the other. As local control variables are associated with a transaction, the corresponding process has to reset them to their initial values between two transactions. Similarly, if a transaction creates a local copy of an object, that copy is destroyed when the transaction terminates (a given copy of an object is meaningful for one transaction only).

3.3 The Algorithms of the STM System

The three operations that constitute the STM system X.read$_T()$, X.write$_T(v)$, and try_to_commit$_T()$, are described in Figure 3.

The operation X.read$_T()$. The algorithm implementing this operation is pretty simple. If there is a local copy of X, its value is returned (lines 01 and 07). Otherwise, space for X is allocated in the local memory (line 02), X is added to the set of objects read by T (line 03), T is added to the read set RS_X of X, and the current value of X is read from the shared memory and saved in the local memory (line 04).

```
operation $X$.read$_T()$:
(01) if (there is no local copy of $X$) then
(02)    allocate local space for a copy;
(03)    $lrs_T \leftarrow lrs_T \cup \{X\}$;
(04)    lock $X$; local copy of $X \leftarrow X$; $RS_X \leftarrow RS_X \cup \{T\}$; unlock $X$;
(05)    if $(T \in FBD_X)$ then return$(abort)$ end if
(06) end if;
(07) return(value of the local copy of $X$)
==========================================================
operation $X$.write$_T(v)$:
(08) $read\_only_T \leftarrow false$;
(09) if (there is no local copy of $X$) then allocate local space for a copy end if;
(10) local copy of $X \leftarrow v$;
(11) $lws_T \leftarrow lws_T \cup \{X\}$
==========================================================
operation try_to_commit$_T()$:
(12) if $(read\_only_T)$
(13)    then return$(commit)$
(14)    else lock all the objects in $lrs_T \cup lws_T$;
(15)         if $(T \in OW)$ then release all the locks; return$(abort)$ end if;
(16)         for each $X \in lws_T$ do $X \leftarrow$ local copy of $X$ end for;
(17)         $OW \leftarrow OW \cup \big(\cup_{X \in lws_T} RS_X\big)$;
(18)         for each $X \in lws_T$ do $FBD_X \leftarrow OW$; $RS_X \leftarrow \emptyset$ end for;
(19)         release all the locks;
(20)         return$(commit)$
(21) end if
```

Fig. 3. A lock-based STM system

Due to asynchrony, it is possible that the value read by T is overwritten before T uses it. The predicate $T \in FBD_X$ is used to capture this type of read/write conflict. If this predicate is true, T is aborted (line 05). Otherwise, the value obtained from X is returned (line 07). It is easy to see that any object X is read from the shared memory at most once by a transaction.

The operation X.write$_T()$. The text of the algorithm implementing X.write$_T()$ is even simpler than the text of X.read$_T()$. The transaction first sets a flag to record that it is not a read-only transaction (line 08). If there is no local copy of X, corresponding space is allocated in the local memory (line 09); let us remark that this does not entail a reading of X from the shared memory. Finally, T updates the local copy of X (line 10), and records that it has locally written the copy of X (line 11).

It is important to notice that an invocation of X.write$_T()$ is purely local: it involves no access to the shared memory, and cannot entail an immediate abort of the corresponding transaction.

The operation try_to_commit$_T()$. This operation works a follows. If the invoking transaction is a read-only transaction, it is committed (lines 12-13). So, a read-only transaction can abort only during the invocation of a X.read$_T()$ operation (line 05 of that operation).

If the transaction T is an update transaction, $\mathsf{try\_to\_commit}_T()$ first locks all the objects accessed by T (line 14). (In order to prevent deadlocks, it is assumed that these objects are locked according to a predefined total order, e.g., their identity order.) Then, T checks if it belongs to the set OW. If this is the case, there is a read-write conflict: T has read an object that since then has been overwritten. T consequently aborts (after having released all the locks, line 15). If the predicate $T \in OW$ is false, T will necessarily commit. But, before committing (at line 20), T has to update the control variables to indicate possible conflicts due to the objects it has written, the ids of which have been kept by T in the local set lws_T during its execution.

So, after it has updated the shared memory with the new value of each object $X \in lws_T$ (line 16), T computes the union of their read sets; this union contains all the transactions that will have a write/read conflict with T when they will read an object $X \in lws_T$. This union set is consequently added to OW (line 17), and the set FBD_X of each object $X \in lws_T$ is updated to OW (line 18). (It is important to notice that each set FBD_X is updated to OW in order not to miss the transitive conflict dependencies that have been possibly created by other transactions). Moreover, as now the past read/write conflicts are memorized in FBD_X (line 18), the transaction T resets RS_X to $\emptyset$ just after it has set FBD_X to OW. Finally, before committing, T releases all its locks (line 19).

On locking. As in TL2 [9], it is possible to adopt the following systematic abort strategy. When a transaction T tries to lock an object that is currently locked, it immediately aborts (after releasing the locks it has, if any).

3.4 On the Management of the Sets RS_X, FBD_X and OW

Let us recall that these sets are kept in atomic variables.

Management of RS_X and FBD_X. The set RS_X is written only at line 04 ($\mathsf{read}_T()$ operation), and reset to $\emptyset$ at line 18 ($\mathsf{try\_to\_commit}_T()$ operation), and (due to the lock associated with X) no two updates of RS_X can be concurrent; so, no update of RS_X is missed. Its only read (line 16) is protected by the same lock. So, there is no concurrency problem for RS_X.

The set FBD_X is read at line 06 ($\mathsf{read}_T()$ operation), and its only write (line 18, $\mathsf{try\_to\_commit}_T()$ operation) is protected by a lock. As it is an atomic variable, there is no concurrency problem for FBD_X.

Management of the set OW. This set is read and written only by $\mathsf{try\_to\_commit}_T()$ which reads it at lines 15 and 17, and writes it at line 17 (its read at line 18 can benefit from a local copy saved at line 17).

Concurrent invocations of $\mathsf{try\_to\_commit}_T()$ can come from transactions accessing distinct sets of objects. When this occurs, the set OW is not protected by the locks associated with the objects and can consequently be concurrently accessed. As OW is kept in an atomic variable there is no concurrency problem for the reads. Differently, writes of OW (line 17) can be missed. Actually, when we look at the update of the atomic set variable OW, namely $OW \leftarrow OW \cup \big(\cup_{X \in lws_T} RS_X \big)$ (line 17), we can observe that this update is nothing else than a $Fetch\&Add()$ statement that has to atomically add $\cup_{X \in lws_T} RS_X$ to OW. If such an operation on a set variable is not provided by the hardware, there are several ways to implement it. One consists in using a lock to execute this operation is mutual exclusion. Another consists in using specialized hardware operations such as $Compare\&swap()$ (manipulating a pointer on the set OW, or LL/SC (load-linked/store-conditional) [15,18]. Yet, another possible implementation consists in considering the set OW as a shared array with one entry per process, p_i being the only process that can write $OW[i]$. Moreover, for efficiency, the current value of $OW[i]$ can be saved in a local variable ow_i. A write by p_i in $OW[i]$ then becomes $ow_i \leftarrow ow_i \cup_{X \in lws_T} RS_X$ followed by $OW[i] \leftarrow ow_i$; while the atomic read of the set OW is implemented by a snapshot operation on the array $OW[1..n]$ [1] (there are efficient implementations of the snapshot operation, e.g., [4,5]).

Differently from the pair of sets RS_X and FBD_X, associated with each object X, the set OW constitutes a global contention point. This contention point can be suppressed by replacing OW by independent boolean variables (see Section 5). We have adopted here an incremental presentation, to make the final protocol easier to understand.

3.5 Early Abort and Contention Manager

When the predicate $T \in OW$ is satisfied, the transaction T has read an object that since then has been overwritten. This fact is not sufficient to abort T if it is a read-only transaction. Differently, if T is an update transaction, it cannot be linearized; consequently, it will be aborted when executing line 15 of $\mathsf{try\_to\_commit}_T()$. It is possible to abort such an update transaction T earlier than during the execution of $\mathsf{try\_to\_commit}_T()$. This can be simply done by adding the statement "**if** $T \in OW$ **then** $\mathsf{return}(abort)$ **end if**" just before the first line of the operation $\mathsf{write}_T()$. Similarly, the statement "**if** $T \in FBD_X$ **then** $\mathsf{return}(abort)$ **end if**" can be inserted between the first and the second line of the operation $\mathsf{read}_T()$.

Interestingly, it is important to notice that the sets RS_X, FBD_X, and OW can be used by an underlying contention manager [11,24] to abort transactions according to predefined rules (namely, there are configurations where aborting a single transaction can prevent the abort of other transactions).

4 Proof of the Base Protocol

4.1 Base Formalism and Definitions

Events and history at the shared memory level. An event is associated with the execution of each operation on the shared memory (base object, lock, set variable). We use the following notation.

- Let B_T denote the event associated with the beginning of the transaction T, and E_T the event associated with its termination. E_T can be of two types, namely A_T and C_T, where A_T is the event "abort of T" (line 05 or 15), and C_T is the event "commit of T" (line 20).
- Let $r_T(X)v$ denote the event associated with the read of X from the shared memory issued by the transaction T; v denotes the value returned by the read. Given an object X, there is a most one event $r_T(X)v$ per transaction T. If any, this read occurs at line 04 (operation $X.\mathsf{read}_T()$).

- Let $w_T(X)v$ denote the event associated with the write of the value v in X. Given an object X, there is at most one event $w_T(X)v$ per transaction T. If any, it corresponds to a write issued at line 16 in the $\mathsf{try\_to\_commit}_T()$ operation. If the value v is irrelevant $w_T(X)v$ is abbreviated $w_T(X)$.
  Without loss of generality we assume that no two writes on the same object X write the same value.
  We also assume that all the objects are initially written by a fictitious transaction.
- Let $AL_T(X, op)$ denote the event associated with the acquisition of the lock on the object X issued by the transaction T during an invocation of op where op is $X.\mathsf{read}_T()$ or $\mathsf{try\_to\_commit}_T()$.
  Similarly, let $RL_T(X, op)$ denote the event associated with the release of the lock on the object X issued by the transaction T during an invocation of op.

Given an execution, let H be the set of all the events generated by the shared memory accesses issued by the STM system described in Figure 3. As these shared memory accesses are atomic, the previous events are totally ordered. Consequently, at the shared memory level, an execution can be represented by the pair $\widehat{H} = (H, <_H)$ where $<_H$ denotes the total ordering on its events. $\widehat{H}$ is called a *shared memory history*.

As $<_H$ is a total order, it is possible to consider each event in H as a date of the time line. This "date" view of a sequential history on events will be used in the proof.

History at the transaction level. Given an execution, let TR be the set of transactions issued during that execution. Let $\rightarrow_{TR}$ be the order relation defined on the transactions of TR as follows: $T1 \rightarrow_{TR} T2$ if $E_{T1} <_H B_{T2}$ ($T1$ has terminated before $T2$ starts). If $T1 \not\rightarrow_{TR} T2 \wedge T2 \not\rightarrow_{TR} T1$, we say that $T1$ and $T2$ are concurrent (their executions overlap in time). At the transaction level, that execution is defined by the partial order $\widehat{TR} = (TR, \rightarrow_{TR})$, that is called a *transaction level history*.

The *read-from* relation between transactions, denoted $\rightarrow_{rf}$, is defined as follows: $T1 \overset{X}{\rightarrow}_{rf} T2$ if $T2$ reads the value that $T1$ wrote in the object X.

A transaction history $\widehat{ST} = (ST, \rightarrow_{ST})$ is *sequential* if no two of its transactions are concurrent. Hence, in a sequential history, $T1 \not\rightarrow_{ST} T2 \Leftrightarrow T2 \rightarrow_{ST} T1$, thus $\rightarrow_{ST}$ is a total order. A sequential transaction history is *legal* if each of its read operations returns the value of the last write on the same object (because the history is sequential and transactions are executed sequentially, no two operations can overlap).

A sequential transaction history $\widehat{ST}$ is *equivalent* to a transaction history $\widehat{TR}$ if (1) $ST = TR$ (i.e., they are made of the same transactions -same invocations and same replies- in $\widehat{ST}$ and in $\widehat{TR}$), and (2) the total order $\rightarrow_{ST}$ respects the partial order $\rightarrow_{TR}$ (i.e., $\rightarrow_{TR} \subseteq \rightarrow_{ST}$).

A transaction history $\widehat{AA}$ is *linearizable* if there exists a history $\widehat{SA}$ that is sequential, legal and equivalent to $\widehat{AA}$.

Let $\widehat{\rho(TR)}$ denote the transaction history obtained from the history $\widehat{TR}$ as described in Section 2.2. This means that $\widehat{\rho(TR)}$ includes all the transactions of $\widehat{TR}$ that commit, and contains $\rho(T)$ for each transaction $T \in \widehat{TR}$ that aborts. As defined in Section 2.2, a transaction history $\widehat{TR}$ is *opaque* if there exists a transaction history $\widehat{ST}$ that is sequential, legal and equivalent to $\widehat{\rho(TR)}$.

4.2 Principle of the Proof of the Opacity Property

According to the algorithms implementing the operations $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T(v)$ described in Figure 3, we ignore all the read operations on an object that follow another operation on the same object within the same transaction, and all the write operations that follow another write operation on the same object within the same transaction (these are operations local to the memory of the process that executes them). Building $\rho(TR)$ from TR is then a straightforward process.

To prove that the protocol described in Figure 3 satisfies the opacity consistency criterion, we need to prove that, for any transaction history $\widehat{TR}$ produced by this protocol, there is a sequential legal history $\widehat{ST}$ equivalent to $\widehat{\rho(TR)}$. This amounts to prove the following properties (where $\widehat{H}$ is the shared memory level history generated by the transaction history $\widehat{TR}$):

1. $\rightarrow_{ST}$ is a total order,
2. $\forall T \in TR : \big(T \text{ commits} \Rightarrow T \in ST\big) \wedge \big(T \text{ aborts} \Rightarrow \rho(T) \in ST\big)$,
3. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$,
4. $T1 \xrightarrow{X}_{rf} T2 \Rightarrow \nexists T3$ such that $\big(T1 \rightarrow_{ST} T3 \rightarrow_{ST} T2\big) \wedge \big(w_{T3}(X) \in H\big)$,
5. $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \rightarrow_{ST} T2$.

4.3 Definition of the Linearization Points

ST is produced by ordering the transactions according to their linearization points. The linearization point of the transaction T is denoted ℓ_T. The linearization points of the transactions are defined as follows:

- If a transaction T aborts, ℓ_T is the time just before T is added to the set OW (line 17 of the $\mathsf{try\_to\_commit_T}()$ operation that entails its abort).
- If a read only transaction T commits, ℓ_T is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 05 of the $X.\mathsf{read}()$ operation) and (2) the time just before it is added to OW (if it ever is). (An example is depicted in Figure 4.)
- If an update transaction T commits, ℓ_T is placed just after the execution of line 17 by T (update of OW).

The total order $<_H$ (defined on the events generated by $\widehat{TR}$) can be extended with these linearization points. Transactions whose linearization points happen at the same time (for example, in multi-core systems) are ordered arbitrarily. An example is given in Figure 4.

4.4 Safety: Proof of the Opacity Property

Let $\widehat{TR} = (TR, \rightarrow_{TR})$ be a transaction history. Let $\widehat{ST} = (\rho(TR), \rightarrow_{ST})$ a history whose transactions are the transactions in $\rho(TR)$, and such that $\rightarrow_{ST}$ is defined according to linearization points of each transaction in $\rho(TR)$. If two transactions in $\rho(TR)$ have the same linearization point, they are ordered arbitrarily. Finally, let us observe that

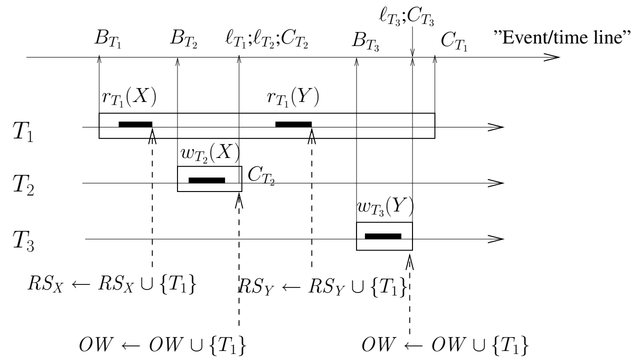

$\ell_{T_3};C_{T_3}$
B_{T_1}
B_{T_2}
$\ell_{T_1};\ell_{T_2};C_{T_2}$
B_{T_3}
C_{T_1}
"Event/time line"
$r_{T_1}(X)$
$r_{T_1}(Y)$
T_1
$w_{T_2}(X)$
T_2
C_{T_2}
$w_{T_3}(Y)$
T_3
$RS_X \leftarrow RS_X \cup \{T_1\}$
$RS_Y \leftarrow RS_Y \cup \{T_1\}$
$OW \leftarrow OW \cup \{T_1\}$
$OW \leftarrow OW \cup \{T_1\}$

Fig. 4. An example of linearization points

the linearization points can be trivially added to the sequential history $\widehat{H} = (H, \rightarrow_H)$ defined on the events generated by the transaction history $\widehat{TR}$. So, we consider in the following that the set H includes the linearization points of the transactions.

Lemma 1. *$\rightarrow_{ST}$ is a total order.*

Proof. Trivial from the ordering of the linearization points. □

Lemma 2. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$.

Proof. This lemma follows from the fact that, given any transaction T, its linearization point is placed within its lifetime. Therefore, if $T1 \rightarrow_{\rho(TR)} T2$ ($T1$ ends before $T2$ begins), then $T1 \rightarrow_{ST} T2$. □

Let $ow(T, t)$ be the predicate "at time t, T belongs to OW".

Lemma 3. *$ow(T, t) \Rightarrow \ell_T <_H t$.*

Proof. We show that the linearization point of a transaction T cannot be after the time at which the transaction's id is added to OW. There are three cases.

- By construction, if T aborts, its linearization ℓ_T is the time just before its id is added to OW, which proves the lemma.
- If T is read-only and commits, again by construction, its linearization ℓ_T point is placed at the latest just before the time at which its id is added to OW (if it ever is), which again proves the lemma.
- If T writes and commits, its linearization point ℓ_T is placed during try_to_commit $()$, while T holds the locks of every object that it has read. If T was in OW before it acquired all the locks, it would not commit (due to line 15). Let us notice that T can be added to OW only by an update transaction holding a lock on a base object previously read by T. As T releases the locks just before committing (line 19), it follows that ℓ_T occurs before the time at which its id is added to OW (if it ever is), which proves the last case of the lemma. □

Let $rs_X(T, t)$ be the predicate "at time t, T belongs to RS_X or OW".

Lemma 4. $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \nexists T'_W$ *such that* $\big(T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R\big) \wedge \big(w_{T'_W}(X) \in H\big)$.

Proof. By contradiction, let us assume that there are transactions T_W, T'_W and T_R and an object X such that:

- $T_W \xrightarrow{X}_{rf} T_R$,
- $w_{T'_W}(X)v' \in H$,
- $T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R$.

As both T_W and T'_W write X in shared memory, they have necessarily committed (a write in shared memory occurs only at line 16 during the execution of $\mathsf{try\_to\_commit}()$, abbreviated ttc in the following). Moreover, their linearization points ℓ_{T_W} and $\ell_{T'_W}$ occur while they hold the lock on X (before committing), from which we have the following implications:

$$T_W \rightarrow_{ST} T'_W \Leftrightarrow \ell_{T_W} <_H \ell_{T'_W},$$
$$\ell_{T_W} <_H \ell_{T'_W} \Rightarrow RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}),$$
$$RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}) \Rightarrow w_{T_W}(X)v <_H w_{T'_W}(X)v',$$
$$\big(T_W \xrightarrow{X}_{rf} T_R\big) \wedge \big(w_{T_W}(X)v <_H w_{T'_W}(X)v'\big) \Rightarrow w_{T_W}(X)v <_H r_{T_R}(X)v <_H w_{T'_W}(X)v'.$$

A transaction T that reads an object X always adds its id to RS_X before releasing the lock on X. Therefore, the predicate $rs_X(T, RL_T(X, X.\mathsf{read}_T()))$ is true (RS_X is set to $\emptyset$ only after being added to the set OW). Using this observation, we have the following:

$$r_{T_R}(X)v <_H w_{T'_W}(X)v' \wedge rs_X(T_R, RL_{T_R}(X, X.\mathsf{read}_{T_R}())) \Rightarrow rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})),$$
$$\text{(Due to Line 17)} \quad rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})) \wedge \big(w_{T'_W}(X)v' \in H\big) \Rightarrow ow(T_R, \ell_{T'_W}),$$
$$\text{(Due to Lemma 3)} \quad ow(T_R, \ell_{T'_W}) \Rightarrow \ell_{T_R} <_H \ell_{T'_W},$$
$$\text{(and finally)} \quad \ell_{T_R} <_H \ell_{T'_W} \Leftrightarrow T_R \rightarrow_{ST} T'_W,$$

which proves that, contrarily to the initial assumption, T'_W cannot precede T_R in the sequential transaction history $\widehat{ST}$. □

Let $\mathit{fbd}_X(T, t)$ be the predicate "at time t, T belongs to FBD_X".

Lemma 5. $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.

Proof. The proof is made up of two parts. First it is shown that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W})$, and then it is shown that $\neg ow(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.

Proof of $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W})$. Let us assume by contradiction that the predicate $ow(T_R, \ell_{T_W})$ is true. Due to line 18 we have

$$ow(T_R, \ell_{T_W}) \Rightarrow fbd_X(T_R, RL_{T_W}(X, \mathsf{ttc})).$$

If the read of X from shared memory by T_R is before the write by T_W, we cannot have $T_W \xrightarrow{X}_{rf} T_R$. So, in the following we consider that the read of X from shared memory by T_R is after its write by T_W. We have then $RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}())$, and consequently

$$fbd_X(T_R, RL_{T_W}(X, \mathsf{ttc})) \Rightarrow fbd_X(T_R, AL_{T_R}(X, X.\mathsf{read}_{T_R}())).$$

As $T_R \in FBD_X$ when it locks X, it follows that T_R aborts at line 05 and consequently we cannot have $T_W \xrightarrow{X}_{rf} T_R$. Summarizing the previous reasoning we have $(ow(T_R, \ell_{T_W}) \Rightarrow \neg(T_W \xrightarrow{X}_{rf} T_R))$, and taking the contrapositive we finally obtain

$$T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W}).$$

Proof of $\neg ow(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$. As defined in Section 4.3, the linearization point ℓ_{T_R} depends on the fact that T_R commits or aborts, and is a read only or an update transaction. The proof considers the three possible cases.

- If T_R is an update transaction that commits, its linearization point ℓ_{T_R} (that is defined as line 17 when it updates the set OW) occurs after its invocation of $\mathsf{try\_to\_commit}()$. Due to this observation, the fact that T_W releases its locks after its linearization point, and $T_W \xrightarrow{X}_{rf} T_R$, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.
- If T_R is a (read only or update) transaction that aborts, its linearization point ℓ_{T_R} is the time at which T_R is added to OW. Because $T_W \xrightarrow{X}_{rf} T_R$ we have $\neg ow(T_R, \ell_{T_W})$. Moreover, due to $\neg ow(T_R, \ell_{T_W})$ and the fact that T_R aborts, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$. It follows that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.
- If T_R is a read only transaction that commits, its linearization point ℓ_{T_R} is placed either at the time at which it is added to OW (then the case is the same as a transaction that aborts, see before), or at the time of the test during its last read operation (line 05). In the latter case, we have $w_{T_W}(X)v <_H \ell_{T_W} <_H RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}()) <_H r_{T_R}(X)v <_H \ell_{T_R}$, from which we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.

Hence, in all cases, we have $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$. □

Theorem 1. *Every transaction history $\widehat{TR}$ produced by the protocol described in Figure 3 satisfies the opacity consistency criterion.*

Proof. The proof follows from the construction of the set $\rho(TR)$ (Section 4.1), the definition of the linearization points (Section 4.3), and the Lemmas 1, 2, 4 and 5. □

4.5 Safety: Proof of the Progressiveness Property

Theorem 2. *If a transaction $T1$ aborts, then there is a time t at which $T1$ conflicts with another transaction $T2$ that is neither committed nor aborted by time t.*

Proof. The abort of a transaction $T1$ is due to a $X.\mathsf{read}()$ operation (test at line 05) or a $\mathsf{try\_to\_commit}()$ operation (test at line 15). In both cases, $T1$ has read an object Y and has been added first to the set OW (line 17), and then to the set FBD_X (line 18), during the $\mathsf{try\_to\_commit}()$ operation of the transaction $T2$ that wrote Y. As $T1$ belongs to OW, we conclude that there is a read/write conflict on Y between $T1$ and $T2$ (where $T1$ is the reader and $T2$ is the writer). Because $T2$ adds $T1$ to OW during its $\mathsf{try\_to\_commit}()$ operation, the conflict happens during the lifetime of $T2$, from which it follows that $T1$ can be aborted only due to a conflict with a transaction $T2$ that is still alive at the time of the conflict (as depicted in Figure 1). □

4.6 Liveness: Termination of Transactions

It is easy to see that each transaction terminates. Concerning the fact that a transaction terminates successfully (commit) or not (abort), we have the following properties.

- If a transaction $T1$ entails the abort of a transaction $T2$, then $T1$ necessarily commits.
  This is because the abort of $T2$ occurs at line 05 (test $T2 \in FBD_X$) or at line 15 (test $T2 \in OW$). In both cases the addition of $T2$ to the set FBD_X or OW is done by a transaction $T1$ while it executes the lines 17 or 18, i.e., just before $T1$ commits at line 20.
- If none of the values it has read is overwritten, an update transaction cannot abort. Moreover, a write-only transaction never aborts.
- A transaction that reads an object X, is not necessarily aborted, despite the fact that it has previously read an object Y that is now overwritten, as long as the values of X and Y it has obtained belong to the same consistent state.

5 A Lock-Based STM System: Final Version

5.1 The Improvements

From transaction ids to process ids. While this is not necessary from a correctness point of view, it is desirable that the identities of transactions that have terminated (committed or aborted) be suppressed from the sets RS_X, FBD_X, and OW. Moreover, the fact that the domain of these identities is unbounded can become a real drawback. As there a is a fixed number of processes, and each process issues one transaction at a time, a solution consists in using the id of the issuing process as the id of the corresponding transaction. From the point of view of transaction ids, this means that they are now recyclable. This recycling can be obtained with an appropriate update of the relevant control variables each time a transaction terminates.

Eliminating the contention point OW. The set OW can actually be replaced by a set of atomic boolean variables, one per process p_i. The boolean associated with p_i, denoted OW_i, has the following meaning: $OW_i = \mathit{true}$ means that the current transaction issued by p_i has read an object whose value is no longer up to date.

The resulting improvement. It follows from the previous improvements that all the shared control variables do have a bounded domain. They are either boolean variables, or set variables that contain at most n process ids. Let us remark that there are very efficient management algorithms for such sets.

5.2 The Final (Improved) STM System

The three operations X.read$_i()$, X.write$_i(v)$ and try_to_commit$_i()$ use the id i of the invoking process p_i instead of the transaction id. They also use an internal operation denoted init$_i()$ the aim of which is to reset control variables and suppress the id i of the invoking process from the sets it belongs to. The local variables lws_i, lrs_i and $read\_only_i$ replace their counterparts used in the base algorithms. Once this replacement has been done, the algorithms for X.read$_i()$ and X.write$_i(v)$ are verbatim the same as before (Figure 3). Consequently, they are not reproduced in Figure 5.

The code of the try_to_commit$_i()$ operation appears in Figure 5. It is the same as in Figure 3 with three modifications. To emphasize the incremental presentation, the lines that are not modified (but for the use of the process id) have the same number in Figure 3 and Figure 5. Differently, the lines that are modified keep the same number but are postfixed with a letter.

The first modification concerns line 15: the test $T \in OW$ is replaced by the test of the boolean OW_i (line 15.a). The second modification concerns the update of the set OW (line 17 in Figure 3). This update now consists in setting to $true$ the boolean OW_j associated with each process p_j that belongs to the read sets RS_X of the objects X written by p_i. This modified update appears at line 17.a of Figure 5. The last modification concerns the sets FBD_X associated with the objects written by p_i (line 18 in Figure 3). These updates are now based on the booleans OW_j instead of the set OW; they appear at line 18.b in Figure 5. Finally, the lines A1-A3 describe the internal init$_i()$ operation.

Remark. Due to the combined effect of asynchrony, the use of the boolean values OW_j, and the fact that a transaction uses the id of the issuing process, it is possible that a

```
operation try_to_commit_i():
(12)   if ($read\_only_i$)
(13)      then init_i(); return($commit$)
(14)      else lock all the objects in $lrs_i \cup lws_i$;
(15.a)         if $OW_i$ then release all the locks; init_i(); return($abort$) end if;
(16)           for each $X \in lws_i$ do $X \leftarrow$ local copy of $X$ end for;
(17.a)         for each $j \in \left(\cup_{X \in lws_i} RS_X\right)$ do $OW_j \leftarrow true$ end for;
(18.a)         for each $X \in lws_i$ do
(18.b)             for each $j$ such that $OW_j$ do $FBD_X \leftarrow FBD_X \cup \{j\}$ end for;
(18.c)             $RS_X \leftarrow \emptyset$;
(18.d)         end for;
(19)           release all the locks;
(20.a)         init_i(); return($commit$)
(21)   end if
=====================================================
operation init_i():
(A1) for each $X \in lrs_i$ do $RS_X \leftarrow RS_X \setminus \{i\}$ end for;
(A2) $OW_i \leftarrow false$; $lws_i \leftarrow \emptyset$; $lrs_i \leftarrow \emptyset$; $read\_only_i \leftarrow true$;
(A3) for each $X$ such that $(i \in FBD_X)$ do $FBD_X \leftarrow FBD_X \setminus \{i\}$ end for
```

Fig. 5. The improved STM system (bounded variables and no centralized contention point)

transaction be aborted while it would not be aborted when using the base algorithm described in Figure 3. This appears rarely. In these very rare cases, progressiveness is not guaranteed, while the opacity property is always ensured.

6 Conclusion

The focus of the paper was the design of a provably correct STM protocol that satisfies the opacity and progressiveness safety properties. As shown in [12], a price has to be paid to obtain both these properties (see Table 1). The price paid by the proposed protocol is read visibility (a read operation issued by a transaction has to write control information in the shared memory). The proposed protocol enjoys several additional properties: it uses neither clocks nor timestamps, manages a single version of each base object, never aborts a write-only transaction, uses only bounded control variables, and has no centralized control.

Acknowledgments

We would like to thank Pascal Felber for providing us with good advices and nice food on STM systems. We would also thank the referees for their constructive comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic Snapshots of Shared Memory. Journal of the ACM 40(4), 873–890 (1993)
2. Almeida, P.S.D., Baquero, C., Preguiça, N., Hutchinson, D.: Scalable Bloom Filters. Information Processing Letters 101(6), 255–261 (2007)
3. Attiya, H.: Needed: Foundations for Transactional Memory. ACM Sigact News, Distributed Computing Column 39(1), 59–61 (2008)
4. Attiya, H., Guerraoui, R., Ruppert, E.: Partial Snapshot Objects. In: Proc. 20th ACM Symposium on Parallel Algorithms and Architectures (SPAA 2008). ACP Press (2008)
5. Attiya, H., Rachman, O.: Atomic Snapshots in O($n \log n$) Operations. SIAM Journal on Computing 27(2), 319–340 (1998)
6. Avni, H., Shavit, N.: Maintaining Consistent Transactional States without a Global Clock. In: Shvartsman, A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058, pp. 121–140. Springer, Heidelberg (2008)
7. Bloom, B.H.: Space/Time Tradeoffs in Hash-coding with Allowable Errors. Communications of the ACM 13(7), 422–426 (1970)
8. Cachopo, J., Rito-Silva, A.: Versioned Boxes as the Basis for Transactional Memory. Science of Computer Programming 63(2), 172–175 (2006)
9. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
10. Felber, P., Fetzer, C., Guerraoui, R., Harris, T.: Transactions are coming Back, but Are They The Same? ACM Sigact News, Distributed Computing Column 39(1), 48–58 (2008)
11. Guerraoui, R., Herlihy, M.P., Pochon, S.: Towards a Theory of Transactional Contention Managers. In: Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC 2005), pp. 258–264. ACM Press, New York (2005)

12. Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2008), pp. 175–184. ACM Press (2008)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
14. Imbs, D., Raynal, M.: A Lock-based Protocol for Software Transactional Memory. Tech Report, #1893, IRISA, Université de Rennes 1, France (May 2008)
15. Jayanti, P., Petrovic, S.: Efficient and Practical Constructions of LL/SC Variables. In: Proc. 22nd ACM Symp. on Principles of Dist. Computing, pp. 285–294. ACM Press, New York (2003)
16. Spear, M.F., Marathe, V.J., Scherer III, W.N., Scott, M.L.: Conflict Detection and Validation Strategies for Software Transactional Memory. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 179–193. Springer, Heidelberg (2006)
17. Mitzenmatcher, M.: Compressed Bloom Filters. IEEE Transaction on Networks 10(5), 604–612 (2002)
18. Moir, M.: Practical Implementation of Non-Blocking Synchronization Primitives. In: Proc. 16th ACM Symposium on Principles of Distributed Computing, pp. 219–228. ACM Press, New York (1997)
19. Papadimitriou, C.H.: The Serializability of Concurrent Updates. Journal of the ACM 26(4), 631–653 (1979)
20. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
21. Riegel, T., Fetzer, C., Felber, P.: Time-based Transactional Memory with Scalable Time Bases. In: Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2007), pp. 221–228. ACM Press, New York (2007)
22. Scott, L.M.: Sequential Specification of Transactional Memory Semantics. In: Proc. First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Trans. Computing (TRANSACT 2006). ACM Press, New York (2006)
23. Shavit, N., Touitou, D.: Software Transactional Memory. Distributed Computing 10(2), 99–116 (1997)
24. Scherer III, W.N., Scott, M.L.: Advanced Contention Management in Dynamic Software Transactional Memory. In: Proc. 24th ACM Symp. on Principles of Dist. Computing (PODC 2005), pp. 240–248. ACM Press, New York (2005)

The 0–1-Exclusion Families of Tasks

Eli Gafni[*]

Computer Science Department
UCLA, Los Angles, CA 90095
eli@cs.ucla.edu

Abstract. Interesting tasks are scarce. Yet, they are essential as an investigation material, if we are to understand the structure of the tasks world. We propose a new collection of families of tasks called 0-1 Exclusion tasks, and show that families in this collection are interesting.

A 0-1 Exclusion task on n processors is specified by a sequence of $n-1$ bits $b(1), b(2), ..., b(n-1)$. For participating set of size $k,\ 0 < k < n$, each processor is to output 0 or 1 but they should not all output $b(k)$. When the participating set is of size n, then they should all output neither all 0's nor all 1's. A family of tasks, one for each n, is created by considering an infinite sequence of bits $b(k),\ k = 2, 3, ...$, such that the sequence that specifies instant n, is a prefix of the sequence that specifies the $n+1$'st instance.

Only one family in the collection, the one specified by $b(1) = b(2) = \ldots = 1$, was implicitly considered in the past and shown to be equivalent to Set-Consensus. In this initial investigation of the whole collection we show that not all of its members are created equal. We take the family specified by $b(1) = 1, b(2) = b(3) = ... = 0$, and show that it is read-write unsolvable for all n, but is strictly weaker than Set-Consensus for n odd.

We show some general results about the whole collection. It is sandwiched between Set-Consensus from above and Weak-Symmetry-Breaking from below. Any Black-Box of n ports that solves a 0-1 Exclusion task, can be used to solve that task for n processors with ids from unbounded domain.

Finally we show an intriguing relation between Strong-Renaming and the 0-1 Exclusion families, and make few conjectures about the implementations relationships among members of the collection, as well as possibly tasks outside it.

1 Introduction

Recently, in a surprising seminal result, it was shown that the sequence of tasks WSB $= WSB_2, WSB_3, ...$ is read-write solvable for certain values of n, and unsolvable for others [2]. The task WSB_n is a task on an infinite number of processors which are to output 0 or 1. The only requirement the task imposes is that n processors break-symmetry - i.e. when the participating set is of size n, at least one processor outputs 1 and at least one processor outputs 0. The family of tasks WSB was considered interesting since proving it impossible to solve was challenging. Indeed, it was proved in [3] that for all odd $n > 1$, WSB_n is weaker than n-processors Set-Consensus. The latter differ from the former in that rather than a task on infinite number of processors,

[*] This work was done while the author was on Sabbatical at Microsoft MSR-SV.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 246–258, 2008.

it is a task on processor 1 to n, and for participating set smaller than n it imposes the constraint that at least one processor outputs 0.

What can one conclude from this erratic behavior of the members of the WSB family, some read-write solvable some not? To us it indicated that in the first place there was no precise rationale to consider all size instances of WSB simultaneously. The tasks in the sequence do not make a cohesive whole. What makes a sequence a cohesive whole, a family? We have no answer to this general question, but to get there, we first asked whether we can find a task on n processors which is unsolvable for all n and is strictly weaker than Set-Consensus.

We started with a set of tasks which is as cohesive as they come: A Uniformly Solvable tasks [5]. We took The Uniformly Solvable task specified by an infinite sequence of bits $b(1), b(2), ...$ in which a processor outputs 0 or 1. When the size of the participating set is $k > 0$ then the only constraint is that processors will not all output $b(k)$. Obviously this is a Uniformly Solvable task as there is a single algorithm to solve it: A processor that observes a size of participating set k outputs $1 - b(k)$. We then asked what happens to this task if we impose one additional constraint: When the size of the participating set is some single value $n > 1$, then processors are not only precluded from all outputing $b(n)$, but they are also precluded from all outputing $1 - b(n)$, i.e. when the size of the participating set is n at least one processor outputs 1 and at least one outputs 0. Notice that we got the the 0-1 member by removing a single n-tuple from the Uniform task. For each n we get a different task on n processors $p_1, ..., p_n$.

It is known that when we start with the Uniformly Solvable task in which for all k $b(k) = 1$, when we fix an n and require that for participating set of size n at least one processor outputs 1 and one processor outputs 0, then we get a task which is equivalent to Set-Consensus on n processors [8]. But what about other sequences $b(k)$?

The main contribution of this paper is the discovery of this novel collection of families of tasks - each family specified by a different $b(k)$ sequence. In the initial investigation of this collection, to show that families in this collection are interesting, we concentrate on the family specified by the sequence $b(1) = 1$ and for all $k > 1$, $b(k) = 0$. We call the resulting sequence of tasks one for each $n > 1$, SEA (Single-Exclusion-Alternation). Our reasoning will hold verbatim for $n > k$ and the sequence $b(j) = 1,\ j = 1, ..., k - 1$ and $b(j) = 0,\ j > k - 1$.

We show some broad brush results pertaining the whole collection, and some results for SEA in particular. As for the particular, we first show that the question of the solvability of 0-1 tasks is decidable [11]: I.e. we show how given a 0-1 task we can construct a combinatorial expression and evaluate it to decide whether the task is solvable or not. We were able to evaluate the expression for all SEA member and thus show that SEA is read-write unsolvable for all n, and for n odd is strictly easier than set consensus. As yet we did not find a 0-1 solvable task.

We speculate that a family has a "monotonicity" property - the $n + 1$'st instant in a family is not "less-solvable" than the n'th instant. For instance, if the $n + 1$'st instant is say equivalent to Set-Concensus on $n + 1$ processors then the n'th instant is at least as hard as Set-Consensus on n processors.

Before we enumerate all our 0-1 Exclusion results pertaining to the whole collection, we need some simple definitions. Lets T_n be an n-processors 0-1 task. For $0 < k < n$

let $b_{T_n}(k)$ be the bit excluded for participating set of size k. We define two associated tasks, the complement task $C(T_n)$, and the dual task $D(T_n)$, both on n processors. For $C(T_n)$, $b_{C(T_n)}(k) = 1 - b_{T_n}(k)$. I.e it is the same task only replacing 0's with 1's and 1's with 0's. For $D(T_n)$, $b_{D(T_n)}(k) = 1 - b_{T_n}(n - k)$. I.e. if we take the complement of the dual we get a task whose exclusion sequence is the reverse sequence of original exclusion sequence. Obviously, the dual of the dual is the original. Finally, if T_n is a 0-1 Exclusion task, then the task $SYM(T_n)$ is a task on $2n - 1$ processors, that for any participating set of size $0 < k < n + 1$ allows the same set of tuples that T_n allows. The task $SYM(T_n)$ when viewed on n processor of ids 1,...,n essentially requires that a solution is symmetric, processors can only compare their ids rather have, say, a priori different programs for processors whose ids is odd and another one for processors whose ids is even. Thus $SYM(T_n)$ is potentially harder than T_n.

Let T_n be a member of the 0-1 Exclusion family. Here we enumerate the results:

1. The four tasks T_n, $C(T_n)$, $D(T_n)$, and $C(D(T_n))$, are equivalent.
2. $SYM(T_n)$ is not harder (i.e. it is equivalent) than T_n.
3. SEA is unsolvable for all n.
4. SEA is strictly weaker than Set-Consensus for n odd.
5. Any member of the 0-1 Exclusion family is weaker (not necessarily strictly) than Set-Consensus.
6. Any member in the family of 0-1 Exclusion solves WSB.

Thus the 0-1 Exclusion family is sandwiched between Set-Consensus on the top and WSB at the bottom.

We apply the idea of creating an unsolvable task by “thinning” a Uniform Solvable task, to the task of $(n, 2n - 1)$-Strong Renaming. This is a task on infinitely many processors. When the participating set is of size n, each processor has to output an integer between 1 and $2n - 1$ and no two processors output the same integer.

We propose many potentially unsolvable sequences of tasks by choosing a combination of n integers between 1 and $2n - 1$, and eliminating all the output n-tuples that contain just them. Unlike $(n, 2n - 2)$-Strong-Renaming which eliminates an “integer,” $2n-1$, in the sense that it precludes any processor from outputing it, we eliminate much less - just a single combination of n integers out of the n choose $2n - 1$ available. We show that each instance created this way solves a member of the family of the 0-1 Exclusion. Furthermore, we show that all these instances are not as strong a Set-Consensus. We solve Strong-Renaming instance on 3 processors that excludes the combination $(2, 3, 5)$, by using SEA_3 which we prove is strictly weaker than Set-Consensus. Thus 0-1 exclusion has the potential to be the “weakest” among unsolvable tasks that are created in a way yet to be formalized, from Uniformly Solvable task.

The paper is organized as follows: The next section is the Model Section. “Walking the talk,” there is very little “topology” in our paper. Most of our argumentation is algorithmic. Nevertheless, in arguing the unsolvability of SEA, we will assume that the reader is familiar with the definition of subdivided simplex, its faces, etc., notions that appear in most standard 3rd year Computer-Science Algorithms books. We will just quote a result concerning invariance over the different subdivision of a subdivided simplex. The “weakness” of SEA viz Set-Consensus follows verbatim from [3], we thus do not copy that model section. The "weakness" here is exactly in the same sense that it is there. It was there and it is here, proved under the Round-by-Round Hypothesis. The result holds in a Round-by-Round model [12], a model hypothesized to be equivalent to the general model. Aside from the section on tasks to get the vocabulary settled, the results we rely on are just out of other papers. We do not develop any new topological tool in this paper. We nevertheless, in the model section, have gone the path of unifying terminology in terms of output $tuples$. Since this is a sideshow of this paper, it may end up to be a bit terse. We then prove our results viz the 0-1 family, and SEA in particular. Following that, we discuss the relation between Strong-Renaming and the 0-1 exclusion family. Finally, we conclude with rehashing the numerous open problems and conjectures left on the table.

2 Model

2.1 Tasks

In Distributed Computing, the analogue of a $function$ that a Turing-Machine computes, is a $task$. A task is independent of the model of computation it is to be solved in. A task encompasses the level of coordination it requires. A consensus task requires the strongest coordination, while a task in which processors output their id requires no coordination. Thus given a task it will be solved in a model only if this model provides the level of coordination the task requires. In that sense, if we somehow define the notion of how one task can solve another, then a model is a set of tasks that is closed under task-solvability.

The only thing we will require from any model of distributed computing is to have the notion of $participating - set$, that is a definition that allows us to take an infinite distributed computation in the model and decide which processors participate in that computation.

To define a task T on n processors, $p_1, ..., p_n$, we consider any subset of the processors as participating set. The specification of the task is a map from each participating set P to a set $\Delta_T(P)$ of $|P| - tuple$s where a $|P|$-tuple is a set of $|P|$ pairs $\{(p_i, w \in O) | p_i \in P\}$ for some w in an output values set O. We will discard the $|P|$ in the definition of a tuple as it will be understood from the context.

The interpretation is that if the participating set is P, and all processors in P output, then the combination results in a tuple in $\Delta_T(P)$.

For example, in the $k-$Set-Consensus task [13] on processor $p_1, ..., p_n$, processors output ids of processors. $\Delta_T(P)$ is such that for each pair $(p_i, w) \in tuple$, $tuple \in \Delta_T(P)$ implies $w \in P$, and $|\{w | (p_i, w) \in tuple\}| < k + 1$. We will use the shorthand Set-Consensus when the universe of processors is n processors and $k = n - 1$.

If we want to model a situation when "p_i may start with different inputs," as traditionally tasks are defined [6], we just increase the universe of processors and distribute the inputs among processors as to get each processor to have a single input value. With a single input value we can ignore the value as the processor's id incorporates it.

For instance, to define the task $binary - consensus$ [9] we define it on $2n$ processors. The output set is $O = \{0, 1\}$. For participating set of size less then $n + 1$ output values in a tuple have to agree, they are either all 0, or all 1. Furthermore, if the participating set is a subset of $\{p_1, ..., p_n\}$ then processors are to output 0, if the participating set is subset of $\{p_{n+1}, ..., p_n\}$ then they are to output 1. If the cardinality of the participating set is strictly bigger than n then anything goes.

The the nth instance of Weak-Symmetry-Breaking (WSB_n) task [3], is a task on $2n-1$ processors. The output set is $O = \{0, 1\}$. When the cardinality of the participating set is n, then each output tuple contains some pairs in which the output value is 0, and some pairs in which the output value is 1. For other cardinalities, anything goes.

In the $(2n-1, q)$-Weak-Renaming task [1], the task is defined on $2n-1$ processors. The output set is $O = \{1, 2, ..., q\}$. For participating set of size $k = n$, no two pairs in the pairs of a tuple agree on the output value. For other cardinalities, anything goes.

The task (n, q)-Strong-Renaming (also called Adaptive-Renaming) on n processors [1], the output set is $O = \{1, 2, ..., q\}$. For participating set of size k, no two pairs in the pairs of a tuple agree on the output value, moreover, no output value in a pair is larger then $2k-1$.

Of particular interest is the Immediate-Snapshot task IS_n on n processors [10]. We define it recursively, as IS is a Uniformly Solvable task [5]. Task IS_1 on processor p_1 has a singleton tuple $(p_1, \{p_1\})$. Task IS_n extend IS_{n-1} in the following way: For participating set of size $k < n$ that include processor p_n, the output set of k-tuples is isomorphic to the output set for $p_1, ...p_k$ with any fixed one to one correspondence between processor's ids. For participating set of size $k = n$ we take add a new value to O say $o = \{p_1, ..., p_n\}$, create the n pairs $(p_1, o), ..., (p_n, o)$. We now create the n-tuples by taking any k-tuple from any participating set of size k and inserting enough new pairs to make it syntactically valid n-tuple.

2.2 Tasks Solving Tasks

Let task A and B be defined on the same set of processors. We say that task A *solves* task B if there is a function s from pairs in A to pairs in B that preserves the first entry of the pair, i.e. the processor associated with the pair, such that every tuple in A is mapped to a tuple in B.

2.3 Sequential-Composition of Tasks

Given two two tasks A and B on the same set of processors we define the task BA which we call A *followed* by B. We create the output tuple for participating set P by taking a tuple $\{(p_i, w_a)\}_{p_i \in P} \in \Delta_A(P)$, and a tuple $\{(p_i, w_B)\}_{p_i \in P} \in \Delta_B(P)$, and adding a tuple $\{(p_i, (w_A, w_B))\}p_i \in P$ to $\Delta_{BA}(P)$.

A composition may come adjoined with a *termination* map. A termination map is a partial function on pairs that return $HALT$ or $CONT$. Once a pair $(p_i, *)$ is mapped to $HALT$, processor p_i is not in the participating set of the next task and the composition does not add anything further to the $(p_i, *)$.

Of particular interest is self composition. Consider a self composition of A with itself infinitely many times. If the composition is adjoined with a termination functions such that after finite number of composition every pair has $HALT$ed, then we'll call the composition $well-composed$. A k-stage composition for instance can be modeled by a termination map that $HALT$s a pair once it "accumulates" k output values.

2.4 Well-Composition of IS_n

For the purposes of this paper, every well-composition of IS_n with itself is called an $n-1$ dimensional *subdivided simplex*. The pairs of a well composed IS_n are of the form $(p_i, (P_1, P_2, ..., P_k))$ where this pair maps to $HALT$ and no prefix $(p_i, P_1, P_2, ..., P_q)$, $q < k$ does. Notice that in any tuple all the pairs map to $HALT$. A a k-tuple is called a k-simplex.

A *carrier* of a pair q, denoted $carrier(q)$, is the union of the sets in the sequence in the output value of q. For a k-tuple, the $(k-1)$-tuples that are sub-tuple of a k-tuple are its $k-2$-faces. The $k-1$-face of the subdivided simplex defined by a set of processors P such that $|P| = k$, are all the tuples the carrier of each pair of which is a subset equal P. All the $n-1$-tuples whose carriers are not of cardinality n are the *boundary* of the subdivided-simplex.

Consider IS_n. An n-tuple is of the form $\{p_i, P_i\}$. Let the parity of the tuple be odd or even according to the number of distinct sets in the pairs of the tuple. It is easy to check that if two n-tuple share $n-2$ face (that is made of $n-1$ processors), then they have different parity. We say that a sub-divided simplex is *orientable*. The parity function that maps n-tuples to "+" or "-" is the orientation.

We will use the following invariance result. It says that a 0,1 coloring of pairs in tuples that are on the boundary of a subdivided simplex determines a property below of the inside pairs for any legal "inside."

Let S be oriented subdivided simplex. If the pairs on the boundary are colored by 0 or 1, then for any 0,1 coloring of the rest of the pairs, we count in a certain way the number of n-tuples where all the pairs of the tuple are colored by the same color. This number is called the *content* of the subdivided simplex. We call a tuple whose all pairs are of the same color c, mono-c.

1. If n is odd: Sum the number of n-tuples mono tuple considering the sign it gets by the orientation.
2. If n is even: Do the above only that if a tuple is a mono-1 add an extra "-," i.e. a mono-1 with with positive orientation adds -, while negative orientation adds +1.

The result we will use is that once the boundary and its coloring is fixed the content is the same no matter what is the "inside" as long as the boundary remains fixed [2]. In particular given a boundary, then it was created by some stages of IS's, and some termination rules. In this paper will create a subdivided simplex with the same boundary by changing the halting rule. On the boundary we main the same rule as before while we halt a processor after the first stage if it returns the set of all processors. I.e. we take the boundary, we plant a single "inside" simplex and cone it off with the appropriate faces.

If a well-composed IS_n subdivided simplex A_n solves a 0-1 Exclusion task T_n, let c be the binary coloring of vertices of A_n induced by the solution, then the content of A_n is 0. This is a corollary of [6] as it implies that to be a solution the induced coloring has to have no mono n-tuple.

Let B_n be a well-composed IS_n. Eliminate some internal n-tuples (an n-tuple that has no pair on the boundary) to get $B_n(holes)$. If the task $B_n(holes)$ solves a 0-1

Exclusion task T_n, then the binary coloring induced on the boundary of B_n is such that the content of B_n for any coloring of the interior vertices is some a priori fixed number $C(T_n)$.

This follows from the fact that we can count the content by the particular subdivision mentioned above of planting a single internal n-tuple and coloring it 0, then the only possible monos are mono-0's. According to Corollary 3 to follow, if $B_n(holes)$ solves a 0-1 Exclusion task T_n, then the boundary faces are w.l.o.g. a single IS where the middle simplex in the boundary is colored by $1 - b_{T_n}(k)$. Thus it is a simple combinatorial exercise to find $C(T_n)$

2.5 SWMR-Atomic-SM and Immediate-Snapshots

We will assume the standard asynchronous Atomic SM [7]. In this model processors alternate between writing in their SM cell and reading all SM cells in a snapshot. Cells are initialized to $\perp$. An execution is a sequence of processors ids where the first appearance of a processor is interpreted as a $write$ and its next appearance in the sequence as a $read$ etc. A processor $participates$ if it appears in the sequence. We will assume "full-information" model in which each $write$ appends to its cell what it read in the preceding $read$. Following a $read$ a processor holds a $view$. The view is what would be the content of its cell in the next $write$.

A protocol π is a partial function for each processor from its view to an output.

We will say that a protocol π solves T_n wait-free in the model, if for all infinite executions, all processors that appear infinitely many times have views that map to outputs. Furthermore, if all processors in an execution with a participating set P output, then the tuple $tuple$ they created satisfies $tuple \in \Delta_{T_n}(P)$.

We will use the following result [12]:

Theorem 1. *A task T_n is solvable in SM, iff it is solvable by some well-composed IS_n.*

2.6 Task Implementation

We say that task A implements task B, if the SM model equipped with A as a procedure, B is solvable.

2.7 The Round-by-Round Hypothesis

The following has not been proved yet, but is believed by all. When we say that SEA is weaker than Set-Consensus it is modulo this Hypothesis.

The Round-by-Round Hypothesis: A implements B, iff some well composition of $A(IS_n)$ solves B.

3 Properties of 0-1 Exclusion and Family and the SEA Task

3.1 The Four Tasks T_n, $C(T_n)$, $D(T_n)$, and $C(D(T_n))$, are Equivalent

The only nontrivial part is the follows:

Theorem 2. *The task T_n and $D(T_n)$ are equivalent.*

Proof. Since $D(D(T_n)) = T_n$, it suffice to show that T_n implements $D(T_n)$.

Let each processor register in shared memory and then each take a snapshot to see the number of registrants. If the number is $k,\ 0 < k < n$, then the processor outputs $b_{T_n}(n-k)$, else, if $k = n$ it goes to T_n and output from it.

To see that this solves $D(T_n)$, notice that if the participating set is of size $k,\ 0 < k < n$, then at least one processor will output $b_{T_n}(n-k)$. Thus the tuple of all $1 - b_{T_n}(n-k) = b_{D(T_n)}(k)$, is avoided.

If the participating set is $k = n$, then consider the case that $r,\ n > r > 0$ is the number of processors that went to obtain output from T_n. Since T_n excluded $b_{T_n}(r)$ at least one of the r processors will output $1 - b_{T_n}(r)$. Of the $(n-r)$ processors that did not go to T_n, at least one will see $n-r$ initial registrants and output $b_{T_n}(n-(n-r)) = b_{T_n}(r)$. This one will be the processor that executed the last $read$ that resulted in that the number of registrants is less than n. Thus, not all output 0 and not all output 1.

If $r = n$, then by the specification of T_n not all output 0 and not all output 1.

Corollary 1. *If there exits a read-write algorithm for T_n then there exists a algorithm by which a processor stops after its first write-read step if it observes a participating set of size $k < n$. It then outputs $1 - b_{T_n}(k)$.*

Proof. Consider the implementation of T_n from $D(T_n)$.

3.2 $SYM(T_n)$ Is Not Harder (i.e. Equivalent) to T_n

Theorem 3. *$SYM(T_n)$ is not harder (i.e. equivalent) than T_n.*

Proof. Take T_n with ports 1 to n. Processors register in a $registered$ set, and then observe a toggle bit called a "gate" which has two states, it can be "open" or "closed." The gate is initially open. A processor that observes the gate open, Strongly Rename [1] to obtain an output we call *port*. If he number of the port obtained is less than $n+1$ it invokes T_n at that port and departs with an output from T_n. If the port it gets is greater than n or initially it observed the gate closed it registers in a $spill-over$ set, and toggles the gate to close.

If all processors renamed into T_n we are done. Else, at least one processor must rename into T_n. If $r,\ 0 < r < q \leq n$ rename into T_n then at least one of those will output the bit $1 - b_{T_n}(r)$. To get the complement bit by at least one processor, let each p_j in the $spill-over$ set, take a snapshot and get $spill-over_j$ and $registered_j$. It then outputs $b_{T_n}(|registered_j| - |spill - over_j|)$. Since at least one processor, the last one to take a snapshot, will obtain $|registered_j| = q$ and $|spill - over_j| = q - r$, then at least one will output $b_{T_n}(q-(q-r)) = b_{T_n}(r)$. Thus at least two processors will output different bits. A crucial observation is that Strong-Renaming will never rename all the processors invoking it to the $spill - over$ set, hence we are allowed to state $r > 0$. The function of the gate is to prevent a processor who arrived after a processor from the $spill - over$ set departed, to go the T_n. This will follow from the fact the processors in the $spill - over$ set close the gate, and no processor ever opens it. We do this in order to maintain that at least one processor from the spill-over set will see the correct

Algorithm 1. $SYM(T_n)$ from T_n

```
Initially: $REG[1..n] = SPLOVR[1..n] = FALSE,, GATE = FALSE, PORT[1..n]$
  $RREG[1..n], RSPLOVR[1..n]$

$REG[i] := TRUE$;
if $GATE = TRUE$ then
    $PORT[i] :=$call(Strong-Renaming);
    if $PORT[i] < n + 1$ then
        return(call($T_n$ at port $PORT[i]$))
    end if
end if
$GATE := FALSE$;
$SPLOVR[i] := TRUE$;
$RREG[i] := |\{j|REG[j] = TRUE\}|$;
$RSPLOVR[i] := |\{j|RSPLOVR[j] = TRUE\}|$;
return($b_{T_n}(RREG[i] - RSPLOVR[i])$)
```

final cardinality of the $registered$ set, q, and the final cardinality of the $spill - over$ set $(q - r)$. This follows from he fact that after the last processor to register into the $spill - over$ set does so, no process will register, period.

The code appears as algorithm 1.

3.3 SEA Is Unsolvable for All n

Recall that SEA is a 0-1-Exclusion task specified by the sequence of bits $b(1) = 1$, $b(2) = \ldots = b(n-1) = 0$.

Theorem 4. *SEA is unsolvable for all $n > 1$.*

Proof. Let NC be a subdivided simplex that solves SEA. To be a solution, the number of mono n-tuples should be zero, while the boundary should satisfy the specification of the task.

It follows from [2], and discussed in the Model Section, that given a boundary of a chromatic subdivided simplex that is colored by 0 and 1 any other chromatic subdivision that agrees with it on the boundary will result in the same Content. Thus we can count the Content using a particularly convenient subdivision. If this subdivision results in a Content that in absolute value different than 0, then it is a proof that no other subdivision will get what we need, which is a count of 0. We will do this by taking the boundary of NC, planting an $n - 1$ chromatic simplex in it and coning the appropriate faces of the boundary. We will consider the binary coloring of all the nodes of the middle n-tuple to be 0.

By the specification of the problem aside from the 0-dim face, no proper face has a mono-0 tuple. (Alternatively, by the corollary 3 we can assume w.l.o.g that the only 0's on the boundary are the nodes that are the 0-dim faces.) Since the nodes internal to the $n - 1$ face are all 0 by construction, then the only mono n-tuples are the mono-0. Thus it is easy to see that the only mono-0 n-tuple are the middle one, and all n combinations of choosing $n-1$ vertices out of the middle n-tuple and combining it with the 0-dim face across from it. Thus, we get $n+1$ mono-0 simplexes. If the orientation of the middle simplex is +, then all other simplexes share an $n-1$ face with it and therefore are all of orientation -. Consequently the Content is $-(n-1)$. Thus, for $n>1$ we have that the content is strictly different than 0.

Corollary 2. *For n odd if we use the technique in [2] to eliminate a pair of positive and negative mono-0's, and we unify the rest of $n-1$ mono-0 simplexes in pairs, we obtain a pseudo-manifold that solves SEA_n.*

3.4 SEA Is Strictly Weaker Than Set-Consensus for n Odd

Theorem 5. *SEA is strictly weaker than Set-Consensus for n odd.*

Proof. The construction in [3] that shows that Renaming is easier than set consensus happen to be binary colored in a way that satisfies not only the requirements of WSB, but also satisfies the requirements of the stronger problem SEA, and therefore it also proves that Set-Consensus is strictly stronger than SEA. (See also Corollary 6.)

3.5 Any Member of the 0-1 Exclusion Family Is Weaker (Not Necessarily Strictly) Than Set-Consensus

Theorem 6. *Any member of the 0-1 Exclusion family is implementable by Set-Consensus.*

Proof. Using $(n, n-1)$-Set-Consensus by which n processor output at most $n-1$ distinct names of participating processors one can solve $(n, 2n-2)$-Strong Renaming where for participating set of size $k<n$ processors rename between 1 and $2k-1$, while for $k=n$, they rename between 1 and $2n-2$ [4].

Consider a member in the 0-1 Exclusion family specified by $b(k),\ k=1,...,n-1$. We color the integers $r=1,...,2n-2$ by 0 and 1. Let the color of i be $CS(i)$. We color as follows:

1. $CS(1)=1-b(1)$,
2. If $b(k)=b(k-1), 1<k<n$ then $CS(2k-1)=0$, and $CS(2k-2)=1$,
3. If $b(k)\neq b(k-1), 1<k<n$ then $CS(2k-1)=CS(2k-2)=b(k-1)$,
4. $CS(2n-2)$ is colored by the color that is minority slots in $CS(1)$ up to $CS(2n-3)$.

It is an easy induction to see that this coloring has the invariant that $abs(|\{0<i<2k<2n-1|CS(i)=1\}|-|\{0<i<2k<2n-1|CS(i)=0\}|)=1$, and moreover the majority of colors is $1-b(k)$. Also if we consider all the integers, the number that is colored by 1, equal the number that is colored 0, equals $n-1$. Consequently, if a processor outputs the color of the integer it obtains we satisfy the specification of the 0-1 Exclusion problem.

3.6 Any Member in the Family of 0-1 Exclusion Solves Impossible Weak-Renaming

Theorem 7. *Any member in the family of 0-1 Exclusion solves WSB.*

Proof. Follows from the result that T_n is equivalent to $SYM(T_n)$, and the latter is a restriction of WSB.

4 Strong-Renaming and the 0-1 Exclusion Family

In the task $(n, 2n-1)$-Strong-Renaming if the participating set if of size $1 \leq k \leq n$ then the k processor each return a unique integer from the range 1 to $2k-1$. It is known that $(n, 2n-2)$-Strong-Renaming is equivalent to Set-Consensus [4].

What if we fix some n integers out of the integers 1 to $2n-1$ and we preclude all n processors to together return these n combination? Call such a ask Strong-Renaming-Minus. Precluding processors from returning the integer $2n-1$ is much more, it precludes them from returning any tuple with a pair the contains the integer $2n-1$. Thus, this is potentially an easier problem. Is there a single n-combination that can be eliminated and the problem can still be read-write solvable.

We only partially answer this question. We show that the elimination of an n-combination results in a problem that is potentially harder than some member of the 0-1 Exclusion. If it will be proven that all members of the 0-1 Exclusion family are unsolvable then the elimination of any n-combination results in an unsolvable task.

On the flip side we show that there exists a collection n-combinations the elimination of any will results in a task which is equivalent to Set-Consensus. Perhaps any single combination elimination results in a task equivalent to set consensus? We answer on the negative. We show that for $(2, 3, 5)$-Strong-Renaming-Minus, if we eliminate the combination 2,3,and 5 then SEA on 3 processors solves the problem. Since for 3 processors SEA is shown to be strictly easier than Set-Consensus, it shows the result.

4.1 Reducing Strong-Renaming-Minus to 0-1 Exclusion

It is easy to see that if we eliminate the combination $(n, n+1, ..., 2n-1)$ then we get a task which is equivalent to set consensus: Set-Consensus solves $(n, 2n-2)$-Strong-Renaming [4]. The task $(n, 2n-2)$-Strong-Renaming is harder than the task at hand. Thus Set-Consensus solves the task. To see that our task solves Set-Consensus, let processors that output between 1 and $n-1$ output 0, and above $n-1$, output 1. It is easy to see by the specification of Strong-Renaming that for $k < n$, at least one processor must output 0. Once we have a participating set of size n, they cannot all output 0, as there are only $n-1$ positions of outputing 0. On the other hand they cannot all output 1, as the any tuple that will result it is eliminated. Thus, the elimination of that single combination is equivalent to the elimination of the $2n-1$'st integer, i.e. all the tuples the contain position $2n-1$. To get the same result we do not necessarily have to eliminate the combination $(n, n+1, ..., 2n-1)$. Obviously the above reasoning goes through as long as for any $k < n$ the number of integers in the n-combination whose value is less equal $2k-1$ is less than k.

What about eliminating other tuples? Suppose we create the task SR_{comb} from $(n, 2n-1)$-Strong-Renaming by eliminating the combination $comb$. We show how to create a 0-1 Exclusion member E_{comb} such that SR_{comb} solves E_{comb}.

Consider all the integers in $comb$ to be colored by 1, and the rest of the integers by 0. Denote the color of integer i by $CS(i)$. Create the 0-1 Exclusion task E_{comb}, such that for participating set $0 < k < n$ one excludes $b(k)$. The bit $b(k)$ is the minority bit of the colors of the integers 1,...,$2k-1$, i.e. if $|\{i|CS(i) = 0, i \leq 2k-1\}| \leq k-1$ then $b(k) = 1$, else $b(k) = 0$.

If processors in SR_{comb} return the color of the integer they obtained, then they solved E_{comb}.

4.2 A Strong-Renaming-Minus Task Equivalent to a Weak 0-1 Exclusion Member

We now show how for 3 processors, $E_{(2,3,5)}$ implements $SR_{(2,3,5)}$: For $E_{(2,3,5)}$ we have $b(1) = 1, b(2) = 0$. The implementation follows. For historical reasons and metaphors we will use the word "slot" as a synonym with integer. Processors obtain an output from $E_{(2,3,5)}$. If a processor obtained 0 it raises a flag for slot 1, if it obtained 1 it raises a flag for slot 2. It then checks whether another processors raises a flag for the same slot (it cannot be that 3 processor will). If not then it return the slot. Else, if it is smaller among the two and its flag is on slot 2, it returns slot 3. If it is the bigger among the two and its flag is on slot 2, it raises a flag for slot 1. If it smaller among the two and its flag is on slot 1 it returns 4, else it returns 5.

This algorithm is a finite state machine and I have checked all possibilities. The idea though is that processors that return 0, start strong renaming on slots colored 0 and processors that returned 1 start strong renaming on slots color 1. They nevertheless are not allowed to go above slot $2k - 1$. Thus if they spill-over they start going backward on the complement colored slots.

5 Conclusions

The trigger to this paper is the recent discovery that WSB is solvable for certain values of n [2]. This means that different size instance of WSB do not make a cohesive whole. To make a cohesive collection of solvable tasks we proposed in the past the notion of Uniform Solvability [5]. The idea behind this paper is to create a cohesive unsolvable collection by taking a solvable cohesive collection and eliminating output tuples. It resulted in the discovery of the 0-1 Exclusion family as well as the weaker unsolvable versions of Strong-Renaming.

The main advance we miss is a formalization of this process of deriving what we would term a Uniformly Unsolvable task out of Uniformly solvable one.

For instance, why excluded the all 1's or the all 0's tuple and not a collection of tuples of the type, say, one processor output 0 and all the rest 1's? Why eliminate an n-combination the case of Strong-Renaming, rather than a single n-tuple. We have no good rationale for that as yet.

On the technical side the main question left is the impossibility of all the instances of the 0-1 family for any n, as well as finding the implementation relationships between members of the family. It is easy to see that unlike WSB given an exclusion task on n processors the the Content of any solution is the same over all solutions. Thus we conjecture that given two Exclusion tasks $T1$ and $T2$ on n processors, one with Content a_1 and the other with a_2, then $T1$ implements $T2$ if and only if a_1 divides a_2, otherwise, if none is a divisor of the other, then they are unrelated. Since the Content of Set-Consensus is 1 it solves the whole family.

We also conjecture that once the notion of Uniform Unsolvability is formalized, then any Uniformly Unsolvable task solves some instance of the 0-1 Exclusion family. We have shown this promise via example by taking Strong-Renaming, creating supposedly Uniformly Unsolvable task out of it, and show that indeed, every instance implements some 0-1 Exclusion family member.

For one and half decades Weak-Renaming was the only game in town. When pressed for a task possibly weaker than Set-Consensus only Renaming came to mind. How did we miss the 0-1 Exclusion and Excluding a combination from Strong-Renaming rather than a whole integer? I guess the answer to this does not lie within Distributed Computing but within Human-Nature. Once our guns turned on Weak-Renaming impossible, back in 1993 [6], too much focus on it held us back from looking to the sides for an easier prey.

Acknowledgment. Yehuda Afek caught a mistake that brought in the "gate" in the proof of Theorem 4. He is also responsible for the Exclusion of some tasteless jokes, and some of too many remarks about "life" as in the last paragraph of the Conclusions. I am in debt to Armando Castaneda for answering my numerous questions. I owe Sergio for being Sergio.

References

1. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an Asynchronous Environment. J. ACM 37(3), 524–548 (1990)
2. Castaneda, A., Rajsbaum, S.: New Combinatorial Topology Upper and Lower Bounds for Renaming. In: PODC 2008 (to appear, 2008)
3. Gafni, E., Rajsbaum, S., Herlihy, M.: Subconsensus Tasks: Renaming Is Weaker Than Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 329–338. Springer, Heidelberg (2006)
4. Mostfaoui, A., Raynal, M., Travers, C.: Exploring Gafni's Reduction Land: From Omegak to Wait-Free Adaptive (2p-[p/k])-Renaming Via k-Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 1–15. Springer, Heidelberg (2006)
5. Gafni, E.: A Simple Algorithmic Characterization of Uniform Solvability. In: FOCS 2002, pp. 228–237 (2002)
6. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
7. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merrit, M., Shavit, N.: Atomic Snapshots of Shared Memory. In: Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC 1990), pp. 1–13. ACM Press, New York (1990)
8. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: Proc. 25th ACM Symposium on the Theory of Computing (STOC 1993), pp. 91–100. ACM Press, New York (1993)
9. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
10. Borowsky, E., Gafni, E.: Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). In: Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC 1993), pp. 41–51. ACM Press, New York (1993)
11. Gafni, E., Koutsoupias, E.: Three-Processor Tasks Are Undecidable. SIAM J. Comput. 28(3), 970–983 (1999)
12. Borowsky, E., Gafni, E.: A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). In: Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC 1997), pp. 189–198. ACM Press, New York (1997)
13. Chaudhuri, S.: Agreement is Harder than Consensus: Set Consensus Problems in Totally Asynchronous Systems. In: PODC 1990, pp. 311–324 (1990)

Interval Tree Clocks

A Logical Clock for Dynamic Systems

Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte

DI/CCTC, Universidade do Minho
Largo do Paço, 4709 Braga Codex, Portugal
{psa,cbm,vff}@di.uminho.pt

Abstract. Causality tracking mechanisms, such as vector clocks and version vectors, rely on mappings from globally unique identifiers to integer counters. In a system with a well known set of entities these ids can be preconfigured and given distinct positions in a vector or distinct names in a mapping. Id management is more problematic in dynamic systems, with large and highly variable number of entities, being worsened when network partitions occur. Present solutions for causality tracking are not appropriate to these increasingly common scenarios. In this paper we introduce *Interval Tree Clocks*, a novel causality tracking mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralized creation of processes/replicas without need for global identifiers or global coordination. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. The representation is so compact that the mechanism can even be considered for scenarios with a fixed number of entities, which makes it a general substitute for vector clocks and version vectors.

Keywords: Causality, logical clock, version vectors, vector clocks, dynamic systems.

1 Introduction

Ever since causality was introduced in distributed systems [12], it has played an important role in the modeling of distributed computations. In the absence of global clocks, causality remains as a means to reason about the order of distributed events. In order to be useful, causality is implemented by concrete mechanisms, such as Vector Clocks [7,16] and Version Vectors [18], where a compressed representation of the sets of events observed by processes or replicas is kept.

These mechanisms are based on a mapping from a globally unique identifier to an integer counter, so that each entity (i.e. process or replica) keeps track of how many events it knows from each other entity. A special and common case is when the number of entities is known: here ids can be integers, and a vector of counters can be used.

Nowadays, distributed systems are much less static and predictable than those traditionally considered when the basic causality tracking mechanisms were created. In dynamic distributed systems [17], the number of active entities varies during the system execution and in some settings, such as in peer-to-peer deployments, the level of change, due to churn, can be extremely high.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 259–274, 2008.

Causality tracking in dynamic settings is not new [8] and several proposals analyzed the dynamic creation and retirement of entities [21,9,20,13,2]. However, in most cases localized retirement is not supported: all active entities must agree before an id can be removed [21,9,20] and a single unreachable entity will stall garbage collection. Localized retirement is only partially supported in [13], while [2] has full support but the mechanism itself exhibits an unreasonable structural growth that its practical use is compromised [3].

This paper addresses causality tracking in dynamic settings and introduces Interval Tree Clocks (ITC), a novel causality tracking mechanism that generalizes both Version Vectors and Vector Clocks. It does not require global ids but is able to create, retire and reuse them autonomously, with no need for global coordination; any entity can fork a new one and the number of entities can be reduced by joining arbitrary pairs of entities; stamps tend to grow or shrink, adapting to the dynamic nature of the system. Contrary to some previous approaches, ITC is suitable for practical uses, as the space requirement scales well with the number of entities and grows modestly over time.

In the next section we review the related work. Section 3 introduces a model based on fork, event and join operations that factors out a kernel for the description of causality systems. Section 4 builds on the identified core operations and introduces a general framework that expresses the properties that must be met by concrete causality tracking mechanisms. Section 5 introduces the ITC mechanism and correctness argument under the framework. Before conclusions, in Section 7, we present in Section 6 a simple simulation based assessment of the space requirements of the mechanism.

2 Related Work

After Lamport's description of causality in distributed system [12], subsequent work introduced the basic mechanisms and theory [18,7,16,5]. We refer the interested reader to the survey in [22] and to the historical notes in [4]. After an initial focus on message passing systems, recent developments have improved causality tracking for replicated data: they addressed efficient coding for groups of related objects [14]; bounded representation of version vectors [1]; and the semantics of reconciliation [10].

Fidge introduces in [8] a model with a variable number of process ids. In this model process ids are assumed globally unique and are gradually introduced by process spawning events. No garbage collection of ids is performed when processes terminate.

Garbage collection of terminated ids requires additional meta-data in order to assess that all active entities already witnessed the termination; otherwise, ids cannot be safely removed from the vectors. This approach is used in [9,21] together with the assumption of globally unique ids. In [20] the assumption of global ids is dropped and each entity is able to produce a globally unique id from local information. A typical weakness in these systems is twofold: terminated ids cannot be reused; and garbage collection is hampered by even a single unreachable entity. In addition, when garbage collection cannot terminate, the associated meta-data overhead cannot be freed. Since this overhead is substantial, when the likelihood of non termination is high, it can be more efficient not to garbage collect and keep the inactive ids.

The mechanism described in [13] provides local retirement of ids but only for restricted termination patterns (a process can only be retired by joining a direct ancestor); moreover, the use of global ids is required.

Our own work in [2] introduced localized creation and retirements of ids and presented Version Stamps, a dynamic substitute to version vectors. Although still of theoretical interest as it does not use counters, and although it inspired the id management technique used in ITC, the technique was later found out to exhibit very adverse growth in common scenarios [3]. The id management technique used in version stamps shares many properties with credit management techniques in termination detection algorithms [15,11].

In order to control version vector growth, in Dynamo [6] old inactive entries are garbage collected. Although the authors tune it so that in production systems errors are unlikely to be introduced, in general this can lead to resurgence of old updates. Mechanisms like ITC may help in avoiding the need for these aggressive pruning solutions.

3 Fork-Event-Join Model

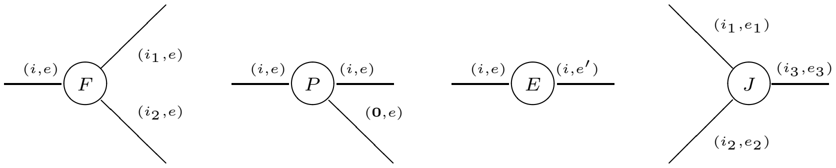
Causality tracking mechanisms can be modeled by a set of core operations: fork, event and join, that act on stamps (logical clocks) whose structure is a pair (i, e), formed by an id and an event component that encodes causally known events. Fidge used in [8] a model that bears some resemblance, although not making explicit the id component.

Causality is characterized by a partial order over the event components, $(E, \leq)$. In version vectors, this order is the pointwise order on the event component: $e \leq e'$ iff $\forall k.\ e[k] \leq e'[k]$. In causal histories [22], where event components are sets of event ids, the order is defined by set inclusion.

Fork. The fork operation allows the cloning of the causal past of a stamp, resulting in a pair of stamps that have identical copies of the event component and distinct ids; $\mathrm{fork}(i, e) = ((i_1, e), (i_2, e))$ such that $i_2 \neq i_1$. Typically, $i = i_1$ and i_2 is a new id. In some systems i_2 is obtained from an external source of unique ids, e.g. MAC addresses. In contrast, in Bayou [20] i_2 is a function of the original stamp $f((i, e))$; consecutive forks are assigned distinct ids since an event is issued to increment a counter after each fork.

Peek. A special case of fork when it is enough to obtain an *anonymous* stamp $(\mathbf{0}, e)$, with "null" identity, than can be used to transmit causal information but cannot register events, $\mathrm{peek}((i, e)) = ((i, e), (\mathbf{0}, e))$. Anonymous stamps are typically used to create messages or as inactive copies for later debugging of distributed executions.

Event. An event operation adds a new event to the event component, so that if (i, e') results from $\mathrm{event}((i, e))$ the causal ordering is such that $e < e'$. This action does a strict advance in the partial order such that e' is not dominated by any other entity and does not dominate more events than needed: for any other event component x in the system, $e' \not\leq x$ and when $x < e'$ then $x \leq e$. In version vectors the event operation increments a counter associated to the identity in the stamp: $\forall k \neq i.\ e'[k] = e[k]$ and $e'[i] = e[i] + 1$.



(a) fork **(b)** peek **(c)** event **(d)** join

Fig. 1. Core operations

Join. This operation merges two stamps, producing a new one. If $\mathrm{join}((i_1, e_1), (i_2, e_2)) = (i_3, e_3)$, the resulting event component e_3 should be such that $e_1 \leq e_3$ and $e_2 \leq e_3$. Also, e_3 should not dominate more that either e_1 and e_2 did. This is obtained by the order theoretical join, $e_3 = e_1 \sqcup e_2$, that must be defined for all pairs; i.e. the order must form a join semilattice. In causal histories the join is defined by set union, and in version vectors it is obtained by the pointwise maximum of the two vectors.

The identity should be based on the provided ones, $i_3 = f(i_1, i_2)$ and kept globally unique (with the exception of anonymous ids). In most systems this is obtained by keeping only one of the ids, but if ids are to be reused it should depend upon and incorporate both [2].

When one stamp is anonymous, join can also model message reception, where $\mathrm{join}((i, e_1), (\mathbf{0}, e_2)) = (i, e_1 \sqcup e_2)$. When both ids are defined, the join can be used to terminate an entity and collect its causal past. Also notice that joins can be applied when both stamps are anonymous, modeling in-transit aggregation of messages.

Classic operations can be described as a composition of these core operations:

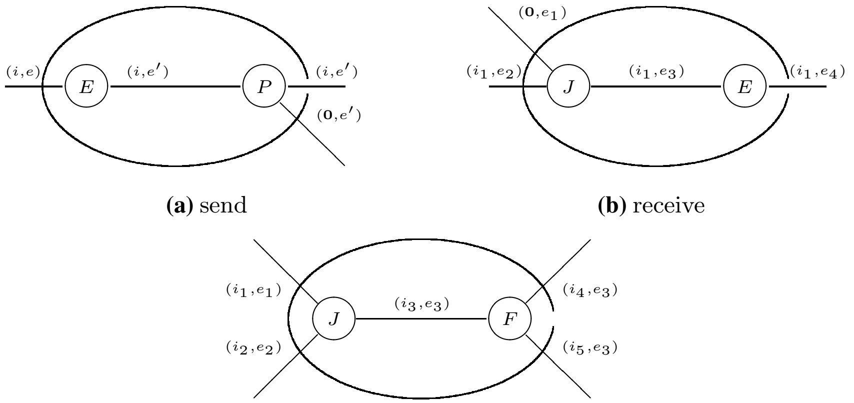
Send. This operation is the atomic composition of event followed by peek. E.g. in vector clock systems, message sending is modeled by incrementing the local counter and then creating a new message.

Receive. A receive is the atomic composition of join followed by event. E.g. in vector clocks taking the pointwise maximum is followed by an increment of the local counter.

Sync. A sync is the atomic composition of join followed by fork. E.g. In version vector systems and in bounded version vectors [1] it models the atomic synchronization of two replicas.

Figure 2 depicts graphical representations of these composite operations, but other composite operations could also be easily described using the same set of core operations. For instance, a message multicast could be modeled as the atomic composition of an event operation followed by a sequence of peek operations.

Traditional descriptions assume a starting number of entities. This can be simulated by starting from an initial *seed* stamp and forking several times until the required number of entities is reached.



(a) send

(b) receive

(c) sync

Fig. 2. Some composite operations

4 Function Space Based Clock Mechanisms

In this section we present a general framework which can be used to explain and instantiate concrete causality tracking mechanisms, such as our own ITC presented in the next section. Here stamps are described in terms of functions and some invariants are presented towards ensuring correctness. Actual mechanisms can be seen as finite encodings of such functions. Correctness of each mechanism will follow directly from the correctness of the encoding and from respecting the corresponding semantics and conditions to be met by each operation. In the following we will make use of the standard pointwise sum, product, scaling, partial ordering and join of functions:

$$(f+g)(x) \doteq f(x) + g(x),$$

$$(f \cdot g)(x) \doteq f(x) \cdot g(x),$$

$$(n \cdot g)(x) \doteq n \cdot g(x),$$

$$f \leq g \doteq \forall x.\ f(x) \leq g(x),$$

$$(f \sqcup g)(x) \doteq f(x) \sqcup g(x),$$

and of a function **0** that maps all elements to 0:

$$\mathbf{0} \doteq \lambda x. 0.$$

A stamp will consist of a pair (i, e): the identity and the event components, both functions from some arbitrary domain to natural numbers. The identity component is a characteristic function (maps elements to $\{0, 1\}$) that defines the set of elements in the domain available to *inflate* ("increment") the event function when an event occurs. We chose to use the characteristic function instead of the set as it leads to better notation. The essential point towards ensuring a correct tracking of causality is to be able to inflate the mapping of some element which no other entity (process or replica) has access to[1]. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities. This is expressed by the following invariant over the identity components of all entities:

$$\forall i.\ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

We adopt a less general but more useful invariant, as it can be maintained by local operations without access to global knowledge. It consists of having disjointness of the parts of the domain that are mapped to 1 in each entity; i.e. non-overlapping graphs for any pair of id functions.

$$\forall i_1 \neq i_2.\ i_1 \cdot i_2 = \mathbf{0}.$$

Comparison of stamps is made through the event component:

$$(i_1, e_1) \leq (i_2, e_2) \doteq e_1 \leq e_2.$$

Join takes two stamps, and returns a stamp that causally dominates both (therefore, the event component is a join of the event components), and has the elements from both identities available for future event accounting:

$$\mathrm{join}((i_1, e_1), (i_2, e_2)) \doteq (i_1 + i_2, e_1 \sqcup e_2).$$

Fork can be any function that takes a stamp and returns two stamps which keep the same event component, but split between them the available elements in the identity; i.e. any function:

$$\mathrm{fork}((i, e)) \doteq ((i_1, e), (i_2, e)) \qquad \text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}.$$

Peek is a special case of fork, which results in one *anonymous* stamp with $\mathbf{0}$ identity and another which keeps all the elements in the identity to itself:

$$\mathrm{peek}((i, e)) \doteq ((i, e), (\mathbf{0}, e)).$$

Event can be any function that takes a stamp and returns another with the same identity and with an event component inflated on any arbitrary set of elements available in the identity:

$$\mathrm{event}((i, e)) = (i, e + f \cdot i) \qquad \text{for any } f \text{ such that } f \cdot i > \mathbf{0}.$$

An event cannot be applied to an anonymous stamp as no element in the domain is available to be inflated.

[1] If this property is not met it can still be possible to form an order that is compatible with causality, but where some concurrent events appear as ordered. This is the case in Lamport clocks [12] and in plausible clocks [23] where the stated invariant does not hold. A Lamport clock can be modeled by having the same identity in all entities.

5 Interval Tree Clocks

We now describe *Interval Tree Clocks*, a novel clock mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralized creation of processes/replicas without need for global identifiers. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. There are two essential differences between ITC and classic clock mechanisms, from the point of view of our function space framework:

- in classic mechanisms each entity uses a fixed, pre-defined function for id; in ITC the id component of entities is manipulated to adapt to the dynamic number of entities;
- classic mechanisms are based on functions over a discrete and typically finite domain; ITC is based on functions over a continuous infinite domain ($\mathbb{R}$) with emphasis on the interval $[0,1)$; this domain can be split into an arbitrary number of subintervals as needed.

The idea is that each entity has available, in the id, a set of intervals that it can use to *inflate* the event component and to give to the successors when forking; a join operation joins the sets of intervals. Each interval results from successive partitions of $[0,1)$ into equal subintervals; the set of intervals is described by a binary tree structure. Another binary tree structure is also used for the event component, but this time to describe a mapping of intervals to integers. To describe the mechanism in terms of functions, it is useful to define a *unit pulse* function[2]:

$$\mathbf{1} \doteq \lambda x. \begin{cases} 1 & x \geq 0 \wedge x < 1, \\ 0 & x < 0 \vee x \geq 1. \end{cases}$$

The id component is an *id tree* with the recursive form (where i, i_1, i_2 range over id trees):

$$i ::= 0 \mid 1 \mid (i_1, i_2).$$

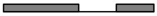
We define a semantic function for the interpretation of id trees as functions:

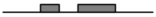
$$\begin{aligned} [\![0]\!] &= \mathbf{0} \\ [\![1]\!] &= \mathbf{1} \\ [\![(i1, i2)]\!] &= \lambda x. [\![i_1]\!](2x) + [\![i_2]\!](2x - 1). \end{aligned}$$

These functions can be 1 for some subintervals of $[0,1)$ and 0 otherwise. For an id (i_1, i_2), the functions corresponding to the two subtrees are transformed so as to be non-zero in two non-overlapping subintervals: i_1 in the interval $[0, 1/2)$ and i_2 in the interval $[1/2, 1)$. As an example, $(1,(0,1))$ represents the function $\lambda x. \mathbf{1}(2x) +$

[2] In this paper we use the *lambda calculus* notation for defining unary functions: a function is anonymously defined by a lambda expression which expresses its action on its argument. For instance, the "increment" function f such that $f(x) = x + 1$ would be expressed as $\lambda x. x + 1$

$(\lambda x.\,\mathbf{1}(2x-1))(2x-1)$. We will also use a graphical notation, which is based on the graph of the function over $[0,1)$. Examples:

$$(1,(0,1)) \sim$$ 

$$((0,(1,0)),(1,0)) \sim$$ 

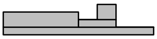
The event component is a binary *event tree* with non-negative integers in nodes; using e, e_1, e_2 to range over event trees and n over non-negative integers:

$$e ::= n \mid (n, e_1, e_2).$$

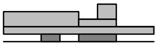
We define a semantic function for the interpretation of these trees as functions:

$$\begin{aligned}
[\![n]\!] &= n \cdot \mathbf{1} \\
[\![(n, e1, e2)]\!] &= n \cdot \mathbf{1} + \lambda x.\,[\![e_1]\!](2x) + [\![e_2]\!](2x-1).
\end{aligned}$$

This means that the value for an element in some subinterval is the sum of a *base* value, common for the whole interval, plus a *relative* value from the corresponding subtree. We will also use a graphical notation for the event component; again, it is based on the graph of the function, obtained by "stacking" the corresponding parts. An example:

$$(1,2,(0,(1,0,2),0)) \sim$$ 

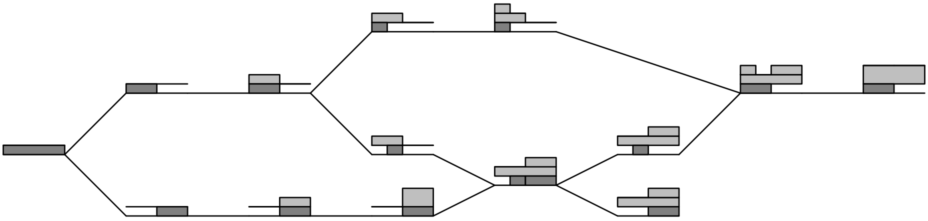
A stamp in ITC is a pair (i,e), where i is an id tree and e an event tree; we will also use a graphical notation based on stacking the two components:

$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \sim$$ 

ITC makes use what we call the *seed* stamp, $(1,0)$, from which we can fork as desired to obtain an initial configuration.

5.1 An Example

We now present an example to illustrate the intuition behind the mechanism, showing a run with a dynamic number of entities in the fork-event-join model. The run starts by a single entity, with the *seed* stamp, which forks into two; one of these suffers one event and forks; the other suffers two events. At this point there are three entities. Then, one entity suffers an event while the remaining two synchronize by doing a join followed by a fork.



The example shows how ITC adapts to the number of entities and allows simplifications to occur upon joins or events. While the first two forks had to split a node in the id tree, the third one makes use of the two available subtrees. The final join leads to a simplification in the id by merging two subtrees. It can be seen that each event always inflates the event tree in intervals available in the id. The event after the final join managed to perform an inflation in a way such that the resulting event function is represented by a single integer.

5.2 Normal Form

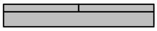
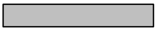
There can be several equivalent representations for a given function. ITC is conceived so as to keep stamps in a *normal form*, for the representations of both id and event functions. This is important not only for having compact representations but also to allow simple definitions of the operations on stamps (fork, event, join) as shown below. As an example, for the unit pulse, we have:

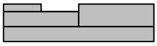
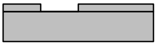
$$\mathbf{1} \sim 1 \equiv (1,1) \equiv (1,(1,1)) \equiv ((1,1),1) \equiv \ldots$$

This means that, if after a join the resulting id is $(1,(1,1))$, we can simplify it to 1. Normalization of the id component can be obtained by applying the following function when building the id tree recursively:

$$\begin{aligned} \mathrm{norm}((0,0)) &= 0, \\ \mathrm{norm}((1,1)) &= 1, \\ \mathrm{norm}(i) &= i. \end{aligned}$$

The event component can be also normalized, preserving its interpretation as a function. Two examples:

$(2,1,1) \sim$  $\equiv$  $\sim 3,$

$(2,(2,1,0),3) \sim$  $\equiv$  $\sim (4,(0,1,0),1).$

To normalize the event component we will make use of the following operators to "lift" or "sink" a tree:

$$\begin{aligned} n{\uparrow}^m &= n+m, \\ (n,e_1,e_2){\uparrow}^m &= (n+m,e_1,e_2), \\ n{\downarrow}_m &= n-m, \\ (n,e_1,e_2){\downarrow}_m &= (n-m,e_1,e_2). \end{aligned}$$

Normalization of the event component can be obtained by applying the following function when building a tree recursively (where m and n range over integers and e_1 and e_2 over normalized event trees) :

$$\begin{aligned} \mathrm{norm}(n) &= n, \\ \mathrm{norm}((n,m,m)) &= n+m, \\ \mathrm{norm}((n,e_1,e_2)) &= (n+m, e_1{\downarrow}_m, e_2{\downarrow}_m),\ \text{where } m = \min(\min(e_1),\min(e_2)), \end{aligned}$$

where $\min$ applied to a tree returns the minimum value of the corresponding function in the range $[0, 1)$:

$$\min(e) = \min_{x \in [0,1)} [\![e]\!](x),$$

which can be obtained by the recursive function over event trees:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n + \min(\min(e_1), \min(e_2)), \end{aligned}$$

or more simply, assuming the event tree is normalized:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n, \end{aligned}$$

which explores the property that in a normalized event tree, one of the subtrees has minimum equal to 0. We will also make use of the analogous $\max$ function over event trees that returns the maximum value of the corresponding function in the range $[0, 1)$, and can be obtained by the recursive function:

$$\begin{aligned} \max(n) &= n, \\ \max((n, e_1, e_2)) &= n + \max(\max(e_1), \max(e_2)). \end{aligned}$$

5.3 Operations over ITC

We now present the operations on ITC for the fork-event-join model. They are defined so as to respect the operations and invariants from the function space based framework presented in the previous section. All the functions below take as input and give as result stamps in the normal form.

Comparison. Comparison of ITC can be derived from the pointwise comparison of the corresponding functions:

$$(i_1, e_1) \leq (i_2, e_2) \doteq [\![e_1]\!] \leq [\![e_2]\!].$$

It is trivial to see that this can be computed through a recursive function over normalized event trees; i.e. $(i_1, e_1) \leq (i_2, e_2) \iff \mathrm{leq}(e_1, e_2)$, with leq defined as (where l and r range over the “left” and “right” subtrees):

$$\begin{aligned} \mathrm{leq}(n_1, n_2) &= n_1 \leq n_2, \\ \mathrm{leq}(n_1, (n_2, l_2, r_2)) &= n_1 \leq n_2, \\ \mathrm{leq}((n_1, l_1, r_1), n_2) &= n_1 \leq n_2 \wedge \mathrm{leq}(l_1{\uparrow}^{n_1}, n_2) \wedge \mathrm{leq}(r_1{\uparrow}^{n_1}, n_2), \\ \mathrm{leq}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= n_1 \leq n_2 \wedge \mathrm{leq}(l_1{\uparrow}^{n_1}, l_2{\uparrow}^{n_2}) \wedge \mathrm{leq}(r_1{\uparrow}^{n_1}, r_2{\uparrow}^{n_2}). \end{aligned}$$

Fork. Forking preserves the event component, and must split the id in two parts whose corresponding functions do not overlap and give the original one when added.

$$\mathrm{fork}(i, e) \doteq ((i_1, e), (i_2, e)), \text{ where } (i_1, i_2) = \mathrm{split}(i),$$

for a function split such that:

$$(i_1, i_2) = \mathrm{split}(i) \implies [\![i_1]\!] \times [\![i_2]\!] = \mathbf{0} \wedge [\![i_1]\!] + [\![i_2]\!] = [\![i]\!].$$

This is satisfied naturally using the following recursive function over id trees, as the two subtrees of an id component always represent functions that do not overlap:

$$\begin{aligned}
\mathrm{split}(0) &= (0, 0),\\
\mathrm{split}(1) &= ((1, 0), (0, 1)),\\
\mathrm{split}((0, i)) &= ((0, i_1), (0, i_2)), \text{ where } (i_1, i_2) = \mathrm{split}(i),\\
\mathrm{split}((i, 0)) &= ((i_1, 0), (i_2, 0)), \text{ where } (i_1, i_2) = \mathrm{split}(i),\\
\mathrm{split}((i_1, i_2)) &= ((i_1, 0), (0, i_2))
\end{aligned}$$

Join. Joining two entities is made by summing the corresponding id functions and making a join of the corresponding event functions:

$$\mathrm{join}((i_1, e_1), (i_2, e_2)) \doteq (\mathrm{sum}(i_1, i_2), \mathrm{join}(e_1, e_2)),$$

for a sum function over identities and a join function over event trees such that:

$$\begin{aligned}
[\![\mathrm{sum}(i_1, i_2)]\!] &= [\![i_1]\!] + [\![i_2]\!],\\
[\![\mathrm{join}(e_1, e_2)]\!] &= [\![e_1]\!] \sqcup [\![e_2]\!].
\end{aligned}$$

The sum function that respects the above condition and also produces a normalized id is:

$$\begin{aligned}
\mathrm{sum}(0, i) &= i,\\
\mathrm{sum}(i, 0) &= i,\\
\mathrm{sum}((l_1, r_1), (l_2, r_2)) &= \mathrm{norm}((\mathrm{sum}(l_1, l_2), \mathrm{sum}(r_1, r_2))).
\end{aligned}$$

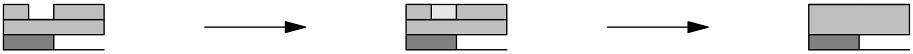
Likewise, the join function over event trees, producing a normalized event tree is:

$$\begin{aligned}
\mathrm{join}(n_1, n_2) &= \max(n_1, n_2),\\
\mathrm{join}(n_1, (n_2, l_2, r_2)) &= \mathrm{join}((n_1, 0, 0), (n_2, l_2, r_2)),\\
\mathrm{join}((n_1, l_1, r_1), n_2) &= \mathrm{join}((n_1, l_1, r_1), (n_2, 0, 0)),\\
\mathrm{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \mathrm{join}((n_2, l_2, r_2), (n_1, l_1, r_1)), \text{ if } n_1 > n_2,\\
\mathrm{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \mathrm{norm}((n_1, \mathrm{join}(l_1, l_2{\uparrow}^{n_2 - n_1}), \mathrm{join}(r_1, r_2{\uparrow}^{n_2 - n_1}))).
\end{aligned}$$

Event. The event operation is substantially more complex than the others. While fork and join have a simple natural definition, event has a larger freedom of implementation while respecting the condition:

$$\mathrm{event}((i, e)) = (i, e'),\ \text{subject to } [\![e']\!] = [\![e]\!] + f \cdot [\![i]\!] \text{ for any } f \text{ such that } f \cdot [\![i]\!] > \mathbf{0}.$$

Event cannot be applied to anonymous stamps; it has the precondition that the id is non-null; i.e. $i \neq 0$. We can use any subset of the available id to inflate the event function. The freedom of which part to inflate is explored in ITC so as to simplify the event tree. Considering the final event in our larger example:



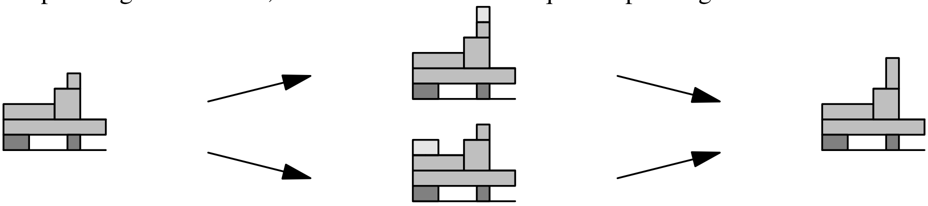
The event operation was able to fill the missing part in a tree so as to allow its simplification to a single integer. In general, the event operation can use several parts of the id, and may simplify several subtrees simultaneously. The operation performs all simplifications in the event tree that are possible given the id tree. If some simplification is possible (which means the corresponding function was inflated), the resulting tree is returned; otherwise another procedure is applied, that "grows" some subtree, preferably only incrementing an integer if possible. The event operation is defined resorting to these two functions (fill and grow) defined below:

$$\mathrm{event}(i, e) = \begin{cases} (i, \mathrm{fill}(i, e)) & \text{if } \mathrm{fill}(i, e) \neq e, \\ (i, e') & \text{otherwise, where } (e', c) = \mathrm{grow}(i, e). \end{cases}$$

Fill either succeeds in doing one or more simplifications, or returns an unmodified tree; it never increments an integer that would not lead to simplifying the tree:

$$\begin{aligned}
\mathrm{fill}(0, e) &= e, \\
\mathrm{fill}(1, e) &= \max(e), \\
\mathrm{fill}(i, n) &= n, \\
\mathrm{fill}((1, i_r), (n, e_l, e_r)) &= \mathrm{norm}((n, \max(\max(e_l), \min(e'_r)), e'_r)), \\
& \text{where } e'_r = \mathrm{fill}(i_r, e_r), \\
\mathrm{fill}((i_l, 1), (n, e_l, e_r)) &= \mathrm{norm}((n, e'_l, \max(\max(e_r), \min(e'_l)))), \\
& \text{where } e'_l = \mathrm{fill}(i_l, e_l), \\
\mathrm{fill}((i_l, i_r), (n, e_l, e_r)) &= \mathrm{norm}((n, \mathrm{fill}(i_l, e_l), \mathrm{fill}(i_r, e_r))).
\end{aligned}$$

In the following example, fill is unable to perform any simplification and grow is used. From the two candidate inflations shown in light grey, the one chosen requires a simple integer increment, while the other would require expanding a node:



Grow performs a dynamic programming based optimization to choose the inflation that can be performed, given the available id tree, so as to minimize the cost of the event tree growth. It is defined recursively, returning the new event tree and cost, so that:

- incrementing an integer is preferable over expanding an integer to a tuple;
- to disambiguate, an operation near the root is preferable to one farther away.

$$\begin{aligned}
\mathrm{grow}(1,n) &= (n+1,0),\\
\mathrm{grow}(i,n) &= (e',c+N),\ \text{where } (e',c)=\mathrm{grow}(i,(n,0,0)),\\
&\quad\ \text{and } N \text{ is some large constant},\\
\mathrm{grow}((0,i_r),(n,e_l,e_r)) &= ((n,e_l,e'_r),c_r+1),\ \text{where } (e'_r,c_r)=\mathrm{grow}(i_r,e_r),\\
\mathrm{grow}((i_l,0),(n,e_l,e_r)) &= ((n,e'_l,e_r),c_l+1),\ \text{where } (e'_l,c_l)=\mathrm{grow}(i_l,e_l),\\
\mathrm{grow}((i_l,i_r),(n,e_l,e_r)) &= \begin{cases}((n,e'_l,e_r),c_l+1) & \text{if } c_l<c_r,\\ ((n,e_l,e'_r),c_r+1) & \text{if } c_l\geq c_r,\end{cases}\\
&\quad\ \text{where } (e'_l,c_l)=\mathrm{grow}(i_l,e_l) \text{ and } (e'_r,c_r)=\mathrm{grow}(i_r,e_r).
\end{aligned}$$

The definition makes use of a constant N that should be greater than the maximum tree depth that arises. This is a practical choice, to have the cost as a simple integer. We could avoid it by having the cost as a pair under lexicographic order, but it would "pollute" the presentation and be a distracting element.

6 Exercising ITCs

In order to have a rough insight of ITC space consumption we exercised its usage for both dynamic and static scenarios, using a mix of data and process causality. For data causality in dynamic scenarios, each iteration consists of forking, recording an event and joining two replicas, each performed on random replicas, leading to constantly evolving ids. This pattern maintains the number of existing replicas while exercising id management under churn. For process causality in a static scenario, we operate on a fixed set of processes doing message exchanges (via peek and join) and recording internal events; here ids remain unchanged, since messages are anonymous.

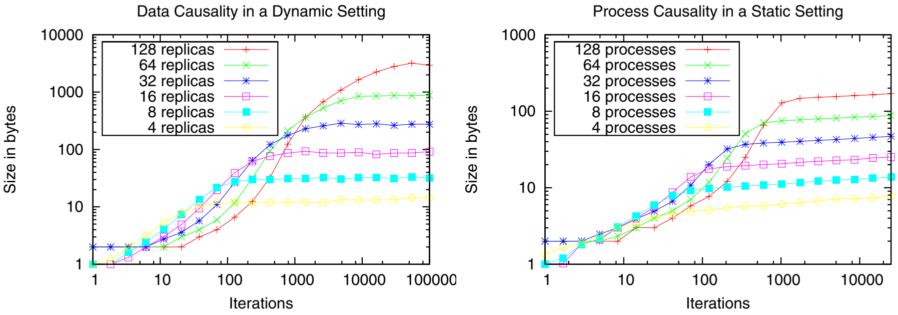


Fig. 3. Average space consumption of an ITC stamp, in dynamic and static settings

The charts in Figure 3 depict the mean size (using the binary encoding shown in Appendix A) of an ITC across 100 runs of 25,000 iterations for process causality and 100,000 iterations for data causality and for different numbers of active entities (pre-created by forking a *seed* stamp before iterating). It shows that space consumption basically stabilizes after a number of iterations. These results show that ITCs can in fact be used as a practical mechanism for data and process causality in dynamic systems, contrary to Version Stamps [2] that have storage cost growing unreasonably over time.

In order to put these numbers in perspective, the Microsoft Windows operating system [19] uses 128 bits Universally Unique Identifiers (UUIDs) and 32 bit counters. The storage cost of a version vector for 128 replicas would be 2560 bytes using a mapping from ids to counters and 512 bytes using a vector. The mean size of an ITC for this scenario (at the end of the iterations) would be less than 2900 bytes for dynamic scenarios and slightly above 170 bytes for static ones. While vectors can be represented in a more compact way (e.g. factoring out the smallest number), such optimizations would be irrelevant for dynamic scenarios, where most of the cost stems from the UUIDs.

7 Conclusions

We have introduced Interval Tree Clocks, a novel logical clock mechanism for dynamic systems, where processes/replicas can be created or retired in a decentralized fashion. The mechanism has been presented using a model (fork-event-join) that can serve as a kernel to describe all classic operations (like message sending, symmetric synchronization and process creation/retirement), being suitable for both process and data causality scenarios.

We have presented a general framework for clock mechanisms, where stamps can be seen as finite representations of a pair of functions over a continuous domain; the event component serves to perform comparison or join (performed pointwise); the identity component defines a set of intervals where the event component can be inflated (a generalization of the classic counter increment). ITC is a concrete mechanism that instantiates the framework, using trees to describe functions on sets of intervals. The framework opens the way for research on future alternative mechanisms that use different representations of functions.

Previous approaches to causality tracking for dynamic systems either require access to globally unique ids; do not reuse ids of retired entities; require global coordination for garbage collection of ids; or exhibit an intolerable growth in terms of space consumption (our previous approach). ITC is the first mechanism for dynamic systems that avoids all these problems, can be used for both process and data causality, and requires a modest space consumption, making it a general purpose mechanism, even for static systems.

References

1. Almeida, J.B., Almeida, P.S., Baquero, C.: Bounded version vectors. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 102–116. Springer, Heidelberg (2004)
2. Almeida, P.S., Baquero, C., Fonte, V.: Version stamps – decentralized version vectors. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), pp. 544–551. IEEE Computer Society, Los Alamitos (2002)

3. Almeida, P.S., Baquero, C., Fonte, V.: Improving on version stamps. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM-WS 2007, Part II. LNCS, vol. 4806, pp. 1025–1031. Springer, Heidelberg (2007)
4. Baldoni, R., Raynal, M.: Fundamentals of distributed computing: A practical tour of vector clock systems. IEEE Distributed Systems Online 3(2) (2002)
5. Charron-Bost, B.: Concerning the size of logical clocks in distributed systems. Information Processing Letters 39, 11–16 (1991)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Bressoud, T.C., Kaashoek, M.F. (eds.) Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pp. 205–220. ACM, New York (2007)
7. Fidge, C.: Timestamps in message-passing systems that preserve the partial ordering. In: 11th Australian Computer Science Conference, pp. 55–66 (1989)
8. Fidge, C.: Logical time in distributed computing systems. IEEE Computer 24(8), 28–33 (1991)
9. Golden III., R.G.: Efficient vector time with dynamic process creation and termination. Journal of Parallel and Distributed Computing (JPDC) 55(1), 109–120 (1998)
10. Greenwald, M.B., Khanna, S., Kunal, K., Pierce, B.C., Schmitt, A.: Agreeing to agree: Conflict resolution for optimistically replicated data. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 269–283. Springer, Heidelberg (2006)
11. Huang, S.-T.: Detecting termination of distributed computations by external agents. In: International Conference on Distributed Computing Systems, Newport Beach, California, pp. 79–84 (1989)
12. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
13. Landes, T.: Tree clocks: An efficient and entirely dynamic logical time system. In: Burkhart, H. (ed.) Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, as part of the 25th IASTED International Multi-Conference on Applied Informatics, Innsbruck, Austria, February 13-15, 2007, pp. 349–354. IASTED/ACTA Press (2007)
14. Malkhi, D., Terry, D.B.: Concise version vectors in winfs. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 339–353. Springer, Heidelberg (2005)
15. Mattern, F.: Global quiescence detection based on credit distribution and recovery. IPL: Information Processing Letters 30 (1989)
16. Mattern, F.: Virtual time and global clocks in distributed systems. In: Workshop on Parallel and Distributed Algorithms, pp. 215–226 (1989)
17. Mostefaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., El Abbadi, A.: From static distributed systems to dynamic systems. In: Proceedings 24th IEEE Symposium on Reliable Distributed Systems (24th SRDS 2005), Orlando, FL, USA, pp. 109–118. IEEE Computer Society, Los Alamitos (2005)
18. Parker, D.S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. Transactions on Software Engineering 9(3), 240–246 (1983)
19. Peek, D., Terry, D.B., Ramasubramanian, V., Walraed-Sullivan, M., Rodeheffer, T.L., Wobber, T.: Fast encounter-based synchronization for mobile devices. In: 2nd International Conference on Digital Information Management, 2007. ICDIM 2007, vol. 2, pp. 750–755 (2007)
20. Petersen, K., Spreitzer, M., Terry, D., Theimer, M.: Bayou: Replicated database services for world-wide applications. In: 7^{th} ACM SIGOPS European Workshop, Connemara, Ireland (1996)

21. Ratner, D., Reiher, P., Popek, G.: Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles (1997)
22. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed Computing 3(7), 149–174 (1994)
23. Torres-Rojas, F.J., Ahamad, M.: Plausible clocks: constant size logical clocks for distributed systems. Distributed Computing 12(4), 179–196 (1999)

A A Binary Encoding for ITC

Here we describe a compact encoding of ITC as strings of bits. It may be relevant when stamp size is an issue, e.g. when many entities are involved; it is appropriate to being transmitted or stored persistently as a single *blob*. We do not attempt to present an optimal (in some way) encoding, but a sensible one, which was used in the space consumption analysis.

As an event tree tends to have very few large numbers near the root and many very small numbers at the leaves; this prompts a variable length representation for integers, where small integers occupy just a few bits. Also, common cases like trees with only the left or right subtree, or with 0 for the base value are treated as special cases.

We use a notation (inspired by the bit syntax from the Erlang programming language) where: $\ll x, y, z \gg$ is a string of bits resulting from concatenating x, y and z; and $n{:}b$ represents number n encoded in b bits. An example: $\ll 2{:}3, 0{:}1, 1{:}2 \gg$ represents the string of 6 bits 010001.

$$enc((i,e)) = \ll enc_i(i), enc_e(e) \gg.$$

$$\begin{aligned}
enc_i(0) &= \ll 0{:}2, 0{:}1 \gg,\\
enc_i(1) &= \ll 0{:}2, 1{:}1 \gg,\\
enc_i((0,i)) &= \ll 1{:}2, enc_i(i) \gg,\\
enc_i((i,0)) &= \ll 2{:}2, enc_i(i) \gg,\\
enc_i((i_l,i_r)) &= \ll 3{:}2, enc_i(i_l), enc_i(i_r) \gg.
\end{aligned}$$

$$\begin{aligned}
enc_e((0,0,e_r)) &= \ll 0{:}1, 0{:}2, enc_e(e_r) \gg,\\
enc_e((0,e_l,0)) &= \ll 0{:}1, 1{:}2, enc_e(e_l) \gg,\\
enc_e((0,e_l,e_r)) &= \ll 0{:}1, 2{:}2, enc_e(e_l), enc_e(e_r) \gg,\\
enc_e((n,0,e_r)) &= \ll 0{:}1, 3{:}2, 0{:}1, 0{:}1, enc_e(n), enc_e(e_r) \gg,\\
enc_e((n,e_l,0)) &= \ll 0{:}1, 3{:}2, 0{:}1, 1{:}1, enc_e(n), enc_e(e_l) \gg,\\
enc_e((n,e_l,e_r)) &= \ll 0{:}1, 3{:}2, 1{:}1, enc_e(n), enc_e(e_l), enc_e(e_r) \gg,\\
enc_e(n) &= \ll 1{:}1, enc_n(n,2) \gg.
\end{aligned}$$

$$enc_n(n,B) = \begin{cases} \ll 0{:}1, n{:}B \gg & \text{if } n < 2^B,\\ \ll 1{:}1, enc_n(n-2^B, B+1) \gg & \text{otherwise.} \end{cases}$$

Ordering-Based Semantics for Software Transactional Memory*

Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott

Department of Computer Science, University of Rochester, Rochester, NY 14627, USA
{spear,luked,vmarathe,scott}@cs.rochester.edu

Abstract. It has been widely suggested that memory transactions should behave as if they acquired and released a single global lock. Unfortunately, this behavior can be expensive to achieve, particularly when—as in the natural *publication/privatization* idiom—the same data are accessed both transactionally and nontransactionally. To avoid overhead, we propose *selective strict serializability* (SSS) semantics, in which transactions have a global total order, but nontransactional accesses are globally ordered only with respect to explicitly marked transactions. Our definition of SSS formally characterizes the permissible behaviors of an STM system without recourse to locks. If all transactions are marked, then SSS, single-lock semantics, and database-style strict serializability are equivalent.

We evaluate several SSS implementations in the context of a TL2-like STM system. We also evaluate a weaker model, *selective flow serializability* (SFS), which is similar in motivation to the *asymmetric lock atomicity* (ALA) of Menon et al. We argue that ordering-based semantics are conceptually preferable to lock-based semantics, and just as efficient.

1 Introduction

With the proliferation of multicore processors, there is widespread recognition that traditional lock-based synchronization is too complex for "mainstream" parallel programming. *Transactional memory* attempts to address this complexity by borrowing the highly successful notion of transactions from database systems.

Transactions constrain the ways in which thread histories may legally appear to interleave. If all shared data were accessed only within transactions, the constraints would be relatively simple. In a break from the database world, however, memory transactions are generally expected to coexist with nontransactional memory accesses, some of which may also access shared memory. This coexistence complicates the task of specifying, for every read in the program, which values may be returned.

Many possible semantics for TM have been proposed, including *strong and weak isolation* (also known as strong and weak atomicity) [2, 18], *single lock atomicity* (SLA) [8], and approaches based on language memory models [5], linearizability [16, 6], and operational semantics [1, 13]. Of these, SLA has received the most attention. It specifies that transactions behave as if they acquired a single global mutual exclusion lock.

* This work was supported in part by the US National Science Foundation under grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 275–294, 2008.

Unfortunately, as several groups have noted [8,12,19], SLA requires behavior that can be expensive to enforce, particularly when a thread *privatizes* shared data (rendering it logically inaccessible to other threads), works on it for a while (ideally without incurring transactional overheads), and then *publishes* it again [11,12,19].

Most software TM (STM) implementations today do not correctly accommodate privatization and publication, and forcing them to do so—at the boundaries of every transaction—would impose significant costs. In an attempt to reduce those costs, Menon et al. [12] have proposed a series of semantics that relax the requirement for serialization among transactions. These semantics are described in terms of locking protocols significantly more complex than SLA. While we appreciate the motivation, we argue, with Luchangco [9], that locks are the wrong way to formalize transactions. We prefer the more abstract approach of language-level memory models [5,10,3]; like traditional formalizations of database semantics, these directly specify permissible access orderings. We also argue that if one wishes to reduce the cost of SLA, it makes more sense to relax the ordering between nontransactional and transactional accesses *within a single thread*, rather than the ordering between transactions.

We argue that SLA is equivalent to the traditional database ordering condition known as *strict serializability* (SS). As a candidate semantics for STM, we suggest transactions with *selective strict serializability* (SSS), in which nontransactional accesses are ordered with respect to a subset of their thread's transactions. Whether this subset is all, none, or some explicitly or implicitly identified set in between, a single formal framework suffices to explain program behavior. We also propose a slightly weaker semantics, *selective flow serializability* (SFS), that orders nontransactional accesses with respect to subsequent transactions in other threads only in the presence of a forward dataflow dependence that could constitute publication.

Like Menon et al., we take the position that races between transactional and nontransactional code are program bugs, but that (as in Java) the behavior of buggy programs should be constrained to avoid "out of thin air" reads. SSS and SFS allow the programmer or compiler to eliminate races by labeling a minimal set of transactions, while still constraining behavior if the labeling is incorrect.

After a more detailed discussion of background in Section 2, we formalize our ordering-based semantics in Section 3. To the best of our knowledge, this is the first attempt to formalize privatization and publication safety without recourse to locks. We discuss implementation options in Section 4, in the context of an STM system patterned on TL2 [4]. We compare the performance of these implementations experimentally in Section 5, and conclude in Section 6.

2 Background

Arguably the most intuitive semantics for transactions would build on sequential consistency, providing a global total order, consistent with program order, for both transactions and nontransactional accesses. Unfortunately, most architects already consider sequential consistency too expensive to implement, even without transactions. Among other things, it precludes standard compiler optimizations like write reordering. In light of this expense, sequentially consistent transactions appear to be a non-starter.

Several researchers have argued instead for what Blundell et al. [2] call *strong atomicity*, otherwise known as *strong isolation*. This drops the requirement that a thread's nontransactional memory accesses be seen by other threads in program order. It still requires, however, that nontransactional accesses serialize with respect to transactions; that is, that each nontransactional reference appears to happen *between* transactions, and that all threads agree as to which transactions it appears between.

Unfortunately, strong isolation still suffers from several significant costs, which make it unappealing, at least for software implementation:

Granularity: In a high-level programming language, it is not immediately clear what constitutes an individual memory access. Is it acceptable, for example, for a transaction to intervene between the read and write that underlie `x++` on a load-store machine? How about `x` = 0x300000003, where `x` is a `long long` variable but hardware does not provide an atomic 64-bit store?

Instrumentation: It is generally agreed that speculative execution of software transactions in the face of concurrent nontransactional accesses will require the latter to inspect and (in the case of writes) modify transactional metadata. Particularly in programs that make extensive nontransactional use of data that are temporarily private, this instrumentation can have a major performance impact.

Optimization obstruction: Nontransactional accesses cannot safely be reordered if they refer to locations that may also be accessed by transactions. The sequence `x = 1; y = 2` admits the possibility that a concurrent transaction will see `x == 1 && y != 2`. If the compiler is unable to prove that no such transaction (or series of transactions) can exist, it must, (a) treat `x` and `y` as volatile variables, access them in program order, and insert a write-write memory barrier between them; or (b) arrange for the assignments to execute as a single atomic unit.[1]

In light of these costs, most STM systems provide some form of *weak isolation*, in which nontransactional accesses do not serialize with transactions. As several groups have noted [5,12,13,19], the exact meaning of weak isolation is open to interpretation. Perhaps the most popular interpretation is *single lock atomicity* (SLA) [8, p. 20], which states that transactions behave as if they held a global mutex lock. Unfortunately, even these semantics have nontrivial cost, and are unsupported by most TM implementations, particularly for programs that publish or privatize data.

Publication (Figure 1, left) occurs when a thread initializes or otherwise modifies a data structure that is logically private, and then modifies shared state to make the structure accessible to other threads. *Privatization* (right) is the reverse: a thread's modification of shared state makes some structure logically private. The appeal of privatization and publication is the possibility that temporarily private data might be accessed without transactional overhead.

[1] Note that while strong isolation is sometimes equated with making every nontransactional access execute as if it were an isolated transaction [2], this characterization is problematic: it would force a global total order not only between these isolated transactions and "real" programmer-specified transactions, but *among* the isolated transactions. The end result would be equivalent to sequential consistency for shared locations.

```
// initialize node                    atomic {
atomic {                                node = PQ.extract_min()
  PQ.insert(node)                     }
}                                     // use node privately
```

Fig. 1. Examples of publication (left) and privatization (right). More obscure examples, involving antidependences or even empty transactions, are also possible [12].

Traditionally, when a thread releases a mutual exclusion lock, all prior accesses by the thread are guaranteed to have occurred from the point of view of every other thread. Similarly, when a thread acquires a lock, all subsequent accesses by the thread are guaranteed not to have occurred. These facts suggest that the natural implementation of SLA would be *publication-* and *privatization-safe*.

The Privatization Problem. In previous work [19], we identified two dimensions of the privatization problem. Both arise from the fact that STM systems may perform operations that logically precede, but physically follow, a transaction's linearization point. If nontransactional code is unaware of the behavior of the TM system, transactional and nontransactional work may overlap.

In the *delayed cleanup problem* (also described by Larus and Rajwar [8, pp. 22–23]), transactional writes may appear to occur too late from the point of view of nontransactional code. Specifically, a thread that privatizes shared data may fail to see logically prior updates by a transaction that has committed but has not yet written its "redo log" back to main memory.[2] In the *doomed transaction problem* (also described by Wang et al. [20, pp. 6–7]), private writes may appear to occur too early from a doomed transaction's point of view. The resulting inconsistent view of memory may then allow that transaction to fall into an infinite loop, suffer an exception, or (in the absence of runtime sandboxing) perform erroneous actions that cannot be undone.

The Publication Problem. One might not initially expect a publication problem: private accesses are assumed to occur before publication, and there is no notion of cleanup for nontransactional code. Menon et al. show, however, that problems can arise if the programmer or compiler prefetches data before it is actually published (Figure 2).

Relaxing SLA. Under SLA, straightforward solutions to the privatization and publication problems [19, 12] require transactions to begin and clean up in serialization order. While heuristic optimizations may relax this requirement in some cases [11], it seems clear that the general case will remain expensive for STM. To reduce this expense, Menon et al. propose to relax the ordering among transactions. Of their three candidate semantics, *asymmetric lock atomicity* (ALA) seems most likely to reduce transactional overhead without precluding standard compiler optimizations. ALA transactions behave as if (1) there is a separate reader-writer lock for every datum, (2) read locks are acquired (presciently) at the beginning of the transaction, and (3) write locks are acquired immediately prior to writing. The asymmetry of reads and writes reflects the fact

[2] Conversely, in an STM system based on undo logs, a privatizing thread may see erroneous updates made (temporarily) by a transaction that has aborted but not yet cleaned up. As observed by Menon et al., such reads appear to be fundamentally incompatible with the prohibition against "out of thin air" reads. We therefore assume a redo log in the remainder of this paper.

```
// initially x == 0 and x_is_public == false
T1:                          T2:                          T3:
                                                          e: j = 0
                                                          F: atomic {
                                                              t = prefetch(x)
a: x = 1
B: atomic {                  c: i = 0
    x_is_public = true       D: atomic {
}                                if (x_is_public) {           if (x_is_public) {
                                     i = x                        j = t
                                 }                            }
                             }                            }
```

Fig. 2. Publication (left) in parallel with a safe (middle) or unsafe (right) use of published data. The programmer has made `B` a transaction to ensure that it orders, globally, after prior initialization of `x`. Vertical spacing is meant to suggest a possible interleaving of operations across threads. Both `D` and `F` will serialize after `B`.

that (a) in most TM systems it is much easier for a reader to detect a conflict with a previous writer than vice versa, and (b) in most programs publication can be assumed to require a write in one transaction followed by a read in another.

In our view, ALA and similar proposals suffer from three important problems. First, they explain transaction behavior in terms of a nontrivial fine-grain locking protocol—something that transactions were intended to eliminate! Second, they give up one of the key contributors to the success of database transactions—namely serializability (see for example Fig. 11 of Menon et al. [12]). Third, they impose significant overheads on transactions that do serialize, even in the absence of publication and privatization.

3 Ordering-Based TM Semantics

Our proposed alternative to lock-based semantics begins with a programming model in which, in every execution history H, each thread i has a memory access history $H^i \subset H$ in which certain maximal contiguous strings of accesses are identified as (outermost) transactions. We use $\mathcal{T}_H$ to denote the set of all transactions. We do not consider open nesting here, nor do we permit overlapping but non-nested transactions. Moreover, from the programmer's point of view, transactions are simply atomic: there is no notion of speculation or of committing and aborting. A typical implementation will need to ensure that abortive attempts to execute a transaction are invisible; among other things, this will require that such attempts retain a consistent view of memory [6].

The goal of a semantics for TM is to constrain the ways in which thread histories may legally interleave to create a global history. In keeping with the database literature and with memory models for programming languages like Java [10] and C++ [3], we believe that the appropriate way to specify these constraints is not by reduction to locks, but rather by specification of a partial order on program operations that restricts the set of writes that may be "seen" by a given read.

3.1 Strict Serializability

The standard database ordering criterion is *serializability* [14], which requires that the result of executing a set of transactions be equivalent to (contain the same operations and results as) some execution in which the transactions take place one at a time, and any transactions executed by the same thread take place in program order. *Strict serializability* imposes the additional requirement that if transaction A completes before B starts in the actual execution, then A must occur before B in the equivalent serial execution. The intent of this definition is that if external (nontransactional) operations allow one to tell that A precedes B, then A must serialize before B. For transactional memory, it seems reasonable to equate external operations with nontransactional memory accesses, and to insist that such accesses occur between the transactions of their respective threads, in program order.

More formally, we define the following strict (asymmetric, irreflexive) ordering relations:

Program order, $<_p$, is a union of disjoint total orders, one per thread. We say $a <_p b$ iff a and b are executed by the same thread, and a comes before b in the natural sequential order of the language in which the program is written. Because transactions do not overlap, if transactions A and B are executed by the same thread, we necessarily have either $\forall a \in A, b \in B : a <_p b$ or $\forall a \in A, b \in B : b <_p a$. For convenience, we will sometimes say $A <_p B$ or $B <_p A$. For $a \notin B$, we may even say $a <_p B$ or $B <_p a$.

Transaction order, $<_t$, is a total order on all transactions, across all threads. It is consistent with program order. That is, $A <_p B \implies A <_t B$. For convenience, if $a \in A$, $b \in B$, and $A <_t B$, we will sometimes say $a <_t b$.

Strict serial order, $<_{ss}$, is a partial order on memory accesses. It is consistent with transaction order. It also orders nontransactional accesses with respect to preceding and following transactions of the same thread. Formally, for all accesses a and c in an execution history H, we say $a <_{ss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\, A \in \mathcal{T}_H$: $(a \in A \wedge A <_p c)$; (3) $\exists\, C \in \mathcal{T}_H$: $(a <_p C \wedge c \in C)$; (4) $\exists$ access $b \in H : a <_{ss} b <_{ss} c$. Note that this definition does *not* relate accesses performed by a given thread between transactions.

An execution with program order $<_p$ is said to be strictly serializable if there exists a transaction order $<_t$ that together with $<_p$ induces a strict serial order $<_{ss}$ that permits all the values returned by reads in the execution, as defined in the following subsection. A TM implementation is said to be strictly serializable if all of its executions are strictly serializable.

3.2 Values Read

To avoid the need for special cases, we assume that each thread history H^i begins with an initial transaction T_0^i, that T_0^0 writes values to all statically initialized data, and that for all $i > 0, T_0^0 <_t T_0^i$.

We say a memory access b *intervenes* between a and c if $a <_p b <_p c$ or $a <_{ss} b <_{ss} c$. Read r is then permitted to return the value written by write w only if r and w access the same location l and either (1) r and w are incomparable under both program and strict serial order or (2) $w <_p r \vee w <_{ss} r$ and there is no intervening write of l between w and r.

For the sake of generality across languages and machines, we have kept these rules deliberately parsimonious. In a practical language definition, we would expect them to be augmented with additional rules to capture such concepts as coherence and causality [10]. For example, in

```
initially x == 0
T1: x = 1  ||  T2: atomic{a = x}; atomic{b = x}
```

the language memory model will probably insist that if `a == 1` when `T2` completes, then `b != 0`; that is, `x` is coherent. Likewise, in

```
initially x == y == 0
T1: x = 1  ||  T2: if (x == 1) y = 1  ||  T3: atomic{a = y};
                                              atomic{b = x}
```

the language memory model will probably insist that if `a == 1` when `T3` completes, then `b == 1` also; that is, the system is causally consistent. Deciding on a complete set of such rules is a Herculean and potentially controversial task (witness the memory models for Java and C++); we do not attempt it here. For any given choice of underlying model, however, we argue that strict serializability, as defined above, is equivalent to SLA. If an execution is strictly serializable, then it is equivalent by definition to some execution in which transactions occur in a serial order ($<_t$) consistent with program order and with nontransactional operations. This serial order is trivially equivalent to an execution in which transactions acquire and release a global lock. Conversely, if transactions acquire and release a global lock, they are guaranteed to execute one at a time, in an order consistent with program order. Moreover given a common underlying memory model, SLA and strict serial order will impose identical constraints on the ordering of nontransactional accesses relative to transactions.

3.3 Selective Strictness

A program that accesses shared data only within transactions can be considered properly synchronized, and can safely run on any TM system that respects $<_t$. A program P that sometimes accesses shared data outside transactions, but that is nonetheless data-race-free with respect to $<_{ss}$ (this is what Abadi et al. term *violation-free* [1]) can also be considered properly synchronized, and can safely run on any TM system S that respects $<_{ss}$. Transactions in P that begin and end, respectively, a region of data-race-free nontransactional use are referred to as privatization and publication operations, and S is said to be *publication and privatization safe* with respect to $<_{ss}$.

Unfortunately, most existing TM implementations are not publication and privatization safe with respect to strict serializability, and modifying them to be so would incur nontrivial costs. It is not yet clear whether these costs will be considered an acceptable price to pay for simple semantics. It therefore seems prudent to consider weaker semantics with cheaper implementations. Menon et al. [12] approach this task by defining more complex locking protocols that relax the serialization of transactions. In contrast, we propose a weaker ordering, *selective strict serializability*, that retains the serialization of transactions, but relaxes the ordering of nontransactional accesses with respect to transactions. Specifically, we assume a set of *acquiring* (privatizing) transactions

$\mathcal{A}_H \subseteq \mathcal{T}_H$ and a set of *releasing* (publishing) transactions $\mathcal{R}_H \subseteq \mathcal{T}_H$. $\mathcal{A}_H$ and $\mathcal{R}_H$ are not required to be disjoint, nor are they necessarily proper subsets of $\mathcal{T}_H$. We require that the initial transaction T_0^i be acquiring for all i; aside from this, it is permissible for all transactions, or none, to be identified as acquiring and/or releasing.

Selective strict serial order, $<_{sss}$, is a partial order on memory accesses. Like strict serial order, it is consistent with transaction order. Unlike strict serial order, it orders nontransactional accesses only with respect to preceding acquiring transactions and subsequent releasing transactions of the same thread (and, transitively, transactions with which those are ordered). Formally, for all accesses $a, c \in H$, we say $a <_{sss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\, A \in \mathcal{A}_H : (a \in A \ \wedge\ A <_p c)$; (3) $\exists\, C \in \mathcal{R}_H : (a <_p C \ \wedge\ c \in C)$; (4) $\exists$ access $b \in H : a <_{sss} b <_{sss} c$.

3.4 Asymmetric Flow

Strict serializability, whether "always on" or selective, shares a problem with the DLA (disjoint lock atomicity) semantics of Menon et al. [12]: it requires the useless but expensive guarantee illustrated in Figure 3. Specifically, $a <_p B <_t F \implies a <_{ss} F$, even if B and F are ordered only by antidependence.

```
// initially x == 0, x_is_public == false, and T3_used_x == false
T1:                          T2:                            T3:
                                                            e: j = 0
                                                            F: atomic {
                                                                t = prefetch(x)
a: x = 1
B: atomic {                  c: i = 0
    x_is_public = true       D: atomic {
    k = T3_used_x                if (x_is_public) {
}                                    i = x                      T3_used_x = true
                                 }                              j = t
                             }                              }
```

Fig. 3. "Publication" by antidependence (adapted from Menon et al. [12]). If B is a releasing transaction, selective strict serializability guarantees that the write of x is visible to D (i == 1, which makes sense). In addition, if k == false, then B must have serialized before F, and thus j must equal 1 as well. Unfortunately, it is difficult for an STM implementation to notice a conflict between B and F if the former commits before the latter writes T3_used_x, and undesirable to outlaw the prefetch of x.

We can permit a cheaper implementation if we define a weaker ordering, *selective flow serializability*, that requires nontransactional-to-transactional ordering only when transactions are related by a true (flow) dependence:

Flow order, $<_f \subset <_t$, is a partial order on transactions. We say that $A <_f C$ if there exists a transaction B and a location l such that $A <_t B$, A writes l, B reads l, and $B = C \ \vee\ B <_t C$.

Selective flow serial order, $<_{sfs} \subset <_{sss}$, is a partial order on memory accesses. It is consistent with transaction order. It does not order nontransactional accesses with respect to a subsequent releasing transaction B, but rather with respect to transactions that have a flow dependence on B. Formally, $\forall$ accesses $a, c \in H$, we say $a <_{sfs} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\, A \in \mathcal{A} : (a \in A \;\wedge\; A <_p c)$; (3) $\exists\, B \in \mathcal{R}, C \in \mathcal{T} : (a <_p B <_f C \;\wedge\; c \in C)$; (4) $\exists$ access $b \in H : a <_{sfs} b <_{sfs} c$. It is not difficult to see that $<_{sfs} \subset <_{sss}$.

The sets of values that reads are permitted to return under selective strict and flow serial orders are defined the same as under strict serial order, but with $<_{sss}$ and $<_{sfs}$, respectively, substituted for $<_{ss}$. These induce corresponding definitions of SSS and SFS executions and TM implementations. In comparison to ALA, SFS is not defined in terms of locks, and does not force ordering between nontransactional accesses and unrelated transactions (Figure 4).

```
// initially x == 0, y == 0, and x_is_public == false
T1:                          T2:                          T3:
                                                          e: j = 0
                                                          F: atomic {
                             c: i = 0                         ...
                             D: atomic {
                                 t = prefetch(x)
a: x = 1
B: atomic {
    x_is_public = true
}                                if (x_is_public) {
                                     i = y = t
                                 }
                             }                                j = y
                                                          }
```

Fig. 4. Unnecessary ordering in ALA. When `B` commits and `D` reads `x_is_public`, ALA forces `D` to abort (as it should, since `x_is_public` has to be logically acquired as of the beginning of `D`, and that is impossible once `B` has committed). When `D` commits and `F` reads `y`, ALA will likewise force `F` to abort (since it is flow dependent on a committed transaction with a later start time), though one could, in principle, simply serialize `F` after `D`. With SFS, the programmer would presumably mark `B` but not `D` as a releasing transaction, and `F` would not have to abort.

Like ALA, SFS can lead to apparent temporal loops in racy programs. In Figure 3, for example, both semantics allow `k == false` and `j == 0`, even if `B` is a releasing transaction. Naively, this output would seem to suggest that $a < B < F < a$. ALA breaks the loop by saying that B and F are not really ordered. SFS breaks the loop by saying that a and F are not really ordered. Which of these is preferable is perhaps a matter of taste.

It should be emphasized that private (nontransactional) use of data that are sometimes accessed by other threads is safe only when data-race-free. To be confident that a program is correct, the programmer must identify the transactions that serve to eliminate races. This may sometimes be a difficult task, but it is the inherent and unavoidable cost of privatization, even under SLA. Once it has been paid, the extra effort required to label acquiring and releasing transactions is essentially trivial.

4 Implementing SSS and SFS Systems

Both the doomed transaction problem and the undo log variant of the delayed cleanup problem (footnote 2) involve abortive attempts to execute transactions. Since these attempts play no role in the (user-level) semantics of Section 3, we need to extend our formalism. For discussion of implementation-level issues, we extend thread histories (as in our previous work [16]) to include begin, commit, and abort operations. None of these takes an argument. Begin and abort return no value. Commit returns a Boolean indication of success. Each thread history is assumed to be of the form ((read | write)* begin (read | write)* (commit | abort))* (read | write)* (for simplicity, we assume that nested transactions are subsumed into their parent). A transaction comprises the sequence of operations from begin through the first subsequent commit or abort in program order. A transaction is said to *succeed* iff it ends with a commit that returns true.

With this extended definition, for $<_g \in \{<_{ss}, <_{sss}, <_{sfs}\}$, a read r is permitted to return the value written by a write w iff they access the same location l and (1) w does not belong to an unsuccessful transaction, and r and w are incomparable under both $<_p$ and $<_g$; (2) w does not belong to an unsuccessful transaction, $w <_p r$ or $w <_g r$, and there is no intervening write of l between w and r; or (3) w and r belong to the same transaction, $w <_p r$, and there is no intervening write of l between w and r. A memory access b intervenes between a and c if $a <_p b <_p c$ or $a <_g b <_g c$ and (a) b and c belong to the same transaction, or (b) neither a nor b belongs to an unsuccessful transaction. These rules are roughly equivalent to those of Guerraoui and Kapałka [6], but simplified to merge request and reply events and to assume that histories are complete (i.e., that every transaction eventually commits or aborts), and extended to accommodate nontransactional accesses. In particular, we maintain their requirement that transactions appear to occur in serial order ($<_t$), and that writes in unsuccessful transactions are never externally visible.

We assume that every correct STM implementation suggests, for every execution, a (partial or total) *natural order* $<_n$ on transactions that is consistent with some $<_t$ that (together with $<_p$) can explain the execution's reads. For the implementation to ensure SSS semantics, it must provide publication and privatization safety only around selected releasing and acquiring transactions, respectively. To ensure SFS semantics, it must provide the same privatization safety, but need only provide publication safety beyond selected flow-ordered transactions.

4.1 Preventing the Doomed Transaction Problem

If transactions D and A conflict, and A commits, STM runtimes typically do not immediately interrupt D's execution. Instead, D is responsible for detecting the conflict, rolling itself back, and restarting. This may occur as early as D's next transactional read or write, or as late as D's commit point. Until the conflict is detected, D is said to execute in a "doomed" state. If A privatizes a region accessed by D, subsequent writes to that region may race with concurrent transactional accesses by D.

The doomed transaction problem occurs when an STM implementation admits an execution containing a failed transaction D, a nontransactional write w, an acquiring transaction $A <_p w$, and a natural transaction order $<_n$ such that any $<_t$ consistent with $<_n$, when combined with $<_p$, induces a global (SS, SSS, SFS) order $<_g$ that fails to explain a value read in D—specifically, when there are dependences that force

$D <_t A$, but there exists a read $r \in D$ that returns the value written by w, despite the fact that $D <_g A <_g w$.

In managed code, it appears possible to use run-time sandboxing to contain any erroneous behavior in doomed transactions [12,7]. For unmanaged code, or as an alternative for managed environments, we present three mechanisms to avoid the inconsistencies that give rise to the problem in the first place.

Quiescence. A *transactional fence* [19] blocks the caller until all active transactions have committed or aborted *and cleaned up*. This means that a fence f participates in the natural order $<_n$ on transactions and in program order $<_p$ for its thread. Since $D <_t A <_t f <_p w$, and f waits for D to clean up, we are guaranteed that the implementation respects $D <_g w$.

Polling. Polling for the presence of privatizers can tell a transaction when it needs to check to see if it is doomed. This mechanism requires every privatizing transaction to atomically increment a global counter (e.g., with a fetch-and-increment [`fai`] instruction) that is polled by every transaction, on every read of shared data. When a transaction reads a new value of the counter, it validates its read set and, if doomed, aborts before it can see an inconsistency caused by a private write. Pseudocode for this mechanism appears in Figure 5.

```
TxBegin(desc)
    ...
    desc->priv_cache = priv_count

Acquire()
    fai(&priv_count)
```

```
TxRead(&addr, &val, desc)
    ... // read value consistently
    t = priv_count
    if (t != desc->priv_cache)
        validate()
        desc->priv_cache = t
```

Fig. 5. Polling to detect doomed transactions

Like a transactional fence, the increment c of `priv_count` participates in $<_n$. Suppose D contains a read r that sees (incorrectly) a value written by w, with $D <_t A <_t c <_p w$. Since c increments `priv_count` and D reads `priv_count` as part of every `TxRead`, D must abort before completing r, a contradiction.

Timestamp Polling. In a timestamp-based STM like TL2 [4], every writer increments a global timestamp. If all transactions are writers (and hence all update the global timestamp), polling this timestamp prevents the doomed transaction problem, using the same argument as above. When a read-only transaction A may privatize (privatization by antidependence), it does so by reading a value written by a previous non-privatizing transaction. Thus it suffices to observe that while A may not increment the global timestamp, A is ordered after some other transaction W ($W <_n A$) that committed a write in order for A's privatization to succeed. If D is doomed because of the privatization (and still active), it must read a value written by W. W's increment of the global timestamp is sufficient to force a polling-based abort in D prior to any use of an inconsistent value.

4.2 Preventing the Delayed Cleanup Problem

The delayed cleanup problem occurs when an STM implementation admits an execution containing a successful transaction D whose cleanup is delayed,[3] a nontransactional read r, an acquiring transaction $A <_p r$, and a natural transaction order $<_n$ such that any $<_t$ consistent with $<_n$, when combined with $<_p$, induces a global (SS, SSS, SFS) order $<_g$ that fails to explain the value read by r—specifically, when there are dependences that force $D <_t A$, but r returns the value from some write w despite an intervening write $w' \in D$ to the same location. We propose two mechanisms to avoid this problem.

Quiescence. As before, let A be immediately followed by a transactional fence f, and let D commit before A. Since $D <_t A <_t f <_p r$, and f waits for D to clean up, we are guaranteed that the implementation respects $D <_g r$.

Optimized Commit Linearization. Menon et al. [12] describe a commit linearization mechanism in which all transactions must increment global counters at transaction begin and while committing. Unfortunately, forcing read-only transactions to modify shared data has a serious performance cost. To avoid this cost, we propose an alternative implementation of commit linearization in Figure 6. The implementation is inspired by the classic ticket lock: writer transactions increment the global timestamp, clean up, and then increment a second `cleanups` counter in order. Readers read the timestamp and then wait for `cleanups` to catch up.

```
TxBegin(desc)
    ...
    start = timestamp
    while (cleanups < start)
        yield()
```

```
TxCommit(desc) // not read only
    acquire_locks()
    my_timestamp = fai(timestamp)
    if (validate())
        // copy values to shared memory
    else
        must_restart = true
    release_locks()
    while (cleanups != (my_timestamp - 1))
        yield()
    cleanups = my_timestamp
```

Fig. 6. An implementation of commit linearization in which read-only transactions do not update shared metadata

We argue that this mechanism is privatization safe with respect to (even non-selective) strict serializability. If writer D precedes writer A in natural order but has yet to clean up, then D will not yet have updated the `cleanups` counter, and A's `TxCommit` operation will wait for it. Any subsequent read in A's thread can be guaranteed that D has completed.

Suppose that reader A privatizes by antidependence. If D increments the global timestamp before A begins, A must wait in `TxBegin` for D to clean up, avoiding

[3] In redo log-based STMs, cleanup entails replaying speculative writes and releasing ownership of locations. As noted in footnote 2, undo log-based STMs have an analogous problem, which we ignore here.

the problem. If D is still active when A begins, there must exist some other writer W ($W <_n A$) that committed a write in order for A's privatization to succeed. Clearly $D \neq W$, or else D's write of the location in the antidependence would have forced A to abort. Moreover, since the program is data-race-free, $D <_n W$. For D to still be active when A begins we must have W still active when A begins, a contradiction, since W writes a location that A reads, and A does not abort.

Commit Fence. Our commit fence mechanism combines the best features of the transactional fence and commit linearization. As in the transactional fence, there is no single global variable that is accessed by all committing writer transactions. As in commit linearization, only committing transactions can cause a privatizer to delay. Pseudocode for the mechanism appears in Figure 7.

```
TxCommit(desc) // not read only
  commit_fence[my_slot]++
  acquire_locks()
  my_timestamp = fai(timestamp)
  if (validate())
    // copy values to shared memory
  else
    must_restart = true
  release_locks()
  commit_fence[my_slot]++
```

```
Acquire()
  num = commit_fence.size
  for (i = 0 .. num)
    local[i] = commit_fence[i]
  for (i = 0 .. num)
    if (local[i].is_odd())
      while (commit_fence[i] == local[i])
        yield();
```

Fig. 7. The commit fence

The commit fence ensures that any transaction sets an indicator before acquiring locks, and unsets the indicator after releasing locks. At its acquire fence, a privatizing transaction samples all transactions' indicators, and waits until it observes every indicator in an unset state. This commit fence c provides privatization safety as above: if D accesses data privatized by A, and if $D <_t A$, then D must update the commit fence before A completes its commit sequence. Since $A <_p c$, and c observes D's in-flight modifications, c will not return until D completes, and thus $D <_g c$.

Unlike the full transactional fence, this mechanism does not prevent the doomed transaction problem. Like the transactional fence, it can cause an acquirer A to wait on a committing, nonconflicting transaction B even when $A <_t B$. However, as in commit linearization, A will block only for committing transactions, never for in-flight transactions.

4.3 Preventing Publication Errors

Under SSS, publication safety can be expressed as the condition that if $w <_p R <_t T$, where $R \in \mathcal{R}$, then $w <_{sss} T$, even if T reads the location written by w. We propose two release implementations that guarantee this condition.

Quiescence. Placing a transactional fence between a nontransactional access and a subsequent publishing transaction prevents the publication problem. Suppose $w <_p R$, where w is a nontransactional write of location l and $R \in \mathcal{R}$. The publication problem manifests when some transaction T prefetches l before w writes it, but $R <_n T$. If T

begins before w, a fence between w and R forces T to complete before R begins, so $R \nless_n T$.

Polling. Polling may also be used, as shown in Figure 8, to prevent the publication problem. Instead of having a publisher wait for all active transactions to complete, each active transaction T checks at each read (and at `TxCommit`) to see whether a release operation e has occurred since T began execution. If $e <_p R <_n T$ and T is successful, we are guaranteed that T was not active at the time e occurred, and T could not have prefetched any published datum.

```
TxBegin(desc)                           TxCommit(desc)
    ...                                     ... // acquire locks, validate
    desc->pub_cache = pub_count             if (pub_count != desc->pub_cache)
                                                abort()
TxRead(&addr, &val, desc)                   ...
    ... // read value consistently
    if (pub_count != desc->pub_cache)   Release()
        abort()                             fai(&pub_count)
```

Fig. 8. Polling to detect publication

4.4 Preventing Flow Publication Errors

Flow-serializable publication safety requires only that if $w <_p R <_f T$, where $R \in \mathcal{R}$, then $w <_{sfs} T$. That is, the existence of R need not cause T to abort unless T reads something R wrote. Menon et al. use timestamps to achieve ALA semantics. Their timestamps, however, are not TL2 timestamps, as they are assigned at begin time, even for read-only transactions. We briefly argue that TL2 timestamps [4] provide SFS (Figure 9).

```
TxStart(desc)                             TxRead(addr* addr, addr* val, desc)
    ...                                       // addr not in write set
    desc->start = timestamp;                  orec o = get_orec(addr);
                                              if (o.locked ||
TxCommit(desc)                                        o.timestamp > desc->start)
    ... // acquire locks                          abort()
    endtime = get_timestamp();                ... // read value
    if (validate())                           if (o != get_orec(addr))
        // copy values to shared memory           abort()
        foreach (lock in writeset)            ... // log orec, return value
            lock.releaseAtVersion(endtime)
        ...
    ...
```

Fig. 9. TL2-style timestamps

Let us assume (due perhaps to compiler reordering) that T races with R to prefetch l. This indicates that T could not start after R, and thus $T.\texttt{start} \leq R.\texttt{start}$. If it also holds that $R <_f T$, then when R acquires timestamp t at commit time, $t > R.\texttt{start} \geq T.\texttt{start}$. R subsequently writes t into all lock words that it holds. If $R <_f T$, T must read some location written by R. During T's call to `TxRead` for l, the test of the timestamp will cause T to abort, restart, and re-read l after R. Alternately, if

T reads the lock covering l before R commits, then either $T <_t R$, or T will abort at its next validation. In either case, T cannot be ordered both before w and after R.

We note that TL2 timestamps reduce the scalability of non-publishing, non-privatizing transactions when compared to the *extendable* timestamps of Riegel et al. [15]. They also enforce more ordering than the timestamps of Menon et al. [12], which do not abort a transaction T that reads a value written by a publisher who started (but did not commit) before T began. Finally, TL2-style timestamps preclude partial rollback for closed nested transactions: If B reads a value that C writes, and B is nested within A, then the requirement for B to order after C necessitates that A order after C as well. Even if all accesses prior to B in A do not conflict with C, A must restart to acquire a start time compatible with ordering after C.

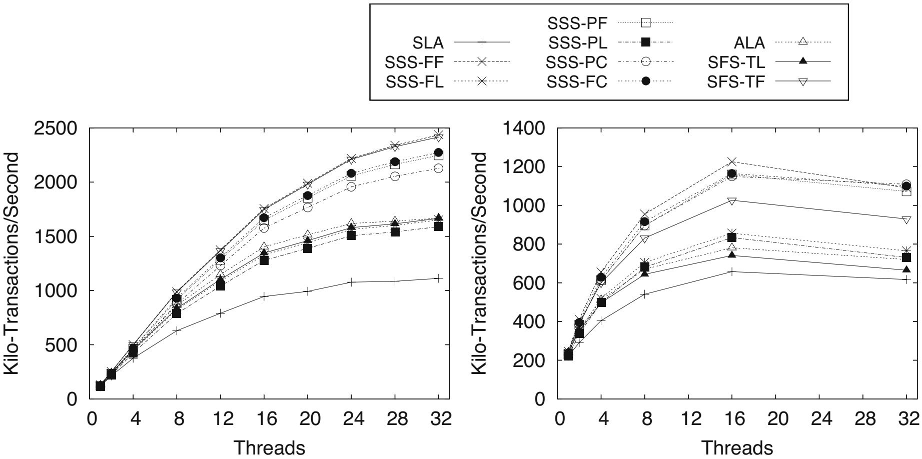
5 Evaluation

In this section, we evaluate the role that selective semantics can play in reducing transaction latency without compromising correctness. We use targeted microbenchmarks to approximate three common idioms for privatization and publication. All experiments were conducted on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks were written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five trials.

5.1 STM Runtime Configuration

We use a word-based STM with 1 million ownership records, commit-time locking, and buffered updates. Our STM uses timestamps to avoid validation overhead, and unless otherwise noted, the timestamps employ a extendable time base scheme [15] to safely allow nonconflicting transactions to ignore some timestamp-based aborts. From this STM, we derive 10 runtimes:

- SLA. Uses the start and commit linearization of Menon et al. [12], and polls the global timestamp on reads to avoid the doomed transaction problem.
- SSS-FF. Uses transactional fences before publishing transactions and after privatizing transactions.
- SSS-FL. Uses our commit linearization for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.
- SSS-PF. Uses polling for publication safety, and transactional fences for privatization.
- SSS-PL. Uses polling for publication safety, commit linearization for privatization, and polling to avoid the doomed transaction problem.
- SSS-PC. Uses polling for publication safety, commit fences for privatization, and polling to avoid the doomed transaction problem.
- SSS-FC. Uses commit fences for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.



(a) Nontransactional phase of a phased privatization workload, modeled as a low contention red-black tree.

(b) Worklist privatization, using transactional producers and a single private consumer.

Fig. 10. Privatization microbenchmarks

- ALA. Uses TL2-style timestamps for publication safety, and commit linearization with polling of the timestamp for privatization.
- SFS-TL. Identical to ALA, but maintains a privatization counter, separate from the timestamp, to avoid the doomed transaction problem.
- SFS-TF. Uses TL2-style timestamps for publication safety, and transactional fences for privatization.

5.2 Phased Privatization

In some applications, program structure, such as barriers and thread join points, ensures that all threads agree that a datum is public or private [17]. Since these phase boundaries are ordered globally, and with respect to $<_t$, no additional instrumentation is required for correctness on acquiring and releasing transactions. However, as in applications with no privatization or publication, ALA or SLA semantics cause ordering latency for all transactions.

We model the transactional phase of a phased workload with a low-contention red-black tree (Figure 10(a)). Threads use 20-bit keys and perform 80% lookups, 10% inserts, and 10% removes. We ensure steady state by pre-populating the tree to 50% full.

Since SLA serializes all transactions, it consistently underperforms the other runtimes. Similarly, ALA and other mechanisms that use commit linearization fail to scale as well as mechanisms that do not impose additional ordering on all writers at commit time. However, the SSS-FL, SSS-PL, and SFS-TL curves show that our optimized mechanism for commit linearization, which does not force read-only transactions to increment a global shared counter, achieves better throughput when writing transactions are rare.[4]

[4] Higher writer ratios show the same separation, with less difference between commit linearization and SLA.

Our test platform clearly matters: the Niagara's single shared L2 provides low-latency write misses for the variables used to provide commit linearization, preventing additional slowdown as the lines holding the commit and cleanup counters bounce between cores. At the same time, the Niagara's single-issue cores cannot mask overheads due to polling for publication safety. Thus SSS-FF and SFS-TF perform best. Since commit fences require polling to prevent the doomed transaction problem, SSS-FC performs worse than SSS-FF.

5.3 Worklist Privatization

The privatization problem was first discovered in worklist-based applications, where transactions cooperatively create a task and enqueue it into a nontransactional worklist. When the task is removed from the worklist, the data comprising the task are logically private. Abstractly, these workloads publish by sharing a reference to previously private data, and privatize by removing all shared references to a datum. In the absence of value speculation, these applications admit the privatization problem, but not the publication problem.

To evaluate selective semantics for worklists, we use a producer/consumer benchmark, in which multiple threads cooperatively produce tasks, and then pass the tasks to a consumer thread. We model tasks as red-black trees holding approximately 32 6-bit values, and build tasks using an equal mix of insert, remove, and lookup operations on initially empty trees. Once a tree is found to hold a distinguished value, it is privatized and sent to a nontransactional consumer thread. For the experiment in Figure 10(b), the consumer is fast enough that even 32 producers cannot oversaturate it.

Mechanisms that impose excessive ordering (SLA and ALA) or use commit linearization (SSS-FL, SSS-PL, and SFS-TL) perform worst. Furthermore, since transactions are small, and since privatizing a task does not prevent other producers from constructing a new task, the overhead of a transactional fence (SSS-FF and SSS-PF) at privatization time is as low as the commit fence (SSS-PC and SSS-FC). TL2-style timestamps (ALA, SFS-TL, and SFS-TF) decrease scalability. Again, due to the architecture of the Niagara CPU, polling for publication (SSS-PF, SSS-PL, and SSS-PC) or doomed transaction safety (SLA, SSS-FL, SSS-PL, SSS-PC, SSS-FC, ALA, and SFS-TL) increases latency slightly.

5.4 Indirect Publication

When the value of a shared variable determines whether another location is safe for private access, both the publication and privatization problems can arise. This programming idiom is analogous to locking: the period of private use corresponds to a critical section. We explore it with extensible hashing.

In Figure 11, transactions perform 8 puts into a set of 256 hash tables, where each table uses per-bucket sorted linked lists. If transaction T encounters a bucket containing more than 4 elements, it sets a local flag. After committing, T privatizes and then rehashes any flagged hash tables, doubling the bucket count (initially 8). In order to maintain a steady rate of privatization, if a table's bucket count reaches 2^{13}, the table is privatized and passed to a worker thread W. W replaces the table with an empty 8-bucket table.

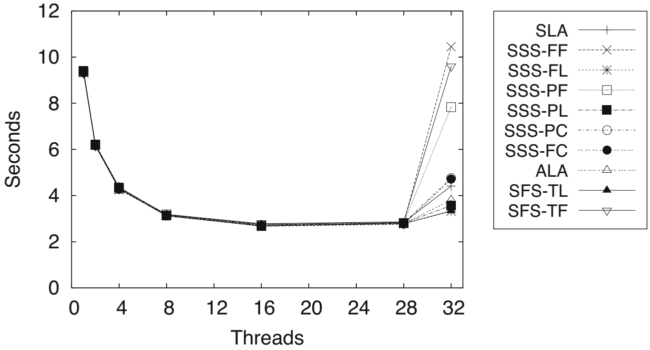


Fig. 11. Indirect publication, modeled as extensible hashing with 256 tables. With time on the y axis, lower values are better.

Scalability is low, because privatization for rehashing essentially locks the hash tables. Even with 256 tables, the duration of rehashing is significantly longer than the duration of several 8-put transactions, and thus even at two threads, when one thread privatizes a table for rehashing, the other thread is likely to block while attempting to access that table.

The different mechanisms vary only when there is preemption (at 32 worker threads, since there is an additional thread for performing table destruction). At this point, all privatization mechanisms risk waiting for a preempted transaction to resume. The effect is worst for the transactional fence (SSS-FF, SSS-PF, SFS-TF), since it must wait for all active transactions. Our commit linearization (SSS-FL, SSS-PL, ALA, SFS-TL) fares much better, since threads only wait for previously committed writers, who necessarily are at the very end of their execution. SLA linearization lies somewhere in between. Unlike transactional fences (which also avoid the doomed transaction problem), its use of a global timestamp avoids waiting for logically "later" transactions that are still in progress, but unlike commit linearization, it also must wait on "earlier" transactions that have not reached their commit point. The commit fence (SSS-PC, SSS-FC) performs slightly worse than SLA, indicating that waiting on "later" transactions is more expensive than waiting for "earlier" transactions.

The low latency of fence-based publication appears to be an artifact of the indirect publication idiom. At a fence, the releasing transaction waits for all concurrent transactions to commit or abort and clean up. Since some hash tables are effectively locked, most concurrent threads will execute "restart" transactions to spin-wait for the tables to become public. In our lazy STM, such transactions do not hold locks and can rapidly restart, preventing delays at the publication fence.

6 Conclusions

In this paper we argued that TM semantics should be specified in terms of permissible memory access orderings, rather than by recourse to locks. In our specification, traditional strict serializability (SS) takes the place of single lock atomicity (SLA). To reduce the cost of publication and privatization, we proposed *selective* strict serializability (SSS), which enforces a global order between transactional and nontransactional accesses of a given thread only when transactions are marked as acquiring or releasing. We also proposed a weaker selective flow serializability (SFS), that enforces release ordering only with respect to transactions that read a location written by the releasing transaction. We described several possible implementations of both SSS and SFS, with informal correctness arguments.

Preliminary experiments suggest several performance-related conclusions: (1) By imposing the cost of publication and privatization only when they actually occur, selective ordering of nontransactional accesses can offer significant performance advantages. (2) Given selectivity, there seems to be no compelling argument to relax the serial ordering of transactions. Moreover we suspect that requiring annotations will ultimately help the programmer and compiler to generate race-free code. (3) At the same time, the additional relaxation of SFS (and, similarly, ALA), offers little if any additional benefit. Since SSS is simpler to explain to novice programmers, permits true closed nesting, and is orthogonal to the underlying STM, we currently see no reason to support more relaxed semantics, whether they are defined in terms of prescient lock acquisition or memory access ordering.

Acknowledgements

We are grateful to Victor Luchangco and to the anonymous referees for insightful feedback on earlier versions of this manuscript.

References

[1] Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of Transactional Memory and Automatic Mutual Exclusion. In: Conf. Record of the 35th ACM Symp. on Principles of Programming Languages, San Francisco, CA (2008)

[2] Blundell, C., Lewis, E.C., Martin, M.M.K.: Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters 5(2) (2006)

[3] Boehm, H.-J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: SIGPLAN 2008 Conf. on Programming Language Design and Implementation, Tucson, AZ (2008)

[4] Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: 20th Intl. Symp. on Distributed Computing, Stockholm, Sweden (2006)

[5] Grossman, D., Manson, J., Pugh, W.: What Do High-Level Memory Models Mean for Transactions? In: ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, San Jose, CA, Held in conjunction with ASPLOS XII (2006)

[6] Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: 13th ACM Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT (2008)

[7] Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: SIGPLAN 2006 Conf. on Programming Language Design and Implementation, Ottawa, ON, Canada (2006)

[8] Larus, J.R., Rajwar, R.: Transactional Memory, Synthesis Lectures on Computer Architecture. Morgan & Claypool (2007)

[9] Luchangco, V.: Against Lock-Based Semantics for Transactional Memory. In: 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany, Brief announcement (2008)

[10] Manson, J., Pugh, W., Adve, S.: The Java Memory Model. In: Conf. Record of the 32nd ACM Symp. on Principles of Programming Languages, Long Beach, CA (2005)

[11] Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: 2008 Intl. Conf. on Parallel Processing, Portland, OR (2008)

[12] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R.L., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (2008)

[13] Moore, K.F., Grossman, D.: High-Level Small-Step Operational Semantics for Transactions. In: Conf. Record of the 35th ACM Symp. on Principles of Programming Languages, San Francisco, CA (2008)

[14] Papadimitriou, C.H.: The Serializability of Concurrent Database Updates. Journal of the ACM 26(4), 631–653 (1979)

[15] Riegel, T., Fetzer, C., Felber, P.: Time-based Transactional Memory with Scalable Time Bases. In: 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures, San Diego, CA (2007)

[16] Scott, M.L.: Sequential Specification of Transactional Memory Semantics. In: 1st ACM SIGPLAN Workshop on Transactional Computin, Ottawa, ON, Canada (2006)

[17] Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Delaunay Triangulation with Transactions and Barriers. In: IEEE Intl. Symp. on Workload Characterization, Boston, MA, Benchmarks track (2007)

[18] Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing Isolation and Ordering in STM. In: SIGPLAN 2007 Conf. on Programming Language Design and Implementation, San Diego, CA (2007)

[19] Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization Techniques for Software Transactional Memory. In: 26th ACM Symp. on Principles of Distributed Computing, Portland, OR (2007); Brief announcement. Extended version available as TR 915, Dept. of Computer Science, Univ. of Rochester (2007)

[20] Wang, C., Chen, W.-Y., Wu, Y., Saha, B., Adl-Tabatabai, A.-R.: Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In: Intl. Symp. on Code Generation and Optimization, San Jose, CA (2007)

CQS-Pair: Cyclic Quorum System Pair for Wakeup Scheduling in Wireless Sensor Networks

Shouwen Lai[1], Bo Zhang[1], Binoy Ravindran[1], and Hyeonjoong Cho[2]

[1] ECE Department, Virginia Tech, Blacksburg VA 24060, USA
{swlai,alexzbzb,binoy}@vt.edu
[2] ETRI, RFID/USN Research Group, Yuseong-gu, Daejeon 305-350, Korea
raycho@etri.re.kr

Abstract. Due to the heterogenous power-saving requirement in wireless sensor networks, we propose the Cyclic Quorum System Pair (CQS-Pair) which can guarantee that two asynchronous nodes adopt different cyclic quorum systems can hear each other at least once in bounded time intervals. To quickly assemble a CQS-Pair, we present a fast construction scheme, which is based on the Multiplier Theorem and the (N, k, M, l)-difference pair defined by us. We show that via the CQS-Pair, two heterogenous nodes can achieve different power saving ratios while maintaining connectivity. The performance of a CQS-Pair is analyzed in terms of average delay and quorum ratio.

1 Introduction

Wireless sensor networks have recently received increased attention for a broad array of applications such as surveillance, environment monitoring, medical diagnostics, and industrial control. As wireless sensor nodes usually rely on portable power sources such as batteries to provide the necessary power, their power management has become a crucial issue. It has been observed that idle energy plays an important role for saving energy in wireless sensor networks [3]. Most existing radios (i.e., CC2420) used in wireless sensor networks support different modes, like transmit/receive mode, idle mode, and sleep mode. In idle mode, the radio is not communicating but the radio circuitry is still turned on, resulting in energy consumption which is only slightly less than that in the transmitting or receiving states.

In order to save idle energy, it is necessary to introduce a wakeup mechanism for sensor nodes in the presence of pending transmission. The major objective of the wakeup mechanism is to maintain network connectivity while reducing the idle state energy consumption. Existing wakeup mechanisms fall into three categories: on-demand wakeup, scheduled rendezvous, and asynchronous wakeup.

In on-demand wakeup mechanisms [7], out-band signaling is used to wake sleeping nodes in an on-demand manner. For example, with the help of a paging signal, a node listening on a page channel can be woken up. As page radios can operate at lower power consumption, this strategy is very energy efficient. However, it suffers from increased implementation complexity.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 295–310, 2008.

In scheduled rendezvous wakeup mechanisms, low-power sleeping nodes wake up at the same time periodically to communicate with one another. One example is the S-MAC [12] protocol.

The third category, asynchronous wakeup [13], is also well studied. Compared to the scheduled rendezvous wakeup mechanisms, asynchronous wakeup does not require clock synchronization. Each node follows its own wakeup schedule in idle state, as long as the wakeup intervals among neighbors overlap. To meet this requirement, nodes usually have to wakeup more frequently than in the scheduled rendezvous mechanisms. But the advantage of asynchronous wakeup is the easiness in implementation. Furthermore, it can ensure network connectivity even in highly dynamic networks.

The quorum-based wakeup scheduling paradigm, also called quorum-based power saving (QPS) protocol [4,11], has recently received significant attentions as an asynchronous wakeup solution. In a QPS protocol, the time axis on each node is evenly divided into beacon intervals. Given an integer n, a quorum system defines a cycle pattern, which specifies the awake/sleep scheduling pattern during n continuous beacon intervals for each node. We call n the *cycle length*, since the pattern repeats every n beacon intervals. A node may stay awake or sleep during each beacon interval. QPS protocols can guarantee that at least one awake interval overlaps between two adjacent nodes with only $O(\sqrt{n})$ beacon intervals being awake in each node. Most previous work only consider homogenous quorum systems for asynchronous wakeup scheduling, which means that quorum systems for all nodes have the same *cycle length*.

However, it is often desirable that heterogenous nodes (i.e, clusterheads and members) have heterogenous quorum-based wakeup schedule (or different *cycle lengths*). We denote two quorums from different quorum systems as *heterogenous quorums* in this paper. If two adjacent nodes adopt heterogenous quorums as their wakeup schedules, they have different cycle lengths and different wakeup patterns. The problem is how to guarantee that the two nodes can discover each other within bounded delay in the presence of clock drift.

In this paper, we present the Cyclic Quorum System Pair (CQS-Pair) which contains a pair of quorum systems suitable for heterogenous quorum-based wakeup schedule. The mechanism of CQS-Pair can guarantee that two adjacent nodes adopt heterogenous quorums from such a pair as their wakeup schedule, can hear each other at least once within one super cycle length (i.e., the larger cycle length in the CQS-Pair). With the help of the CQS-Pair, wireless sensor networks can achieve better trade-off between energy consumption and average delay. For example, all cluster-heads and gateway nodes can pick up a quorum from the quorum system with smaller cycle length as their wake up schedule, to get smaller discovery delay. In addition, all members in a cluster can choose a quorum from the system with longer cycle length as their wakeup schedules, in order to save more idle energy.

Our contribution. We present the Cyclic Quorum system Pair (CQS-Pair) which can be applied as a solution for the problem of heterogeneous quorum-based wakeup scheduling (h-QPS, defined in Section 2.4) in wireless sensor networks, and propose a fast constructing scheme via the Multiplier Theorem and (N, k, M, l)-difference pair defined by us. The CQS-Pair is an optimal design in terms of energy saving ratios given a pair of cycle lengths (n and m, $n \leq m$). The fast constructing scheme can greatly improve the speed of finding an optimal quorum comparing with previous exhaustive methods. We also analyze the performance of CQS-Pair in aspects of expected delay ($\frac{n-1}{2} < E(delay) < \frac{m-1}{2}$), quorum ratio, energy saving ratio, and practical issues on how to support multicast/broadcast. This is the first solution to heterogenous quorum-based wakeup scheduling as we are not aware of any other existed similar solutions.

Paper Structure. The rest of the paper is organized as follows: In Section 2, we outline some basic preliminaries for quorum-based power-saving protocols. The detailed design and construction scheme of Cyclic Quorum System Pair is discussed in Section 3. We analyze the performance of the CQS-Pair in Section 4. Related work is presented in Section 5. We conclude in Section 6.

2 Preliminaries

2.1 Network Model and Assumptions

We represent a multi-hop wireless sensor network by a directed graph $G(V, E)$, where V is the set of network nodes ($|V| = N$), and E is the set of edges. If node v_j is within the transmission range of node v_i, then an edge (v_i, v_j) is in E. We assume bidirectional links. The major objective of quorum-based wakeup scheduling is to maintain network connectivity regardless of clock drift. Here we use the term "connectivity" loosely, in the sense that a topologically connected network in our context may not be connected at any time; instead, all the nodes are reachable from a node within a finite amount of time.

We also make the following assumptions when applying quorum-based system for asynchronous wakeup mechanism: 1) All time intervals/slots have equal length, being 1 in this paper for convenient presentation; 2)In the beginning of a beacon interval, beacon messages will be sent out so that nodes can hear each other; and 3) The overhead of turning on and shutting down radio is negligibly small.

The length of one time interval depends on application-specific requirements. For a radio compliant with IEEE 802.15.4, the bandwidth is approximately 128kb/s and a typical packet size is 512KB. Given this, the slot length (beacon interval) is $\leq 50ms$. The second assumption, also adopted by previous work [13] [4], is for the convenience of theoretical analysis.

2.2 Quorum-Based Power-Saving Protocols (QPS)

We use the following definitions for quorum system. Given a cycle length n, let $U = \{0, \cdots, n-1\}$ be an universal set.

Definition 1. *A quorum system Q under U is a collection of non-empty subsets of U, each called a quorum, which satisfies the intersection property: $\forall G, H \in Q : G \cap H \neq \emptyset$.*

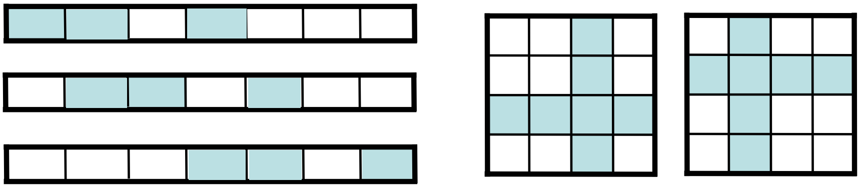


Fig. 1. Cyclic Quorum System (left) and Grid Quorum System (right)

Definition 2. *Given an integer $i \geq 0$ and quorum G in a quorum system $\mathcal{Q}$ under U, we define $G + i = \{(x + i) \bmod n : x \in G\}$.*

Definition 3. *A quorum system $\mathcal{Q}$ under U is said to have the* rotation closure property *if $\forall G, H \in \mathcal{Q}$, $i \in \{0, 1, ...n - 1\}$: $G \cap (H + i) \neq \emptyset$.*

There are two widely used quorum systems, grid quorum system and cyclic quorum system, that satisfy the rotation closure property.

Grid-Quorum System [6]. In a grid quorum system shown in Figure 1, elements are arranged as a $\sqrt{n} \times \sqrt{n}$ array (square). A quorum can be any set containing a column and a row of elements in the array. The quorum size in a square grid quorum system is $2\sqrt{n} - 1$. An alternative is a "triangle" grid-based quorum in which all elements are organized in a "triangle" fashion. The quorum size in "triangle" quorum system is approximately $\sqrt{2}\sqrt{n}$.

Cyclic Quorum System [6]. A cyclic quorum system is based on the ideas of cyclic block design and cyclic difference sets in combinatorial theory [8]. The solution set can be strictly symmetric for arbitrary n. For example, $\{1, 2, 4\}$ is a quorum from the cyclic quorum system with cycle length= 7. Figure 1 illustrates three quorums from a cyclic quorum system with cycle length 7.

Previous work [4] defined the **QPS** (quorum-based power-saving) problem as follows: Given an universal set $U = \{0, 1, ...n - 1\}$ $(n > 2)$ and a quorum system $\mathcal{Q}$ over U, two nodes picks up any quorum from $\mathcal{Q}$ as their wakeup schedule must have at least one overlap in every n consecutive time slots.

Theorem 1. *$\mathcal{Q}$ is a solution to the QPS problem if $\mathcal{Q}$ is a quorum system satisfying the* rotation closure property.

Theorem 2. *Both grid-quorum systems and cyclic quorum systems satisfy the rotation closure property and can be applied for QPS in wireless sensor networks.*

Proofs of Theorems 1 and 2 can be found in [4].

2.3 Neighbor Discovery under Partial Overlap

Since sensor nodes are subject to clock drift, we cannot assume that time slots are exactly aligned to their boundaries. In most case, two nodes only have partial overlap in a certain time interval. It has been proven that two nodes with the quorum-based wakeup schedule can discover each other even under *partial overlap.*

Theorem 3. *If two quorums ensure a minimum of one overlapping slot, then with the help of a beacon message at the beginning of each slot, two neighboring nodes can hear each others' beacons with at least once.*

Theorem 3's proof is presented in [13]. This theorem ensures that two neighboring nodes can always discover each other within bounded time if all beacon messages are transmitted successfully. This property also holds true even in the case that two originally disconnected subsets of nodes join together as long as they use the same quorum system.

2.4 Heterogeneous Quorum-Based Power Saving (h-QPS)

We introduce h-QPS (heterogeneous Quorum-Based Power saving) problem in this section. In sensor networks, it is often desirable that different nodes wakeup in heterogeneous quorum-based schedules. First, many wireless sensor networks have heterogeneous entities, like cluster-heads, gateways, relay nodes, dominating set [1]. They have different requirements regarding average neighbor discovery delay and energy saving ratio. Regarding cyclic quorum systems, generally, cluster-heads should wakeup based on a quorum system with small cycle length, and member nodes with longer cycle length. Second, wireless sensor networks that are used in applications such as environment monitoring typically have seasonally-varying power saving requirements. For example, a sensor network for wild fire monitoring may require a larger energy saving ratio during winter seasons. Thus, they often desire variable cycle-length wakeups in different seasons.

Considering the two heterogeneous quorum systems $\mathcal{X}$ over $\{0, 1, \cdots, n-1\}$ and $\mathcal{Y}$ over $\{0, 1, \cdots, m-1\}$ $(n \leq m)$, we define the **h-QPS** problem as follows: design a pair $(\mathcal{X}, \mathcal{Y})$ in order to guarantee that:

1. two nodes picking up two quorums $G \in \mathcal{X}$ and $H \in \mathcal{Y}$ as their wakeup schedules respectively can hear each other at least once within every m consecutive slots; and
2. $\mathcal{X}$ and $\mathcal{Y}$ are solutions to QPS individually.

A solution to the h-QPS problem is important to keep connectivity when we want to dynamically change the quorum systems between all nodes. Suppose all nodes in a network initially wakeup via a larger cycle length. When there is a need to reduce the cycle length (i.e., to meet a delay requirement or vary with changing seasons), the sink node can send a request to the whole network gradually through some relay nodes. The connectivity between a relay node and the remaining nodes will be lost if the relay node first changes its wakeup scheduling to a new quorum schedule which cannot overlap with original schedules of the remaining nodes.

Although grid-quorum systems and cyclic-quorum systems can be applied as a solution for QPS problem, it is not necessary meaning that they can be a solution to the h-QPS problem. We will show this in Section 3.

3 Cyclic Quorum System Pair

In this section, we present the CQS-Pair which is based on the cyclic block design and cyclic difference sets in combinatorial theory [8]. CQS-Pair can be applied as a solution to the h-QPS problem.

3.1 Heterogeneous Rotation Closure Property

First, we define some concepts to facilitate our presentation. Some definitions are extended from those in [6]. We will also use definitions from [8] to denote $\mathbb{Z}_n$ as an finite field of order n and $(\mathbb{Z}_n, +)$ as an Abelian Group.

Definition 4. *Let A be a set in $(\mathbb{Z}_n, +)$. For any $g \in \mathbb{Z}_n$, we define $A + g = \{(x + g)\ mod\ n : x \in A\}$.*

Definition 5. *(Cyclic set) Let X be a set in $(\mathbb{Z}_n, +)$. The set $C(X, \mathbb{Z}_n)$ is called a cyclic set (or cyclic group) of X if $C(X, \mathbb{Z}_n) = \{X + i | \forall i \in \mathbb{Z}_n\}$.*

Definition 6. *(*p*-extension) Given two positive integers n and p, suppose $U = \{0, 1, \cdots, n-1\}$ and let $U^p = \{0, \cdots, p*n-1\}$. For a set $A = \{a_i | 1 \leq i \leq k, a_i \in U\}$, A's* p*-extension is defined as $A^p = \{a_i + j*n | 1 \leq i \leq k, 0 \leq j \leq p-1, a_i \in U\}$ over U^p. For a cyclic quorum system $\mathcal{Q}=\{A, A+1, \cdots, A+n-1\}$ over U, $\mathcal{Q}$'s p-extension is defined as $\mathcal{Q}^p=\{A^p, (A+1)^p, \cdots, (A+n-1)^p\}$ over U^p.*

For example, if $A = \{1, 2, 4\}$ in $(\mathbb{Z}_7, +)$, $A^3 = \{1, 2, 4, 8, 9, 11, 15, 16, 18\}$ in $(\mathbb{Z}_{21}, +)$. If a quorum system $\mathcal{Q} = \{\{1, 2, 4\}, \{2, 3, 5\}, \{3, 4, 6\}\}$, we have $\mathcal{Q}^2 = \{\{1, 2, 4, 8, 9, 11\}, \{2, 3, 5, 9, 10, 12\}, \{3, 4, 6, 10, 11, 13\}\}$.

Definition 7. (Heterogeneous rotation closure property) *Given two positive integers N and M where $N \leq M$ and $p = \lceil \frac{M}{N} \rceil$, consider two quorum systems $\mathcal{X}$ over the universal set $\{0, \cdots N-1\}$ and $\mathcal{Y}$ over the universal set $\{0, \cdots M-1\}$. Let the quorum system $\mathcal{X}$'s p-extension be denoted as $\mathcal{X}^p$. The pair ($\mathcal{X}$,$\mathcal{Y}$) is said to satisfy the* heterogeneous rotation closure property *if :*

1. *$\forall G \in \mathcal{X}^p, H \in \mathcal{Y}, i \in \{0, \cdots M-1\}$: $G \cap (H + i) \neq \emptyset$, and*
2. *$\mathcal{X}$ and $\mathcal{Y}$ satisfy the* rotation closure property*, respectively.*

Consider two sets $A = \{1, 2, 4\}$ in $(\mathbb{Z}_7, +)$ and $B = \{1, 2, 4, 10\}$ in $(\mathbb{Z}_{13}, +)$. If two cyclic quorum systems $\mathcal{Q_A} = C(A, \mathbb{Z}_7)$ and $\mathcal{Q_B} = C(B, \mathbb{Z}_{13})$, then ${\mathcal{Q_A}}^2 = C(\{1, 2, 4, 8, 9, 11\}, \mathbb{Z}_{14})$. We can verify that any two quorums from ${\mathcal{Q_A}}^2$ and $\mathcal{Q_B}$ must have non-empty intersection. Thus, the pair ($\mathcal{Q_A}$,$\mathcal{Q_B}$) satisfies the *heterogeneous rotation closure property.*

Lemma 1. *If two quorum systems $\mathcal{X}$ and $\mathcal{Y}$ satisfy the heterogeneous rotation closure property, then the pair $(\mathcal{X}, \mathcal{Y})$ is a solution to the h-QPS problem.*

Proof. According to the Definition 7, if two quorum systems $\mathcal{X}$ and $\mathcal{Y}$ satisfy the heterogeneous rotation closure property, a quorum G from $\mathcal{X}$ and a quorum H from $\mathcal{Y}$ must overlap at once within the larger cycle length of $\mathcal{X}$ and $\mathcal{Y}$. So two nodes can hear each other if they pick up G and H as wakeup schedule based on the Theorem 3. It indicates that $(\mathcal{X}, \mathcal{Y})$ is a solution to the h-QPS problem.

Lemma 2. *Let Grid Quorum System-Pair be defined as a pair consisting of two grid quorum systems. Then Grid Quorum System-Pair satisfies the heterogeneous rotation closure property and can be a solution to the h-QPS problem.*

Proof. It has been proven in [4] that the grid quorum system satisfies the rotation closure property. Thus, we only need to prove that for two grid quorum systems $\mathcal{X}$ over $\{0,\cdots,n-1\}$ and $\mathcal{Y}$ over $\{0,\cdots,m-1\}$ ($n \leq m$, $p = \lceil \frac{m}{n} \rceil$), $\forall G^p \in \mathcal{X}^p, H \in \mathcal{Y}$, $i \in \{0,\cdots M-1\}$, there is $G^p \cap (H+i) \neq \emptyset$ or $(G+i)^p \cap H \neq \emptyset$. Consider a quorum G from $\mathcal{X}$ which contains all elements on the column c, namely c,$c+\sqrt{n}$,$\cdots$,$c+\sqrt{n}(\sqrt{n}-1)$, where $0 \leq c < \sqrt{n}$. Then, a quorum $(G+i)^p$ from the $p-extension$ of $\mathcal{X}$ contains elements which has the largest distance of $\sqrt{n}$ between any two consecutive elements. It must have an intersection with H since H contains a full row which has $\sqrt{m}$ ($\geq \sqrt{n}$) consecutive integers. Thus, Grid Quorum System-Pair satisfies the heterogeneous rotation closure property and can be a solution to the h-QPS problem.

3.2 (N, k, M, l)-Difference Pair

Definition 8. (N, k, λ)- ***difference set****. A set $D : \{a_1, ..., a_k\}$($\mod N$), $a_i \in [0, N-1]$, is called a (N, k, λ)- difference set if for every $d \neq 0$, there are EXACTLY λ ordered pair (a_i, a_j), $a_i, a_j \in D$ such that $a_i - a_j \equiv d$ $(modN)$.*

Definition 9. ***Relaxed*** (N, k)- ***difference set****. A set $D : \{a_1, ..., a_k\}$($\mod N$), $a_i \in [0, N-1]$, is called a relaxed (N, k)- difference set if for every $d \neq 0$, there exists at least one ordered pair (a_i, a_j), $a_i, a_j \in D$ such that $a_i - a_j \equiv d$ $(modN)$.*

The definition 8 and 9 were originally introduced in [6]. For example, the set $\{0, 1, 2, 4, 5, 8, 10\}$ *modulo* 15 is a $(15, 7, 3)$-difference set. The set $\{0, 1, 2, 4\}$ *modulo* 8 is a relaxed (8,4)-difference set. In the following, we will introduce two new definitions related with the CQS-Pair.

Definition 10. (N, k, M, l)***-difference pair.*** *Suppose $N \leq M$ and $p = \lceil \frac{M}{N} \rceil$. Consider two sets $A : \{a_1, \cdots a_k\}$ in $(\mathbb{Z}_N, +)$ and $B : \{b_1, \cdots b_l\}$ in $(\mathbb{Z}_M, +)$. The pair (A, B) is called a (N, k, M, l)-difference pair if $\forall d \in \{0, \cdots, M-1\}$, there exists at least one ordered pair $b_i \in B$ and $a_j^p \in A^p$ such that $b_i - a_j^p \equiv d$ $(mod\ M)$.*

Consider an example where $N = 7$ and $M = 13$. Let $A = \{1, 2, 4\}$ and $B = \{1, 3, 6, 7\}$ be two subsets in $(\mathbb{Z}_7, +)$ and $(\mathbb{Z}_{13}, +)$, respectively. Then (A, B) is a $(7, 3, 13, 4)$-difference pair, since for A^2 and B,

$$\begin{array}{lllll} 1 \equiv 3-2 & 2 \equiv 6-4 & 3 \equiv 1-11 & 4 \equiv 6-2 & 5 \equiv 6-1 \\ 6 \equiv 7-1 & 7 \equiv 3-9 & 8 \equiv 6-11 & 9 \equiv 7-11 & 10 \equiv 1-4\ (mod\ 13) \\ 11 \equiv 6-8 & 12 \equiv 1-2 & 13 \equiv 1-1 & & \end{array}$$

Definition 11. ***(Heterogeneous cyclic coterie pair)*** *Given two groups of sets $\mathcal{X} = \{A, A+1, \cdots, A+N-1\}$ over $\{0, \cdots, N-1\}$ and $\mathcal{Y} = \{B, B+1, \cdots, B+M-1\}$ over $\{0, \cdots, M-1\}$, suppose $N \leq M$ and $p = \lceil \frac{M}{N} \rceil$. We call $(\mathcal{X}, \mathcal{Y})$* heterogeneous cyclic coterie pair *if: $\forall (A+i)^p \subseteq \mathcal{X}^p$ and $(B+j) \subseteq \mathcal{Y}$, $(A+i)^p \cap (B+j) \neq \emptyset$.*

Theorem 4. *Suppose* $A=\{a_1,\cdots,a_k\}$ *in* $(\mathbb{Z}_N,+)$ *and* $A_i=A+i$*;* $B=\{b_1,\cdots,b_k\}$ *in* $(\mathbb{Z}_M,+)$ *and* $B_i=B+j$*, where* $N\le M$*. Given two groups of sets* $\mathcal{X}=\{A_i|0\le i\le N-1\}$ *and* $\mathcal{Y}=\{B_j|0\le j\le M-1\}$*, the pair* $(\mathcal{X},\mathcal{Y})$ *is a* heterogeneous cyclic coterie pair *if and only if* (A,B) *is a* (N,k,M,l)*-difference pair.*

Proof. Sufficient Condition. Without loss of generality, we assume that $j>i$ regarding two sets B_i and A_j^p, where $p=\lceil\frac{M}{N}\rceil$. Consider the r^{th} element of B_i and s^{th} element of A_j^p, denoted by $b_{i,r}$ and $a_{j,s}^p$, respectively. We will show that $b_{i,r}=a_{j,s}^p$. Let the r^{th} element of B be b_r and s^{th} element of A^p be a_s^p.

Then $b_{i,r}-a_{j,s}^p=(b_r-a_s^p+i-j)\ mod\ M$. According to the definition of (N,k,M,l)-difference pair, there must be some r and s such that $b_r-a_s^p\equiv j-i\ (mod\ M)$. Therefore, we can always choose a pair of r and s such that $b_{i,r}-a_{j,s}^p\equiv 0\ (mod\ M)$. It implies that $B_j\cap A_i^P\neq\emptyset$.

Necessary Condition. We prove the condition by contradiction. Assume that $B_j\cap A_i^P\neq\emptyset$ but (A,B) is not a (N,k,M,l)-difference pair. Then there exists a number $\in\{0,\cdots,M-1\}$, say t, in which $b_i-a_j^p\not\equiv t\ (mod\ M)$, $\forall i,j$.

Consider the r^{th} element of B_t and the s^{th} element of A^p. We have $b_{t,r}-a_s^p\equiv b_r-a_s^p+t\ (mod\ M)$. Since $B_t\cap A_i^P\neq\emptyset$, $b_{t,r}-a_s^p=0$ for some r and s. This implies that $b_r-a_s^p\equiv t\ (mod\ M)$ for some r and s, which contradicts with derivation of $b_i-a_j^p\not\equiv t\ (mod\ M)\ \forall i,j$ from the assumption.

If two groups of sets $\mathcal{X}$ and $\mathcal{Y}$ can form a *heterogeneous cyclic coterie pair*, they have at least one intersection within the larger cycle length. But the pair does not guarantee that any two sets from the same group, $\mathcal{X}$ or $\mathcal{Y}$, also have an intersection.

3.3 Definition and Verification of Cyclic Quorum System Pair

Definition 12. ***Cyclic Quorum System Pair (CQS-Pair).*** *Given two quorum sets* $\mathcal{X}=\{A,A+1,\cdots,A+N-1\}$ *over* $\{0,\cdots,N-1\}$ *and* $\mathcal{Y}=\{B,B+1,\cdots,B+M-1\}$ *over* $\{0,\cdots,M-1\}$*, suppose* $N\le M$*. We call* $(\mathcal{X},\mathcal{Y})$ CQS-Pair *if*

1. $(\mathcal{X},\mathcal{Y})$ *is a heterogeneous cyclic coterie pair; and*
2. $\mathcal{X}$ *and* $\mathcal{Y}$ *are cyclic quorum systems, respectively.*

Theorem 5. *Given two groups of sets* $\mathcal{X}=\{A,A+1,\cdots,A+N-1\}$ *and* $\mathcal{Y}=\{B,B+1,\cdots,B+M-1\}$*, where* $A=\{a_1,\cdots,a_k\}$ *in* $(\mathbb{Z}_N,+)$ *and* $B=\{b_1,\cdots,b_l\}$ *in* $(\mathbb{Z}_M,+)$ $(N\le M)$*, the pair* $(\mathcal{X},\mathcal{Y})$ *is a* CQS-Pair *if and only if*

1. (A,B) *is a* (N,k,M,l)*-difference pair; and*
2. A *is a relaxed* (N,k)*-difference set and* B *is a relaxed* (M,l)*-difference set.*

Proof. If (A,B) is a (N,k,M,l)-difference pair, we have that $(\mathcal{X},\mathcal{Y})$ is a heterogenous cyclic coterie pair. And if A and B are relaxed difference sets (defined in section 3.2) respectively, $\mathcal{X}$ and $\mathcal{Y}$ are cyclic quorum systems respectively. Similarly, we can prove that the converse is also true.

Corollary 1. *Given a cyclic quorum system* $\mathcal{X}$*,* $(\mathcal{X},\mathcal{X})$ *is a CQS-Pair.*

Theorem 6. *Cyclic Quorum System Pair (CQS-Pair) is a solution to the h-QPS problem.*

Proof. According to the definition of CQS-Pair, it satisfies the heterogeneous rotation closure property. So the CQS-Pair can be a solution to the h-QPS problem according to Lemma 1.

Consider an example where $A = \{1, 2, 4\}$ and $\mathcal{X} = C(A, \mathbb{Z}_7)$, $B = \{7, 9, 14, 15, 18\}$ and $\mathcal{Y} = C(B, \mathbb{Z}_{21})$. The pair $(\mathcal{X}, \mathcal{Y})$ is a CQS-Pair. But if $A = \{3, 5, 6\}$ and $B = \{7, 9, 14, 15, 18\}$, the pair $(\mathcal{X}, \mathcal{Y})$ is NOT a CQS-Pair, although $\mathcal{X}$ and $\mathcal{Y}$ are cyclic quorum systems, respectively.

If $A = \{1, 2, 4\}$ and $\mathcal{X} = C(A, \mathbb{Z}_7)$, $B = \{1, 2, 4\}$ in $(\mathbb{Z}_{14}, +)$ and $\mathcal{Y} = C(B, \mathbb{Z}_{14})$, (A, B) is a $(7, 3, 14, 3)$-difference pair. But $(\mathcal{X}, \mathcal{Y})$ is NOT a CQS-Pair since B is not a relaxed difference set in $(\mathbb{Z}_{14}, +)$ and $\mathcal{Y}$ is not a cyclic quorum system.

3.4 Constructing Scheme for Cyclic Quorum System Pair

In previous work, exhaustive search has been used to find an optimal solution for cyclic quorum design [6]. This is not practical when cycle length (n) is large. In this section, we first present a fast construction scheme for cyclic quorum systems and then apply it to the design of CQS-Pair. First, we define a few concepts.

Definition 13. ***Automorphism.*** *Suppose $(X, \mathcal{A})$ is a design. A transform function α is an automorphism of $(X, \mathcal{A})$ if*

$$[\{\alpha(x) : x \in A\} : A \in \mathcal{A}] = \mathcal{A}$$

Definition 14. ***Disjoint cycle representation:*** *The disjoint cycle representation on a set X is a group of disjointing cycles in which each cycle has the form $(x\ \alpha(x)\ \alpha(\alpha(x)) \cdots)$ for some $x \in X$.*

Suppose the automorphism is $x \mapsto 2x \ mod\ 7$. The disjoint cycle representation of $\mathbb{Z}_7$ is as follows: (0) (1 2 4) (3 6 5).

Definition 15. *Let D be a (v, k, λ)-difference set in $(\mathbb{Z}_v, +)$. For an integer m, let $mD = \{mx : x \in D\}$. Then m is called a multiplier of D if $mD = D + g$ for some $g \in \mathbb{Z}_v$. Also, we say that D is fixed by the multiplier m if $mD = D$.*

Theorem 7. ***(Multiplier Theorem).*** *Suppose there exists a (v, k, λ)-difference set D. Suppose also that the following four conditions are satisfied:*

1. *p is* prime*;*
2. *$gcd(p, v) = 1$;*
3. *$k - \lambda \equiv 0\ (mod\ p)$; and*
4. *$p > \lambda$.*

Then p is a multiplier of D.

The proof of Theorem 7 is given in [8]. According to the Theorem of Singer Difference Set, there exists a $(q^2+q+1, q+1, 1)$-difference set when q is a prime power. So the Multiplier Theorem does not guarantee the existence of a (v, k, λ)-difference sets for any integer v. We only consider the $(q^2+q+1, q+1, 1)$-design, where q is a prime power, in our construction scheme.

We first give an example to illustrate the application of the Multiplier Theorem for the construction of difference sets.

Example. We use the Multiplier Theorem to find a $(21, 5, 1)$-difference set. Observe that $p = 2$ satisfies the conditions of Theorem 7. Hence 2 is a multiplier of any such difference set. Therefore, the *automorphism* is $\alpha(x) \mapsto 2x \; mod \; 21$. Thus, we get the disjoint cycle representation of the permutation defined by $\alpha(x)$ of $\mathbb{Z}_{21}$ as follows:

$$(0) \quad (1\ 2\ 4\ 8\ 16\ 11) \quad (3\ 6\ 12) \quad (5\ 10\ 20\ 19\ 17\ 13) \quad (7\ 14) \quad (9\ 18\ 15)$$

The difference set we are looking for must consist of a union of cycles in the list above. Since the difference set has size five, it must be the union of one cycle of length two and one cycle of length three. There are two possible ways to do this, both of which happen to produce the difference set:

$$(3\ 6\ 7\ 12\ 14) \; and \; (7\ 9\ 14\ 15\ 18)$$

With the Multiplier Theorem, we can quickly construct $(q^2+q+1, q+1, 1)$-difference sets, where q is a prime power. The mechanism can significantly improve the speed of finding the optimal solution relative to the exhaustive method in [6]. After obtaining the difference sets, we use Theorem 5 to build a CQS-pair.

To check the non-empty intersection property of two heterogeneous difference sets $A = \{a_1, a_2, \cdots, a_k\}$ in $(\mathbb{Z}_N, +)$ and $B = \{b_1, b_2, \cdots, b_l\}$ in $(\mathbb{Z}_M, +)$ where $N \leq M$ and $p = \lceil \frac{M}{N} \rceil$, let's define a $pk \times l$ matrix $\mathcal{M}_{l \times pk}$ whose element $m_{i,j}$ is equal to $(b_i - a_j^{'})$ mod M where $a_j^{'} \in A^p$. We can check whether (A, B) is a (N, k, M, l)-difference pair by checking if $\mathcal{M}_{l \times pk}$ contains all elements from 0 to $M-1$. We call $\mathcal{M}_{l \times pk}$ a *verification matrix.*

Since we only consider $(q^2+q+1, q+1, 1)$-design, we describe our algorithm for constructing a CQS-Pair as follows:

Step 1: Given two input integers $n, m (n \leq m)$, find out two prime power q and r which satisfy $n = q^2+q+1$ and $m = r^2+r+1$. Set $k \leftarrow q+1$ and $l \leftarrow r+1$.

Step 2: Obtain the Multiplier p_a for $(n, k, 1)$-difference set, and p_b for $(m, l, 1)$-difference set; then set *automorphisms* $\alpha_n(x) \leftarrow p_a \cdot x \; (mod \; n)$ and $\alpha_m(x) \leftarrow p_b \cdot x \; (mod \; m)$;

Step 3: Construct the disjoint cycle presentation for $\mathbb{Z}_n$ with $\alpha_n(x)$, and the disjoint cycle presentation for $\mathbb{Z}_m$ with $\alpha_m(x)$, respectively;

Step 4: Suppose there are u unions of disjoint cycle being $(n, k, 1)$-difference set, and v unions of disjoint cycle being $(m, l, 1)$-difference set. Construct the *verification matrices* for all $u \times v$ pairs of cyclic quorum systems $(\mathcal{X}, \mathcal{Y})$.

Step 5: For all $u \times v$ *verification matrices*, check whether it contains all elements from 1 to m. If *true*, $(\mathcal{X}, \mathcal{Y})$ is a CQS-Pair, otherwise, it is not a CQS-Pair.

By employing our constructing algorithm, for two different integer n and m satisfying $n = q^2 + q + 1$ and $m = r^2 + r + 1$ (q and r being two prime powers, $n \leq m$), it will take $O(n^2)$ and $O(m^2)$ time for them to build the *disjoint cycle representation* respectively. After that, Step 5 in the algorithm will check $u \times v \times l \times pk \approx uvm^{3/2}n^{-1/2}$ elements since $l \approx \sqrt{m}$ and $k \approx \sqrt{n}$, where u and v are numbers of $(n, k, 1)$-difference sets and $(m, l, 1)$-difference sets, respectively. So the total time complexity is $O(uvm^{3/2}n^{-1/2} + m^2)$ for constructing a CQS-Pair with input parameters n and m $(n \leq m)$.

3.5 A Complete Application Example

As an example, consider $n = 7$, $m = 21$. By Multiplier Theorem, we can easily obtain two $(7, 3, 1)$-difference sets being $\{1, 2, 4\}$ and $\{3, 6, 5\}$ in $(\mathbb{Z}_7, +)$. Similarly, there are two $(21, 5, 1)$-difference sets, $\{3, 6, 7, 12, 14\}$ and $\{7, 9, 14, 15, 18\}$ in $(\mathbb{Z}_{21}, +)$. Let $A_7 = \{1, 2, 4\}$, $B_7 = \{3, 6, 5\}$, $A_{21} = \{3, 6, 7, 12, 14\}$, and $B_{21} = \{7, 9, 14, 15, 18\}$.

Totally, there are four pairs of $(7, 3, 1)$-difference sets and $(21, 5, 1)$-difference sets. First, we check the pair $(C(A_7, \mathbb{Z}_7), C(A_{21}, \mathbb{Z}_{21}))$. The constructed verification matrix is as follows.

$$\begin{bmatrix} 2 & 1 & 20 & 16 & 15 & 13 & 9 & 8 & 6 \\ 5 & 4 & 2 & 19 & 18 & 16 & 12 & 11 & 9 \\ 6 & 5 & 3 & 20 & 19 & 17 & 13 & 12 & 10 \\ 11 & 10 & 8 & 4 & 3 & 1 & 18 & 17 & 15 \\ 13 & 12 & 10 & 6 & 5 & 3 & 20 & 19 & 17 \end{bmatrix}$$

We find that 7 and 14 are not in the matrix. So the pair $(C(A_7, \mathbb{Z}_7), C(A_{21}, \mathbb{Z}_{21}))$ is NOT a CQS-Pair. Similarly, we can check that $(C(B_7, \mathbb{Z}_7), C(B_{21}, \mathbb{Z}_{21}))$ is NOT a CQS-Pair. But $(C(A_7, \mathbb{Z}_7),$
$C(B_{21}, \mathbb{Z}_{21}))$ and $(C(B_7, \mathbb{Z}_7), C(A_{21}, \mathbb{Z}_{21}))$ are a CQS-Pair, respectively.

The CQS-Pair can be applied to sensor networks for dynamically changing the quorum system (i.e., the cycle length) in each node, in order to meet the end-to-end delay constraint without losing connectivity. Table 1 shows the available pairs for cycle lengths ≤ 21.

4 Performance Analysis

4.1 Average Delay

We denote the average delay as the time between data arrival and discovery of the adjacent receiver (two nodes wake-up simultaneously). Note that this metric

Table 1. CQS-Pair (for $n, m \leq 21$)

cycle length	7 $A_7 = \{1, 2, 4\}$ $B_7 = \{3, 5, 6\}$	13 $A_{13} = \{0, 1, 3, 9\}$ $B_{13} = \{0, 2, 6, 5\}$ $C_{13} = \{0, 4, 12, 10\}$ $D_{13} = \{0, 7, 8, 11\}$	21 $A_{21} = \{3, 6, 7, 12, 14\}$ $B_{21} = \{7, 9, 14, 15, 18\}$
7	$(C(A_7, \mathbb{Z}_7), C(A_7, \mathbb{Z}_7))$ $(C(B_7, \mathbb{Z}_7), C(B_7, \mathbb{Z}_7))$	$(C(A_7, \mathbb{Z}_7), C(A_{13}, \mathbb{Z}_{13}))$ $(C(A_7, \mathbb{Z}_7), C(B_{13}, \mathbb{Z}_{13}))$ $(C(B_7, \mathbb{Z}_7), C(C_{14}, \mathbb{Z}_{13}))$ $(C(B_7, \mathbb{Z}_7), C(D_{14}, \mathbb{Z}_{13}))$	$(C(A_7, \mathbb{Z}_7), C(B_{21}, \mathbb{Z}_{21}))$ $(C(B_7, \mathbb{Z}_7), C(A_{21}, \mathbb{Z}_{21}))$
13		$(C(A_{13}, \mathbb{Z}_{13}), C(A_{13}, \mathbb{Z}_{13}))$ $(C(B_{13}, \mathbb{Z}_{13}), C(B_{13}, \mathbb{Z}_{13}))$ $(C(C_{13}, \mathbb{Z}_{13}), C(C_{13}, \mathbb{Z}_{13}))$ $(C(D_{13}, \mathbb{Z}_{13}), C(D_{13}, \mathbb{Z}_{13}))$	$(C(B_{13}, \mathbb{Z}_{13}), C(A_{21}, \mathbb{Z}_{21}))$

does not include the time for delivering a message. And suppose the length of one time slot being 1.

Theorem 8. *The average delay between two nodes wakeup based on quorums from the same Cyclic Quorum System adopting the $(n, k, 1)$-difference set is $E(Delay) = \frac{n-1}{2}$.*

Proof. Let the k elements in $(n, k, 1)$-difference set be denoted as a_1, a_2, $\cdots$, a_k. If a node has a message arrived during the i^{th} time slot, the expected delay (from data arrival to two nodes wake-up simultaneously) is $\frac{1}{k}(a_1 - i)\ mod\ n + \frac{1}{k}(a_2 - i)\ mod\ n + \cdots + \frac{1}{k}(a_k - i)\ mod\ n$. If a message has arrived, the probability of the message arriving at the i^{th} time slot is $\frac{1}{n}$. Thus, the expected delay (average delay) is:

$$\begin{aligned} E(Delay) = & \frac{1}{n}[\frac{1}{k}(a_1 - 1)\ mod\ n + \frac{1}{k}(a_2 - 1)\ mod\ n + \cdots + \frac{1}{k}(a_k - 1)\ mod\ n \\ & + \frac{1}{k}(a_1 - 2)\ mod\ n + \frac{1}{k}(a_2 - 2)\ mod\ n + \cdots + \frac{1}{k}(a_k - 2)\ mod\ n \\ & + \quad \cdots\cdots \\ & + \frac{1}{k}(a_1 - n)\, mod\ n + \frac{1}{k}(a_2 - n)\, mod\ n + \cdots + \frac{1}{k}(a_k - n)\ mod\ n] \\ = & \frac{1}{nk} \cdot (k \cdot 1 + k \cdot 2 + \cdots + k \cdot n - 1) = \frac{n-1}{2} \end{aligned}$$

Corollary 2. *The average delay between two nodes wakeup based on a CQS-Pair in which two cyclic quorum systems have cycle length n and m $(n \leq m)$ respectively, is:*

$$\frac{n-1}{2} < E(Delay) < \frac{m-1}{2}$$

Corollary 2 indicates that the average delay between two nodes adopting CQS-Pair is bounded. When the average one-hop delay constraint is D, we must meet $\frac{m-1}{2} \leq D$.

4.2 Quorum Ratio and Energy Conservation

We define **quorum ratio** (ϕ) as the proportion of the beacon intervals that is required to be awake in each cycle. If the wakeup schedule in a node is a cyclic quorum system which is based on a $(n, k, 1)$-difference set, its quorum ratio is $\frac{k}{n}$. It has been proved that a $(q^2 + q + 1, q + 1, 1)$-difference set exists and is an optimal design for a given quorum size $q + 1$ [13]. The optimal quorum ratio is $\phi = \frac{q+1}{q^2+q+1}$ for such a cyclic quorum system.

For a CQS-Pair, the quorum ratios for systems in the pair which are based on (N, k, M, l)-difference pair are $\phi_1 = \frac{\sqrt{4N-3}-1}{2N}$ and $\phi_2 = \frac{\sqrt{4M-3}-1}{2M}$ respectively, when we only consider $(q^2 + q + 1, q + 1, 1)$-design in the CQS-Pair constructing scheme.

Note that a $\sqrt{n} \times \sqrt{n}$ grid quorum system also satisfies the heterogeneous rotation closure property and can be applied as a solution to the h-QPS problem. The quorum ratio is $\phi = \frac{2\sqrt{n}-1}{n}$ for a grid quorum system.

The energy conservation ratio is correlated with the quorum ratio. If the quorum ratio is ϕ, the energy conservation ratio for a node is $1 - \phi$.

4.3 Multicast/Broadcast Support

The quorum-based asynchronous wakeup protocols cannot guarantee that more than one receiver is awake when the transmitter requests to transmit a multicast/broadcast message.

There are multiple ways to support multicast/broadcast. One method is to adopt relatively prime frequencies among all nodes for wakeup scheduling. This method does not need synchronization between the transmitter and all receivers. The transmitter only needs to notify m receivers to wake up via the pairwise relative primes p_1, p_2,..., p_m, respectively. Then each receiver generates its new wakeup frequency based on the received frequency. Through Chinese Remainder Theorem, it can be proven that the m receivers must wakeup simultaneously at the I^{th} beacon interval $(0 \leq I \leq p_1 \times p_2 ... \times p_m)$. The transmitter can then transmit a multicast/broadcast message at this interval.

Another way to multicast/broadcast is by using synchronization over quorum-based wakeup schedule. The transmitter can book-keep all neighbors' schedules, and synchronize their schedules so that neighboring nodes wake up in the same set of slots with the use of Lamport's clock synchronization algorithm [5]. When all nodes are awake simultaneously, the transmitters then send a message to multiple neighbors simultaneously.

The first mechanism has the advantage that no synchronization is needed between transmitter and multiple receivers. But it cannot bound the average delay. The second approach can bound the average delay but it needs bookkeeping and synchronization over asynchronous wakeup schedules.

5 Past and Related Work

As mentioned in the Section 1, there are three categories of wakeup mechanism for wireless sensor networks. We summarize them and overview past and related work in asynchronous wakeup scheduling.

On-Demand Wakeup. The implementation of on-demand wakeup schemes typically requires two different channels: a data channel and a wakeup channel for awaking nodes when needed. This allows for not deferring the transmission of signal on the wakeup channel if a packet transmission is in progress on the other channel, thus reducing the wakeup latency. The drawback is the additional cost for the second radio. STEM (Sparse Topology and Energy Management) [7] uses two different radios for wakeup signals and data packet transmissions, respectively. The key idea is that a node remains awake until it has not received any message destined for it for a certain time. STEM uses separate control and data channels, and hence the contention among control and data messages is alleviated. The energy efficiency of STEM is dependent on that of the control channel.

Scheduled Rendezvous Schemes. These schemes require that all neighboring nodes wake up at the same time. Different scheduled rendezvous protocols differ in the way network nodes sleep and wakeup during their lifetime. The simplest way is using a Fully Synchronized Pattern, like S-MAC [12]. In this case, all nodes in the network wakeup at the same time according to a periodic pattern. A further improvement can be achieved by allowing nodes to switch off their radio when no activity is detected for at least a timeout value, like that in T-MAC [2]. The disadvantages are the complexity and overhead for synchronization.

Asynchronous wakeup. This was first introduced in [11] with reference to IEEE 802.11 ad hoc networks. The authors proposed three different asynchronous sleep/wakeup schemes that require some modifications to the basic IEEE 802.11 Power Saving Mode (PSM). More recently, Zheng et al. [13] took a systematic approach to design asynchronous wakeup mechanisms for ad hoc networks (applicable for wireless sensor networks as well). They formulate the problem of generating wakeup schedules as a block design problem and derive theoretical bounds under different communication models. The basic idea is that each node is associated with a Wakeup Schedule Function (WSF) that is used to generate a wakeup schedule. For two neighboring nodes to communicate, their wakeup schedules have to overlap regardless of the difference in their clocks.

For the quorum-based asynchronous wakeup design, Luk and Wong [6] designed a cyclic quorum system using difference sets. But they do exhaustive search to obtain a solution for each cycle length N, which is impractical when N is large.

Asymmetric quorum design. In the clustered environment of sensor networks, it is not always necessary to guarantee all-pair neighbor discovery. The Asymmetric Cyclic Quorum (ACQ) system [10] was proposed to guarantee neighbor discovery between each member node and the clusterhead, and between clusterheads in a network. The authors also presented a construction scheme which assembles the ACQ system in $O(1)$ time to avoid exhaustive searching. ACQ is a generalization of the cyclic quorum system. The scheme is configurable for different networks to achieve different distribution of energy consumption between member nodes and the clusterhead.

However, it remains a challenging issue to efficiently design an asymmetric quorum system given an arbitrary value of n. One previous study [13] shows that the problem of finding an optimal asymmetric block design can be reduced to the minimum vertex cover problem, which is NP-complete. However, the ACQ [10] construction is not optimal in that the quorum ratio for symmetric-quorum is $\phi = \lceil \frac{n+1}{2} \rceil$ and the quorum ratio for asymmetric-quorum is $\phi^{'} = \lceil \sqrt{\frac{n+1}{2}} \rceil$. The another drawback is that it cannot be a solution to the h-QPS problem since the two asymmetric-quorums cannot guarantee the intersection property.

Transport layer approach. Wang et al. [9] applied quorum based wakeup schedule at the transport layer which can cooperate with any MAC layer protocol, allowing for the reuse of well-understood MAC protocols. The proposed technique saves idle energy by relaxing the requirement for end-to-end connectivity during data transmission and allowing the network to be disconnected intermittently via scheduled sleeping. The limitation of this work is that each node adopts same grid quorum systems as wakeup scheduling and the quorum ratio is not optimal comparing with that of cyclic quorum systems.

6 Conclusions

This paper presents a theoretical approach for heterogeneous asynchronous wakeup scheduling in wireless sensor networks. We defined the h-QPS problem—i.e., given two cycle lengths n and m ($n < m$), how to design a pair of heterogeneous quorum systems to guarantee that two adjacent nodes picking up heterogenous quorums from the pair as their wakeup schedule can hear each other at least once in every m consecutive time slots. We defined the Cyclic Quorum System Pair (CQS-Pair) which can be applied as a solution to h-QPS problem. We also presented a fast construction scheme to assemble a CQS-Pair. In our construction scheme, we first quickly construct $(n, k, 1)$-difference set and $(m, l, 1)$-difference set. Based two difference sets A in $(\mathbb{Z}_n, +)$ and B in $(\mathbb{Z}_m, +)$, we can construct a CQS-Pair $(C(A, \mathbb{Z}_n), C(B, \mathbb{Z}_m))$ when A and B can form a (n, k, m, l)-difference pair.

The performance of a CQS-Pair was analyzed in terms of average delay, quorum ratio, and issues for supporting multicast/broadcast. We show that the average delay between two node wakeup via heterogenous quorums from a CQS-Pair is bounded between $\frac{n-1}{2}$ and $\frac{m-1}{2}$, and the quorum ratios of the two quorum systems in the pair are optimal respectively given their cycle lengthes n and m.

There are several directions for future work. One direction is to find a scheme to check the existence of a CQS-Pair for any given n and m. Another example is to extend the CQS-Pair to CQS m-pair in which m cyclic quorum systems have the heterogenous rotation closure property with one another.

Acknowledgment

This work was supported by research project grants from MKE/MND, South Korea.

References

1. Blum, J., Ding, M., Thaeler, A., Cheng, X.: Connected dominating set in sensor networks and manets. In: Handbook of Combinatorial Optimization 2005, pp. 329–369 (2005)
2. Dam, T., Langendoen, K.: An adaptive energy-efficient mac protocol for wireless sensor networks. In: The First ACM Conference on Embedded Networked Sensor Systems (Sensys 2003) (2003)
3. Feeney, L.M., Nilsson, M.: Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In: IEEE Conference on Computer Communications (INFOCOM 2001), pp. 1548–1557 (2001)
4. Jiang, C.H.J.R., Tseng, Y.C., Lai, T.: Quorum-based asynchronous power-saving protocols for ieee 802.11 ad hoc networks. ACM Journal on Mobile Networks and Applications (2005)
5. Lamport, L.: Time,clocks, and the ordering of events in a distributed system. Communications of the ACM 21 (1978)
6. Luk, W., Huang, T.: Two new quorum based algorithms for distributed mutual exclusion. In: Proceedings of the International Conference on Distributed Computing Systems (ICDCS 1997), pp. 100–106 (1997)
7. Schurgers, C., Tsiatsis, S.G.V., Srivastava, M.: Topology management for sensor networks: Exploiting latency and density. In: ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2002) (2002)
8. Stinson, D.R.: Combinatorial Designs: Constructions and Analysis. Springer, Heidelberg (2003)
9. Wang, Y., Wan, C., Martonosi, M., Peh, L.: Transport layer approaches for improving idle energy in challenged sensor networks. In: Proceedings of the 2006 SIGCOMM workshop on Challenged networks (SIGCOMM 2006 Workshops), pp. 253–260 (2006)
10. Wu, S.H., Chen, C.M., Chen, M.S.: An asymmetric quorum-based power saving protocol for clustered ad hoc networks. In: Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS 2007) (2007)
11. Tseng, C.H.Y., Hsieh, T.: Power-saving protocols for ieee 802.11-based multi-hop ad hoc networks. In: IEEE Conference on. Computer Communications (INFOCOM 2002), pp. 200–209 (2006)
12. Ye, W., Heidemann, J., Estrin, D.: Medium access control with coordinated adaptive sleeping for wireless sensor networks. IEEE/ACM Transactions on Networking 12, 493–506 (2004)
13. Zheng, R., Hou, J.C., Sha, L.: Asynchronous wakeup for ad hoc networks. In: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking and computing (MobiHoc 2003), pp. 35–45 (2003)

Impact of Information on the Complexity of Asynchronous Radio Broadcasting*

Tiziana Calamoneri[1], Emanuele G. Fusco[1], and Andrzej Pelc[2,**]

[1] Computer Science Department, University of Rome "Sapienza", Roma, Italy
{calamo,fusco}@di.uniroma1.it
[2] Département d'informatique, Université du Québec en Outaouais, Gatineau,
Québec J8X 3X7, Canada
pelc@uqo.ca

Abstract. We consider asynchronous deterministic broadcasting in radio networks. An execution of a broadcasting protocol is a series of events, each of which consists of simultaneous transmitting or delivering of messages. The aim is to transmit the source message to all nodes of the network. If two messages are delivered simultaneously to a node, a collision occurs and this node does not hear anything. An asynchronous adversary may delay message deliveries, so as to create unwanted collisions and interfere with message dissemination. The total number of message transmissions executed by a protocol in the worst case is called the *work* of the protocol, and is used as the measure of its complexity. The aim of this paper is to study how various types of information available to nodes influence the optimal work of an asynchronous broadcasting protocol. This information may concern past events possibly affecting the behavior of nodes (adaptive vs. oblivious protocols), or may concern the topology of the network or some of its parameters. We show that decreasing the knowledge available to nodes may cause exponential increase of work of an asynchronous broadcasting protocol, and in some cases may even make broadcasting impossible.

1 Introduction

1.1 Radio Networks and Asynchronous Adversaries

A *radio network* consists of stations with transmitting and receiving capabililities. The network is modeled as a directed graph with a distinguished node called the *source*. Each node has a distinct identity (label) which is a positive integer. If there is a directed edge from u to v, node v is called an *out-neighbor* of u and

* Partially supported by the European Union under the Project *Algorithmic Principles for Building Efficient Overlay Computers (AEOLUS)*. This work was done during the visit of Andrzej Pelc at the University of Rome "Sapienza".

** Partially supported by the Visiting Fellowship from the University of Rome "Sapienza", by NSERC discovery grant, and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 311–330, 2008.

u is called an *in-neighbor* of v. At some time t a node may send a message to all of its out-neighbors. It is assumed that this message is delivered to all the out-neighbors simultaneously at some time $t' > t$ decided by an adversary that models unpredictable asynchronous behavior of the network. The only constraint (cf. [1,2]) is that the adversary cannot collapse messages coming from the same node, i.e., two distinct messages sent by the same node have to be delivered at different times. We consider two types of asynchronous adversaries. The *strong* adversary, called the *node adversary* in [1], may choose an arbitrary delay $t' - t$ between sending and delivery, possibly different for every message. The *weak* adversary chooses an arbitrary delay for a given node (possibly different delays for different nodes), but must use this delay for all messages sent by this node during the protocol execution. The motivation for both adversaries is similar and follows the one given in [1]. Nodes of a radio network execute a communication protocol while concurrently performing other computation tasks. When a message arrives at a node, it is stored (prepared for transmission) and subsequently transmitted by it, the (unknown) delay between these actions being decided by the adversary; storing for transmission corresponds to sending and actual transmission corresponds to simultaneous delivery to all out-neighbors (at short distances between nodes the travel time of the message is negligible). The delay between storing and transmitting (in our terminology, between sending and delivery) depends on how busy the node is with other concurrently performed tasks. The strong adversary models the situation when the task load of nodes may vary during the execution of a broadcast protocol, and thus delay may vary from message to message even for the same node. The weak adversary models the assumption of a constant occupation load of each node during the communication process: some nodes may be more busy than others but the delay for a given node is constant.

At time t', a message is *heard*, i.e., received successfully by a node, if and only if, a message from exactly one of its in-neighbors is delivered at this time. If messages from two in-neighbors v and v' of u are delivered simultaneously at time t', we say that a *collision* occurs at u. Similarly as in most of the literature concerning algorithmic aspects of radio communication, we assume that in this case u does not hear anything at time t', i.e., we assume that a node cannot distinguish collision from silence.

While in general the network is modeled as an arbitrary directed graph, we also consider two natural smaller classes of networks. The first is modeled by *symmetric* directed graphs, or equivalently by undirected graphs. The second, still smaller class of networks is modeled by *unit disk graphs* (UDG) whose nodes are the stations. These nodes are represented as points in the plane. In the case of UDG networks, each node knows its Euclidean coordinates in the plane. These coordinates also play the role of the label (similarly as, e.g., in [3,4], nodes in UDG networks are not equipped with integer identities). Two nodes are joined by an (undirected) edge if their Euclidean distance is at most 1. Such nodes are called *neighbors*. It is assumed that transmitters of all stations have equal power which enables them to transmit at Euclidean distance 1, and that communication proceeds in a flat terrain without large obstacles. Hence the existence of an edge between two nodes indicates that transmissions of one of them can reach the other, i.e., these nodes can communicate directly. By contrast, arbitrary directed graphs are an appropriate model for radio networks deployed in a terrain with large obstacles and possibly varying power of transmitting devices.

1.2 Centralized vs. Ad Hoc Broadcasting

We consider *broadcasting*, which is one of the basic communication primitives. In the beginning, one distinguished node, called the *source*, has a message which has to be transmitted to all other nodes. Remote nodes get the source message via intermediate nodes, along paths in the network. We assume that only stations that have already received the source message can send messages, hence broadcasting is equivalent to a process of waking up the network, when at the beginning only the source is awake. In order for the broadcasting to be feasible, we assume that there is a directed path from the source to any other node. For symmetric networks this is equivalent to connectivity. In this paper we consider only deterministic broadcasting algorithms.

Two alternative assumptions are made in the literature concerning broadcasting algorithms. It is either assumed that the topology of the underlying graph is known to all nodes, in which case nodes can simulate the behavior of a central monitor scheduling transmissions (*centralized broadcasting*), or it is assumed that the network topology is unknown to nodes (*ad hoc broadcasting*). Moreover, in the latter case, some crucial parameters of the network, such as the number n of nodes, may be known or unknown to nodes. In the case of UDG radio networks, an important parameter is the *density* d of the network, i.e., the smallest Euclidean distance between any two stations. We will see how information about the topology of the network and knowledge of its parameters influence the efficiency of broadcasting protocols. In particular, for UDG networks, optimal work of broadcasting protocols may depend on the *granularity* g of the network defined as the inverse of its density.

1.3 Adaptive vs. Oblivious Protocols

We consider two kinds of broadcasting protocols: oblivious and adaptive. In an oblivious protocol every node has to send all its messages as soon as it is woken up by the source message. More precisely, a node has to commit to a non-negative integer representing the number of messages it will send during the broadcasting process, prior to the execution of the protocol. This number may depend only on the label of the node or on its position in the case of UDG networks. (In [1] only oblivious protocols were considered.) By contrast, an adaptive protocol is more powerful, as a node can decide on the number and content of messages it sends, depending on its history, i.e., depending on the sequence of messages received so far. Hence, while the total number of messages sent by an oblivious protocol is the same for each of its executions, for an adaptive protocol this number may differ depending on the behavior of the adversary.

We define the *work* of a broadcasting protocol as the worst-case total number of messages sent until all nodes are informed. The worst case is taken over all possible behaviors of an asynchronous adversary under consideration. Work is a natural measure of complexity of an asynchronous radio broadcast protocol. It was introduced in [1] for oblivious protocols. We will see that in some cases the rigidity of oblivious protocols may cause exponential increase of their work as compared to adaptive ones.

1.4 Our Results

In the first part of the paper (Sections 3-5) we present our results on optimal work of asynchronous broadcasting against the strong adversary (i.e., the node adversary from [1]), see Table 1.

For UDG networks with known topology we get a tight result: the optimal work is $\Theta(\tau)$, where τ is the number of blocks containing at least one node. (Blocks form a partition of the plane into disjoint squares of side $1/\sqrt{2}$ – see Sect. 3 for a precise definition.) The result holds both for adaptive and for oblivious algorithms. Our upper bound is constructive: we show an oblivious broadcasting algorithm with work $O(\tau)$. For UDG networks with unknown topology the results significantly change and they depend on whether (a lower bound on) the density d of the network is known or not. If it is known, then optimal work depends on the number τ of occupied blocks and on the granularity $g = 1/d$. We show an oblivious broadcasting algorithm with work $O(\tau\alpha^{g^2})$, for some constant $\alpha > 1$. On the other hand, we show that any broadcasting algorithm, even adaptive, must use work $\Omega(\tau\beta^{g^2})$, for some constant $\beta > 1$. If d is unknown, we show that broadcasting against the strong adversary is impossible in UDG networks.

We now summarize our results for networks modeled by graphs that need not come from configurations of points in the plane. (For such networks we assume that all nodes have distinct positive integer labels and each node knows its label.) Symmetric radio networks with known topology are those in which optimal work of asynchronous broadcasting significantly depends on the adaptivity of the algorithm. Indeed, we prove that for adaptive algorithms the optimal work is $\Theta(n)$, where n is the number of nodes in the network. The upper bound is again constructive: we show an adaptive broadcasting algorithm with work $O(n)$ working for any n-node symmetric network of known topology. By contrast, using techniques from [1], it can be proved that any oblivious algorithm uses work $\Omega(c^n)$, for some constant $c > 1$, on some symmetric n-node network, and that there exists an oblivious algorithm working for any symmetric n-node network of known topology, using work $O(2^n)$. Hence we prove an exponential gap between optimal work required by adaptive and by oblivious broadcasting in symmetric networks of known topology. It should be noted that for arbitrary (not necessarily symmetric) networks, broadcasting with linear or even polynomial work is not always possible, even for adaptive algorithms. Indeed, it follows from [1] that exponential work (in the number n of nodes) is needed for some networks, even when the topology is known and the algorithm is adaptive. It is also shown in [1] that, for radio networks of known topology, work $O(2^n)$ is always enough.

Table 1. Optimal work of broadcasting against the strong asynchronous adversary. τ is the number of non-empty tiles, n is the number of nodes, N is the maximal label and g is the granularity of the UDG network ($g = 1/d$); c, α and β are constants.

	UDG networks	Symmetric Networks	Arbitrary Networks
known topology	adaptive or oblivious: $\Theta(\tau)$	adaptive: $\Theta(n)$ oblivious [1]: $O(2^n)$ $\Omega(c^n)$, for some $c > 1$	adaptive or oblivious [1]: $O(2^n)$ $\Omega(c^n)$, for some $c > 1$
unknown topology	known density d adaptive or oblivious: $O(\tau\alpha^{g^2})$, for some $\alpha > 1$ $\Omega(\tau\beta^{g^2})$, for some $\beta > 1$	adaptive or oblivious: known or unknown N: $\Theta(2^N)$	
	unknown density d adaptive or oblivious: impossible		

For networks of unknown topology we have a tight result on optimal work of asynchronous broadcasting. This work is $\Theta(2^N)$, where N is the maximal label of a node, and this result does not depend on whether the networks are symmetric or not, whether the algorithm is adaptive or not, and whether the maximal label N is known to nodes or not. More precisely, we show a lower bound $\Omega(2^N)$ on the required work, even for symmetric networks with known parameter N, and even for adaptive algorithms. On the other hand, we observe that an (oblivious) algorithm described in [1] and working for arbitrary networks without using the knowledge of N has work $O(2^N)$.

In Sect.6 we present our results on optimal work of asynchronous broadcasting against the weak adversary. Introducing this adversary was motivated by the following remark in [1]: "It would be interesting to define a weaker, but still natural, model of asynchrony in radio networks, for which polynomial-work protocols always exist." We show that if nodes are equipped with clocks, then oblivious broadcasting algorithms using work $O(n)$ for n-node networks can always be provided in the presence of the weak asynchronous adversary. This is optimal, as witnessed by the example of the line network. Local clocks at nodes need not be synchronized, we only assume that they tick at the same rate. In fact, even this assumption can be removed in most cases: our algorithm works even when the ratio of ticking rates between the fastest and the slowest clock has an upper bound known to all nodes. The exception is the case of UDG networks of unknown density (for which broadcasting against the strong adversary was proved impossible). In this special case, our algorithm against the weak adversary assumes the same ticking rate of all clocks and relies on the availability of an object obtained non-constructively: if this object is given to nodes, they can perform oblivious broadcasting with work $O(n)$.

1.5 Related Work

Algorithmic aspects of radio communication were mostly studied under the assumption that communication is synchronous and using time as a complexity measure of the algorithms. These results can be partitioned into two subareas. The first deals with centralized communication, in which nodes have complete knowledge of the network topology and hence can simulate a central monitor (cf. [5,6,7,8,9,10,11]). The second subarea assumes only limited (often local) knowledge of the topology, available to nodes of the network, and studies distributed communication in such networks with incomplete information.

The first paper to study deterministic centralized broadcasting in radio networks, assuming complete knowledge of the topology, was [6]. The authors also defined the graph model of radio network subsequently used in many other papers. In [7], an $O(D \log^2 n)$-time broadcasting algorithm was proposed for all n-node networks of diameter D. This time complexity was then improved to $O(D + \log^5 n)$ in [9], to $O(D + \log^4 n)$ in [8], to $O(D + \log^3 n)$ in [10], and finally to $O(D + \log^2 n)$ in [11]. The latter complexity is optimal. On the other hand, in [5] the authors proved the existence of a family of n-node networks of constant diameter, for which any broadcast requires time $\Omega(\log^2 n)$.

Investigation of deterministic distributed broadcasting in radio networks whose nodes have only local knowledge of the topology was initiated in [12]. The authors assumed that each node knows only its own label and labels of its neighbors. Several authors [13,14,15,16,17,18,19] studied deterministic distributed broadcasting in radio networks under an even weaker assumption that nodes know only their own label (but not labels of their neighbors). In [14] the authors gave a broadcasting algorithm working in time $O(n)$ for all n-node networks, assuming that nodes can transmit spontaneously, before getting the source message. A matching lower bound $\Omega(n)$ on deterministic broadcasting time was proved in [19] even for the class of networks of constant radius.

In [14,15,16,18] the model of directed graphs was used. The aim of these papers was to construct broadcasting algorithms working as fast as possible in arbitrary (directed) radio networks without knowing their topology. The currently fastest deterministic broadcasting algorithms for such networks have running times $O(n \log^2 D)$ [18] and $O(n \log n \log \log n)$ [20]. On the other hand, in [17] an $\Omega(n \log D)$ lower bound on broadcasting time was proved for directed n-node networks of radius D.

Randomized broadcasting algorithms in radio networks were studied in [12,18,21,22]. The authors do not assume that nodes know the topology of the network or that they have distinct labels. In [12] the authors constructed a randomized broadcasting algorithm running in expected time $O(D \log n + \log^2 n)$. In [21] it was shown that for any randomized broadcasting algorithm and parameters $D \leq n$, there exists an n-node network of diameter D requiring expected time $\Omega(D \log(n/D))$ to execute this algorithm. The lower bound $\Omega(\log^2 n)$ from [5], for some networks of radius 2, holds for randomized algorithms as well. A randomized algorithm working in expected time $O(D \log(n/D) + \log^2 n)$, and thus matching the above lower bounds, was presented in [22] (cf. also [18]).

Another model of radio networks is based on geometry. Stations are represented as points in the plane and the graph modeling the network is no more arbitrary. It may be a unit disk graph, or one of its generalizations, where radii of disks representing areas that can be reached by the transmitter of a node may differ from node to node [23], or reachability areas may be of shapes different than a disk [24,25]. Broadcasting in such geometric radio networks and some of their variations was considered, e.g., in [23,24,3,25,26,27]. The first paper to study deterministic broadcasting in arbitrary geometric radio networks with restricted knowledge of topology was [23]. The authors used several models, also assuming a positive knowledge radius, i.e., the knowledge available to a node, concerning other nodes inside some disk. In [3] the authors considered broadcasting in radio networks modeled by unit disk graphs. They studied two communication models: one called the spontaneous wake up model that allows transmissions of nodes that have not yet gotten the source message, and the other, called the conditional wake up model, in which only nodes that already obtained the source message can transmit.

Asynchronous radio broadcasting was considered, e.g., in [1,2]. In [1] the authors studied three asynchronous adversaries (one of which is the same as our strong adversary), and investigated centralized oblivious broadcasting protocols working in their presence. They concentrated on finding broadcast protocols and verifying correctness of such protocols, as well as on providing lower bounds on their work. In [2] attention was focused on anonymous radio networks. In such networks not all nodes can be reached by a source message. It was proved that no asynchronous algorithm unaware of network topology can broadcast to all reachable nodes in all networks.

2 Terminology and Preliminaries

A set S of positive integers is *dominated* if, for any finite subset T of S, there exists $t \in T$ such that t is larger than the sum of all $t' \neq t$ in T.

Lemma 1. *Let S be a finite dominated set and let k be its size. Then there exists $x \in S$ such that $x \geq 2^{k-1}$.*

Proof. The proof is by induction on the size k of S. If $k = 1$ then $2^0 = 1$ and the basis of induction holds.

If a set is dominated, all its subsets are dominated. By the inductive hypothesis every subset of S of size $i < k$ contains an element $x \geq 2^{i-1}$. It follows that arranging elements in S in increasing order we have $x_i \geq 2^{i-1}$, for $1 \leq i \leq k-1$. Then $\sum_{i=1}^{k-1} x_i \geq \sum_{i=1}^{k-1} 2^{i-1} = 2^{k-1} - 1$. As x_k is the largest element in S and S is dominated, we have $x_k \geq \sum_{i=1}^{k-1} x_i > 2^{k-1} - 1$, which proves the lemma. □

Any oblivious broadcasting algorithm is fully determined by the number of messages sent by each node of the network. This non-negative integer is called the *send number* of the node. For any execution of a broadcasting algorithm, a *transmitter* is a node that sends at least one message in this execution. Hence, for an oblivious algorithm, a transmitter is a node with positive send number. The following lemma is a consequence of Lemma 1 from [1].

Lemma 2. *Consider any oblivious broadcasting algorithm $\mathcal{A}$. Let u be a node in the network. Let T be the set of transmitters in the in-neighborhood of u. If at least one element in T is informed by $\mathcal{A}$ and the set of send numbers of T is dominated, then u is eventually informed by $\mathcal{A}$.*

3 UDG Radio Networks

We recall the tilings of the plane defined in [3] by means of three different grids. Each of the three grids is composed of atomic squares with generic name *boxes*. The first grid is composed of boxes called *tiles*, of side length $d/\sqrt{2}$, the second of boxes called *blocks*, of side length $1/\sqrt{2}$, and the third one of boxes called *5-blocks*, of side length $5/\sqrt{2}$. All grids are aligned with the coordinate axes, each box includes its left side without the top endpoint and its bottom side without the right endpoint. Each grid has a box with the bottom left point with coordinates $(0, 0)$. Let τ be the number of non-empty blocks (i.e., blocks which contain at least one node).

Tiles are small enough to ensure that only one node can belong to a tile. Blocks are squares with diameter 1, i.e., the largest possible squares such that each pair of nodes in a square are able to communicate. 5-blocks are used to avoid collisions during communication: messages originating from central blocks of disjoint 5-blocks cannot cause collisions.

Every 5-block contains 25 blocks, while every block contains $\Theta\left(g^2\right)$ tiles. Blocks inside a 5-block and tiles inside a block are numbered with consecutive integers (starting from 0) left to right, top to bottom. Hence every tile is assigned a pair of integers (i, j) where i is the block number in the 5-block and j is the tile number in the block. (Tiles lying in more than one block are assigned more than one such pair. This is the case when $\sqrt{2}/n \neq d$ for all n.)

We say that two (distinct) blocks are *potentially reachable* from each other if they contain points at distance ≤ 1. Two blocks are *reachable* from each other if they contain nodes at distance ≤ 1. There are exactly 20 blocks that are potentially reachable from any given block.

3.1 Known Topology

The following algorithm is oblivious, as it consists in an assignment of send numbers to nodes.

Algorithm `UDG1`
For any pair of blocks (B, B') that are reachable from each other, Algorithm `UDG1` elects a pair of transmitters (b, b') s.t. $b \in B$, $b' \in B'$, and b is at distance at most 1 from b'. Any fixed strategy (e.g., taking the smallest such pair in lexicographic order of positions) is suitable to perform the election. Notice that at most 20 transmitters can be elected in every block.

Each elected transmitter in a 5-block is assigned a distinct label from the set $\mathcal{L} = \{0, 1, \ldots, 499\}$. This is done by partitioning the set $\mathcal{L}$ into 25 sets L_i of 20 labels each (in an arbitrary but fixed manner). Transmitters in the i-th block of any 5-block are assigned labels from set L_i. Labels in each block are assigned to transmitters in increasing order according to lexicographic order of their positions.

Assignment of send numbers is done as follows: each elected transmitter with label i is assigned send number 2^i. If the source has not been elected, it is assigned send number 1. All other nodes are assigned send number 0. ■

Lemma 3. *Algorithm* `UDG1` *successfully performs broadcast in any UDG radio network of known topology, with work in $O(\tau)$.*

Proof. We first prove the correctness of the algorithm. As the network is connected, either $\tau = 1$ or, for any non-empty block B, there must exist a sequence of block pairs $\langle (S, X_1), (X_1, X_2), \ldots, (X_{k-1}, X_k), (X_k, B) \rangle$ such that S is the block containing the source and blocks in each pair are reachable from each other. If $\tau = 1$, all nodes in the unique non-empty block will be informed as soon as the message transmitted by the source is delivered, and algorithm `UDG1` successfully completes broadcasting with work 1. If $\tau > 1$, any non-empty block has at least one transmitter, and thus any node has a transmitter in its neighborhood. Moreover, every transmitter is connected to a transmitter located in S by a path containing only transmitters.

Consider an arbitrary node v and its block B, and consider the 5-block that has B in its center (this 5-block is not necessarily part of the 5-block grid). All neighbors of v are inside this 5-block. Blocks in this 5-block are assigned distinct numbers, and thus the set of send numbers assigned to transmitters in the neighborhood of v is dominated. It follows from Lemma 2 that node v will eventually receive the source message provided that at least one of the transmitters in its neighborhood will receive it. Hence it is enough to show that all transmitters receive the source message. This follows by induction on the length of a shortest path, in the subgraph induced by transmitters, between a transmitter in the block S and a transmitter in the neighborhood of v.

In order to estimate the work of the algorithm, notice that only a constant number of nodes in each block have a positive send number, and each send number is bounded by a constant. It follows that the total work is linear in the number τ of non-empty blocks. □

Lemma 4. *The work required to complete broadcast in any UDG radio network is in $\Omega(\tau)$.*

Proof. The proof follows from the fact that at least one node in every non empty 5-block has to transmit at least once. □

Lemma 3 and Lemma 4 imply the following theorem.

Theorem 1. *The optimal work required to complete broadcast in any UDG radio network of known topology is $\Theta(\tau)$.*

3.2 Unknown Topology

When the topology of the network is unknown, elections of transmitters cannot be performed without message exchanges. Here the scenario is different depending on whether (a lower bound on) the density d of the network is known or not.

The following algorithm assumes that each node is provided with the value of d. Similarly as Algorithm `UDG1` it is oblivious.

Algorithm `UDG2`
The algorithm is based on the tilings from [3] defined in the beginning of Sect. 3, and works in a similar manner as Algorithm `UDG1`. The set $\mathcal{L}$ of labels is now composed of integers from the interval $\left[0, \ldots, 25 \cdot \left(\lceil\sqrt{2}/d\rceil + 1\right)^2 - 1\right]$, and it is partitioned in 25 sets L_i, each of size $\left(\lceil\sqrt{2}/d\rceil + 1\right)^2$. All nodes in the network are transmitters, and each node in a 5-block gets a distinct label according to the numbering of the tile and the block it belongs to. More precisely, a node in the tile that is assigned the pair of integers (i, j) gets the label that is the jth element of L_i. Recall that there can be tiles which are partially contained in more than one block. In any case, the only node which can be contained in the tile belongs to only one block and thus its label is uniquely determined.

The send number of each node with label i is set to 2^i. ■

Proposition 1. *Algorithm* `UDG2` *successfully performs broadcast in any UDG radio network of unknown topology and known density d with work in $O\left(\tau\alpha^{g^2}\right)$, for some constant $\alpha > 1$.*

Proof. The correctness of the algorithm follows from Lemma 2 by induction on the length of a shortest path from the source to an arbitrary node v.

The work of the algorithm in every block is upper bounded by $2^{\left(\lceil\sqrt{2}/d\rceil+1\right)^2}$. As $\lceil\sqrt{2}/d\rceil \in \Theta(g)$, the lemma follows. □

We now turn attention to the lower bound on the work of a broadcasting algorithm.

Theorem 2. *The work required to complete broadcast in any UDG radio network of unknown topology and known density d is in $\Omega\left(\tau\beta^{g^2}\right)$, for some constant $\beta > 1$.*

Proof. Consider the class $\mathcal{N}$ of networks depicted in Fig. 1. The source occupies position $(0, 1.2)$ and the target occupies position $(0, 0)$. Nodes in the central part of the network are situated in an arbitrary subset of vertices of the largest regular square grid of side length d, contained in the intersection of the circles of radius 1 centered in the source and in the target, and of the circle of radius 1/2 centered in $(0, 0.6)$. Notice that there are $\Theta\left(g^2\right)$ vertices in the grid.

The set Q of nodes situated in the grid forms a clique, and each node in Q is within distance 1 from the source and from the target. It follows that a network in $\mathcal{N}$ is connected if and only if Q is nonempty.

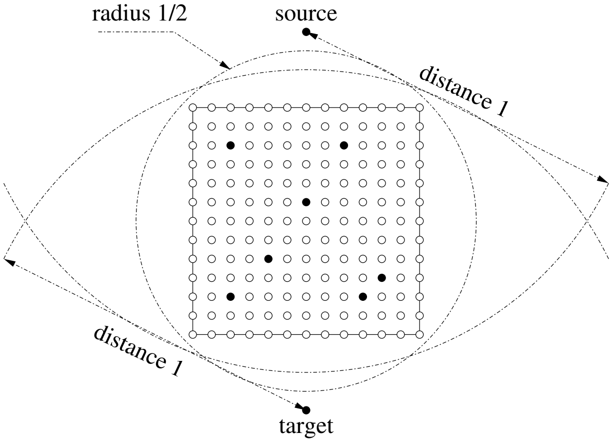


Fig. 1. A network of the class $\mathcal{N}$ used in the proof of Theorem 2

All nodes in Q become informed as soon as the first message sent by the source is delivered. When the first message from an informed node in Q is delivered without colliding with any delivery from other nodes in Q, broadcasting is completed successfully.

It follows that, until the completion of broadcasting, the only events that are perceived by nodes in Q are determined by deliveries of messages sent by the source. The source and the target will not receive any message until the completion of broadcasting.

Consider an arbitrary adaptive algorithm $\mathcal{A}$. $\mathcal{A}$ is forced to provide a send number for the source, and it is not able to modify this number until the end of the execution (no event is perceived by the source). The adversary delays all deliveries of nodes in Q until all messages from the source have been delivered, thus guaranteeing that no node in Q can perceive an event between the first delivery of one of its messages and the end of broadcasting.

This allows us to treat $\mathcal{A}$ as an oblivious algorithm, which is obliged to provide send numbers to all nodes in the network once and forever. In fact we can assume that the algorithm assigns send numbers to vertices in the grid (a node occupying vertex p is assigned the respective send number).

Now consider a vertex p of the grid. If algorithm $\mathcal{A}$ assigns send number 0 to p, then $\mathcal{A}$ is unsuccessful in the network $N \in \mathcal{N}$ where the set Q contains only the node in vertex p. It follows that all vertices in the grid have to be assigned positive send numbers.

If the set of send numbers, assigned by $\mathcal{A}$ to vertices of the grid, is not dominated, then there exists a set T of vertices for which the largest send number x, corresponding to vertex p_0, is at most equal to the sum of all others. The adversary can make $\mathcal{A}$ unsuccessful on the network $N \in \mathcal{N}$ in which nodes in Q occupy exactly vertices from T, by letting all deliveries collide. This can be done as follows. The deliveries of messages from the node in vertex p_0 are done at times $t_1 < t_2 < \ldots < t_x$. Every other message can be delivered at one of those time points, so that at each time point t_i at least two messages are delivered.

This contradiction shows that the set of send numbers, assigned by $\mathcal{A}$ to vertices of the grid must be dominated. As the set of vertices in the grid is of size $\Theta\left(g^2\right)$ and, by Lemma 1, any dominated set on k elements contains a number $\geq 2^{k-1}$, it follows that any algorithm working correctly on all networks in $\mathcal{N}$ requires work in $\Omega\left(\beta^{g^2}\right)$, for some constant $\beta > 1$. By arranging networks of class $\mathcal{N}$ in a chain of length τ, we get a lower bound on work in $\Omega\left(\tau\beta^{g^2}\right)$. □

All results of this subsection remain valid if, instead of density d of the network, only a lower bound d' on d is known to nodes. In this case, in the formulae for the upper and lower bounds on the work, the parameter $g = 1/d$ should be replaced by $g' = 1/d'$. If nothing is known about d, however, broadcasting in UDG radio networks turns out to be impossible, as shown in the following theorem.

Theorem 3. *Broadcast in UDG radio networks of unknown topology and unknown density is impossible.*

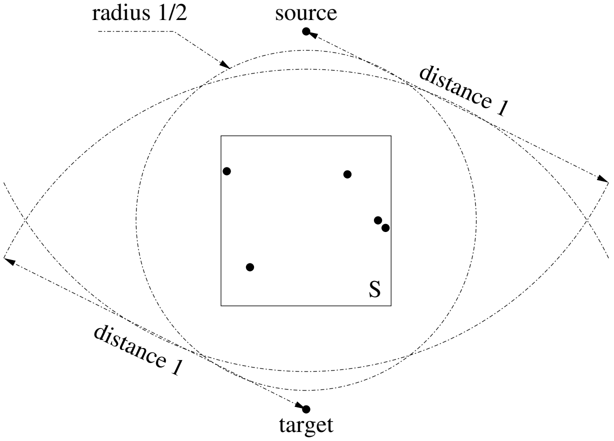


Fig. 2. A network of the class $\mathcal{C}$ used in the proof of Theorem 3

Proof. Consider the class $\mathcal{C}$ of networks depicted in Fig.2. Networks in $\mathcal{C}$ are similar to networks in class $\mathcal{N}$, defined in the proof of Theorem 2. In particular, the source and the target are located in the same positions, while the set Q of nodes is an arbitrary finite set of points in the plane, contained in the square S of side 1/2, centered at $(0, 0.6)$. A network $C \in \mathcal{C}$ is connected if and only if Q is non empty. By following the reasoning of the proof of Lemma 2, we can show that any adaptive algorithm $\mathcal{A}$ can be treated as an oblivious one when working on a network in $\mathcal{C}$. Algorithm $\mathcal{A}$ can then be identified with a function $f : S \mapsto \mathbf{N}$ which assigns send numbers to points in the square.

First assume that the range of f is infinite and suppose that broadcasting ends with work T. This leads to a contradiction, as we can always choose a network $C \in \mathcal{C}$ with 2 nodes in Q located in two points of S that are mapped to values larger than T. By scheduling the first T deliveries of messages sent by these two nodes in the same time points, the adversary can delay completion of broadcasting until the overall work of nodes in C is at least $2T + 1$, while we assumed the total work to be exactly T.

Hence the range of f must be finite. If $f(z) = 0$, for some point $z \in S$, then broadcasting is unsuccessful on the network C in which Q contains only one node located in z. It follows that all points of S have to be mapped by f into positive integers. Then there must exist two points, x and y, such that $f(x) = f(y)$. If this is the case, the adversary can make the algorithm unsuccessful on the network C where Q contains two nodes, one in the point x and the other in the point y, by delivering messages sent by these two nodes at the same time points. □

4 Symmetric Networks of Known Topology

In symmetric networks of known topology we prove an exponential gap between the work of adaptive and oblivious algorithms. Indeed, while an adaptive algorithm can complete broadcasting on n-node symmetric networks with work in $O(n)$, an oblivious algorithm requires work in $\Omega\left(c^n\right)$, for some constant $c > 1$ (cf. [1]).

4.1 Adaptive Broadcast

The following algorithm is adaptive. Each node decides if it sends a message, after each perceived event.

Algorithm SYM

Knowing the topology of the network, all nodes compute the same spanning tree T, rooted at the source. Notice that, even assuming that the source is unknown to other nodes in the network, this information can be appended to the source message and thus it can be made available to each node when it is woken up by the first received message.

All internal nodes of the spanning tree T are then explored in a depth first search manner, using token-based communication in order to avoid collisions. A message is sent only after the previous message has been delivered. Algorithm SYM ends when the token is sent back to the source by its last internal child. ■

Lemma 5. *Algorithm* SYM *successfully performs broadcast in any n-node symmetric radio network of known topology with work in $O(n)$.*

Proof. We first prove correctness of Algorithm SYM. Since any message is sent only after the previous message has been delivered, it follows that no collision can occur during the execution of broadcasting. As all internal nodes in T transmit at least once, and T is a spanning tree of the network, all nodes will eventually receive the source message.

Since the token traverses every edge of T either 0 or 2 times, the total work of the algorithm is smaller than $2n \in O(n)$. □

As the optimal work to perform broadcasting on the n-node line is $n-1$, we have the following theorem.

Theorem 4. *The optimal work required to complete broadcasting in any n-node symmetric radio network of known topology is $\Theta(n)$.*

4.2 Oblivious Broadcast

An oblivious algorithm, performing broadcasting in *any* n-node connected radio network of known topology (not necessarily symmetric) can be obtained by arranging nodes in increasing order of labels, and assigning send number 2^{i-1} to the ith node. Such an algorithm can be proved to be correct by induction on the length of a shortest path connecting the source to an arbitrary node v, using Lemma 2. The work required to complete broadcasting by this algorithm is in $O(2^n)$.

In [1], the following network class has been introduced in order to prove that oblivious broadcasting algorithms against a more powerful adversary require work in $\Omega(c^n)$, for some constant $c > 1$.

Networks in the above mentioned class contain $\binom{k}{3} + k + 1$ nodes, for integers $k > 0$. Nodes are partitioned in three layers: the first layer contains the source, the central layer contains k nodes, while the third layer contains the remaining $\binom{k}{3}$ nodes. Edges in these networks connect the source to all nodes in the second layer, while each node in the third layer is connected to a distinct subset of 3 nodes choosen among those in the second layer. Even though edges were oriented away from the source in [1], the same proof remains valid for oblivious algorithms even if the network is made symmetric, and even against our strong adversary (which was called the node adversary in [1]).

Since the upper bound $O(2^n)$ holds for arbitrary networks and the lower bound $\Omega(c^n)$ holds even for symmetric networks, we have the following theorem.

Theorem 5. *The optimal work of an oblivious algorithm, which completes broadcasting in radio networks of known topology, is in $O(2^n)$ and in $\Omega(c^n)$, for some constant $c > 1$, both for symmetric and for arbitrary networks.*

5 Networks of Unknown Topology

For networks of unknown topology we prove matching upper and lower bounds on the optimal work of broadcasting algorithms. The upper bound we show is based on the oblivious algorithm described below, which works correctly on any network (not necessarily symmetric) containing a directed path from the source to every node. The lower bound, on the other hand, holds even on symmetric networks and for all algorithms, including the adaptive ones.

An oblivious algorithm performing broadcasting in any connected radio network of unknown topology, is obtained by assigning to node with label i send number 2^{i-1}. The algorithm works in the same manner as the one for known topology networks introduced in the previous section, but its work, instead of depending on the number of nodes of the network, depends on the largest label N appearing in the network. (N need not be known to nodes.) Thus the work of this algorithm is in $O\left(2^N\right)$. This work is proved to be optimal by the following lemma.

Lemma 6. *The work required to complete broadcasting in any symmetric radio network of unknown topology is in $\Omega(2^N)$, where N is the largest label that appears in the network.*

Proof. To prove the lemma, consider the following class $\mathcal{Z}$ of networks. Networks in the class $\mathcal{Z}$ contain a source, a target and a set R of nodes. Each node in R is connected to the source s and to the target t. The source has label 1. Nodes in $R \cup \{t\}$ are labeled with distinct integers larger than 1, and N is the largest label appearing in $R \cup \{t\}$. R has to be non-empty, as otherwise the network would be disconnected.

The rest of the proof is based on the same idea as the proof of Lemma 2. Labels larger than 1 play the role of vertices in the grid.

As soon as a node in R delivers a message to the target without collisions, broadcasting in any network $Z \in \mathcal{Z}$ is completed. Hence, we can treat any adaptive algorithm $\mathcal{A}$ as an oblivious one, when working on networks in $\mathcal{Z}$. It follows that algorithm $\mathcal{A}$ has to assign a send number to any integer larger than 1 (which is a potential label of a node in R).

If there exists a label $\ell > 1$ such that $\mathcal{A}$ assigns send number 0 to ℓ, then $\mathcal{A}$ is unsuccessful on the network $Z \in \mathcal{Z}$ where the only node in R is labeled ℓ. It follows that $\mathcal{A}$ has to assign positive send numbers to all integers larger than 1. (Even if the maximum label N is known to $\mathcal{A}$, there is no guarantee that any particular label is assigned to a node in R, as N can be assigned to the target.) If the set of send numbers is not dominated, the adversary can make the algorithm $\mathcal{A}$ unsuccessful on the network $Z \in \mathcal{Z}$ where the (finite) set of send numbers assigned to nodes in R does not contain an element which is larger than the sum of all others (cf. the proof of Lemma 2).

As $R \cup \{t\}$ can contain up to $N-1$ nodes, the lemma follows from Lemma 1. □

6 Broadcasting against the Weak Adversary

In this section we present our results on the work of asynchronous broadcasting against the weak adversary. Recall that this adversary may delay delivery of messages sent by various nodes by arbitrary and unknown time intervals that may vary between nodes, but are equal for all messages sent by a given node. In this section we assume that nodes are equipped with local clocks. These clocks need not be synchronized. In one algorithm, working for UDG networks with unknown density, we assume that they tick at the same rate, and in the other, working for UDG networks with known (lower bound on) the density and also working for arbitrary networks with distinct positive integer labels, we weaken even this assumption and require only that all nodes know an upper bound on the ratio of ticking rates between the fastest and the slowest clock.

The idea of broadcasting algorithms working against the weak adversary comes from the observation that since delivery delay must be the same for all messages sent by a given node, if a node sends two messages at some time interval t, this interval may only be shifted by the adversary when delivering messages, but its length must be kept intact. Thus, using exponential intervals between just two messages sent by every node (where the exponent depends on the node label), blocking of messages can be prevented similarly as sending an exponential *number* of messages permitted preventing blocking by the strong adversary. (This is a similar work-for-time trade-off as, e.g., that in the Time-Slicing algorithm for leader election on the ring.) Due to the above possibility we can restrict the number of messages sent by every node to just 2, and thus use linear work.

We first describe an oblivious broadcasting algorithm working for networks of unknown topology whose nodes are labeled with distinct positive integers. In this algorithm we make a very weak assumption: not only clocks of nodes need not be synchronized, but they need not tick at the same rate, as long as the upper bound α on the ratio of ticking rates between the fastest and the slowest clock is known to all nodes. Without loss of generality we may assume that $\alpha \geq 2$.

Algorithm `Time-Intervals`
The source sends the message once. Upon receiving the source message, any node with label i, different from the source, sends two messages at time interval $4^{i\alpha}$ on its local clock. ■

Theorem 6. *Algorithm* `Time-Intervals` *successfully performs broadcast in an arbitrary n-node network, with work in $O(n)$.*

Proof. Since any node sends at most two messages, the work used is in $O(n)$. It remains to prove the correctness of the algorithm.

Fix the slowest ticking rate among all local clocks and call it *universal*. In the rest of the proof we will use only the universal ticking rate. Since α is the ratio of ticking rates between the fastest and the slowest clock, the (universal) time interval used by node with label i is $T_i = \frac{4^{i\alpha}}{\beta}$, where $1 \leq \beta \leq \alpha$. Fix a node u and its in-neighbors $v_1, \ldots, v_k$ that got the source message. Without loss of generality, assume that nodes v_i are ordered in increasing order of interval lengths T_i. The delivery times of messages sent by nodes v_i are $x_i, x_i + T_i$, for $i = 1, \ldots, k$. In order to prove that at least one of these messages will be heard by node u, it is enough to show that $T_k > T_1 + \cdots + T_{k-1}$. Hence it is enough to show that

$$\frac{4^{k\alpha}}{\alpha} > 4^{\alpha} + 4^{2\alpha} + \cdots + 4^{(k-1)\alpha}. \tag{1}$$

We have $\alpha^{1/\alpha} < 3$, hence

$$\frac{4^k}{\alpha^{1/\alpha}} > \frac{4}{3} \cdot 4^{k-1} > 4^1 + \cdots + 4^{k-1},$$

hence

$$\frac{4^{k\alpha}}{\alpha} > (4^1 + \cdots + 4^{k-1})^\alpha > 4^\alpha + 4^{2\alpha} + \cdots + 4^{(k-1)\alpha},$$

which proves (1) and concludes the proof by induction on the length of the shortest path from the source to a given node. □

We now turn attention to broadcasting against the weak adversary in UDG networks. First notice that if the topology of the network is known, then Algorithm `UDG1` clearly works correctly against the weak adversary as well, and it uses the same work $O(\tau)$, which is at most $O(n)$ for n-node networks. Thus we may restrict attention to networks with unknown topology. If a lower bound on the network density is known to all nodes, then we may use the same tiling as in Algorithm `UDG2` to obtain integer labels of all nodes of the network. Subsequently we use Algorithm `Time-Intervals` and the same argument as before proves its correctness and work complexity.

The only remaining case is that of UDG radio networks in which nothing is known about the density. Recall that in this case we proved that broadcasting against the strong adversary is impossible. Somewhat surprisingly, we will show that if the adversary is weak, then broadcasting in n-node UDG networks with unknown density can be performed with work in $O(n)$. Our algorithm, however, is only of theoretical interest: its main goal is to show a situation when broadcasting is impossible against the strong adversary, but can be done using linear work against the weak adversary. The impracticality of the algorithm has two reasons. First, since it works on networks of arbitrarily small density, it requires infinite precision of the perception of Euclidean coordinates by nodes. Second, the algorithm is non-constructive: it relies on the availability of a function whose existence we prove, but which is not constructed. Once this function is given to nodes, they can perform easy broadcasting with linear work. More precisely, our algorithm relies on the following set-theoretic lemma.

Lemma 7. *There exists a function $f : \mathbf{R} \times \mathbf{R} \longrightarrow \mathbf{R}^+$ such that any distinct elements $v_1, \ldots, v_k$ and $w_1, \ldots, w_r$ from $\mathbf{R} \times \mathbf{R}$ satisfy the inequality $\pm f(v_1) \pm \cdots \pm f(v_k) \neq \pm f(w_1) \pm \cdots \pm f(w_r)$.*

Proof. Let κ be the cardinal of the continuum. Hence the cardinality of sets $\mathbf{R} \times \mathbf{R}$ and $\mathbf{R}^+$ is κ. Using the axiom of choice (this is the non-constructive ingredient in the definition of the function f), order the set $\mathbf{R} \times \mathbf{R}$ in ordinal type κ. Let $x_\gamma : \gamma < \kappa$ be this ordering. We now define the function f by transfinite induction. Suppose that $f(x_\gamma)$ is already defined, for all $\gamma < \delta$. Consider the set Z of all reals $\pm f(x_{\gamma_1}) \pm \cdots \pm f(x_{\gamma_d})$, for any finite set $\{x_{\gamma_1}, \ldots, x_{\gamma_d}\}$ of elements of $\mathbf{R} \times \mathbf{R}$, such that $\gamma_1, \ldots, \gamma_d < \delta$. The set Z has cardinality equal to the maximum of the cardinality of δ and of $\aleph_0$ (the latter is the cardinality of the set of natural numbers). Hence the cardinality of Z is strictly less than κ, and consequently there exists a number $z \in \mathbf{R}^+ \setminus Z$. We put $f(x_\delta) = z$.

Thus the function f is defined by transfinite induction. It remains to verify that it has the desired property. Suppose by contradiction that some elements $v_1, \ldots, v_k$ and $w_1, \ldots, w_r$ from $\mathbf{R} \times \mathbf{R}$ satisfy the equality $\pm f(v_1) \pm \cdots \pm f(v_k) = \pm f(w_1) \pm \cdots \pm f(w_r)$. Let ξ be the largest index of all these elements in the ordering $x_\gamma : \gamma < \kappa$. It follows that $f(x_\xi) = \pm f(x_{\gamma_1}) \pm \cdots \pm f(x_{\gamma_d})$, for some $\gamma_1, \ldots, \gamma_d < \xi$, which contradicts the definition of $f(x_\xi)$. □

The broadcasting algorithm for UDG networks with unknown density assumes that all nodes have clocks ticking at the same rate. Given the function f whose existence follows from Lemma 7, the algorithm can be formulated as follows.

Algorithm `Non-Constructive`
The source sends the message once. Upon receiving the source message, any node with Euclidean coordinates (x, y), different from the source, sends two messages at time interval $f(x, y)$. ■

Theorem 7. *Algorithm* `Non-Constructive` *performs correct broadcasting in an arbitrary n-node UDG network, using work $O(n)$.*

Proof. As before, the complexity of the algorithm is straightforward. It remains to prove its correctness. Suppose that there exists a network with a node u that has in-neighbors $v_1, \ldots, v_k$ that got the source message. Suppose that there exist delays such that the adversary can shift time segments of lengths $f(v_1), \ldots, f(v_k)$ between messages sent by these nodes, so that all message deliveries are blocked by collisions. This implies that, for some nodes $w_1, \ldots, w_r, u_1, \ldots u_m \in \{v_1, \ldots, v_k\}$ we must have $f(w_1) + \cdots + f(w_r) = f(u_1) + \cdots + f(u_m)$, which contradicts the property of the function f established in Lemma 7. This contradiction shows that all nodes in every UDG network will eventually get the source message. □

7 Conclusion

We established upper and lower bounds on the optimal work of asynchronous broadcasting algorithms working against two types of adversaries in several classes of networks: symmetric and arbitrary directed networks and networks represented by unit disc graphs. While the complexity of most presented algorithms has been proved optimal by showing matching lower bounds, in two cases gaps between upper and lower bounds on the optimal work required by asynchronous broadcasting still remain. These gaps concern the strong adversary (bounds against the weak adversary are tight in all cases). For broadcasting in UDG radio networks of unknown topology but known density, our upper and lower bounds on optimal work are $O(\tau\alpha^{g^2})$ and $\Omega(\tau\beta^{g^2})$, respectively, for some constants $\alpha > \beta > 1$. This gap concerns both adaptive and oblivious algorithms. On the other hand, for symmetric networks of known topology, the upper and lower bounds on optimal work of oblivious algorithms are $O(2^n)$ and $\Omega(c^n)$, respectively, for some constant $1 < c < 2$. This latter gap is "inherited" from [1], where it concerned arbitrary directed networks of known topology. Closing these gaps is a natural open problem.

References

1. Chlebus, B.S., Rokicki, M.A.: Centralized asynchronous broadcast in radio networks. Theor. Comput. Sci. 383(1), 5–22 (2007)
2. Pelc, A.: Activating anonymous ad hoc radio networks. Distributed Computing 19, 361–371 (2007)
3. Emek, Y., Gasieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in UDG radio networks with unknown topology. In: PODC, pp. 195–204 (2007)
4. Emek, Y., Kantor, E., Peleg, D.: On the effect of the deployment setting on broadcasting in euclidean radio networks. In: PODC, pp. 223–232 (2008)
5. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. J. Comput. Syst. Sci. 43(2), 290–298 (1991)
6. Chlamtac, I., Kutten, S.: On broadcasting in radio networks - problem analysis and protocol design. IEEE Trans. on Communications 33, 1240–1246 (1985)
7. Chlamtac, I., Weinstein, O.: The wave expansion approach to broadcasting in multihop radio networks. IEEE Trans. on Communications 39, 426–433 (1991)
8. Elkin, M., Kortsarz, G.: Improved schedule for radio broadcast. In: SODA, pp. 222–231 (2005)
9. Gaber, I., Mansour, Y.: Centralized broadcast in multihop radio networks. J. Algorithms 46(1), 1–20 (2003)
10. Gasieniec, L., Peleg, D., Xin, Q.: Faster communication in known topology radio networks. In: PODC, pp. 129–137 (2005)
11. Kowalski, D., Pelc, A.: Optimal deterministic broadcasting in known topology radio networks. Distributed Computing 19(3), 185–195 (2007)
12. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. 45(1), 104–126 (1992)
13. Bruschi, D., Del Pinto, M.: Lower bounds for the broadcast problem in mobile radio networks. Distrib. Comput. 10(3), 129–135 (1997)
14. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Dist. Computing 15(1), 27–38 (2002)
15. Chlebus, B.S., Gasieniec, L., Östlin, A., Robson, J.M.: Deterministic radio broadcasting. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 717–728. Springer, Heidelberg (2000)
16. Chrobak, M., Gasieniec, L., Rytter, W.: Fast broadcasting and gossiping in radio networks. In: FOCS, pp. 575–581 (2000)
17. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: SODA, pp. 709–718 (2001)
18. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. J. Algorithms 60(2), 115–143 (2006)
19. Kowalski, D.R., Pelc, A.: Time complexity of radio broadcasting: adaptiveness vs. obliviousness and randomization vs. determinism. Theor. Comput. Sci. 333(3), 355–371 (2005)
20. De Marco, G.: Distributed broadcast in unknown radio networks. In: SODA, pp. 208–217 (2008)

21. Kushilevitz, E., Mansour, Y.: An Omega($D \log N/D$) lower bound for broadcast in radio networks. SIAM J. Comput. 27(3), 702–712 (1998)
22. Kowalski, D., Pelc, A.: Broadcasting in undirected ad hoc radio networks. Distrib. Comput. 18(1), 43–57 (2005)
23. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. J. of Discrete Algorithms 5(1), 187–201 (2007)
24. Diks, K., Kranakis, E., Krizanc, D., Pelc, A.: The impact of knowledge on broadcasting time in linear radio networks. Theoretical Computer Science 287, 449–471 (2002)
25. Kranakis, E., Krizanc, D., Pelc, A.: Fault-tolerant broadcasting in radio networks. J. Algorithms 39(1), 47–67 (2001)
26. Ravishankar, K., Singh, S.: Broadcasting on $[0, l]$. Discrete Applied Mathematics 53, 299–319 (1994)
27. Sen, A., Huson, M.L.: A new model for scheduling packet radio networks. In: INFOCOM, pp. 1116–1124 (1996)

Distributed Approximation of Cellular Coverage

Boaz Patt-Shamir[1],*, Dror Rawitz[1], and Gabriel Scalosub[2]

[1] School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel
{boaz,rawitz}@eng.tau.ac.il
[2] Department of Computer Science, University of Toronto, Toronto, ON, Canada
scalosub@cs.toronto.edu

Abstract. We consider the following model of cellular networks. Each base station has a given finite capacity, and each client has some demand and profit. A client can be covered by a specific subset of the base stations, and its profit is obtained only if its demand is provided in full. The goal is to assign clients to base stations, so that the overall profit is maximized subject to base station capacity constraints.

In this work we present a distributed algorithm for the problem, that runs in polylogarithmic time, and guarantees an approximation ratio close to the best known ratio achievable by a centralized algorithm.

1 Introduction

In future cellular networks, base stations capacities, as well as clients diversity, will become a major issue in determining client coverage and service. The main service provided by current cellular networks is voice traffic, which has relatively small bandwidth requirement, compared to the capacity available at the base stations. However, in future 4G cellular networks the services offered by cellular providers are expected to require higher rates, and client diversity is expected to increase. Such services include video traffic, and other high-rate data traffic. In such settings, maximizing the usage of available resources would become a more challenging task, and as recent evidence has shown, current solutions might end up being far from optimal.

In this work we address one of the basic optimization problems arising in such settings, namely, the assignment of clients to base stations, commonly known as *cell selection*. We take into account both base stations diversity, encompassed by (possibly) different capacities for each base station, as well as clients diversity, encompassed by different clients having different demands, and different profits. A major obstacle in tackling this problem is due to the fact that, naturally, different base stations have different coverage areas, and therefore can contribute to covering only some subset of the set of clients.

The currently used scheme for assigning clients to base stations is the greedy best-SNR-first approach, where clients and base stations interact locally, and assignment is made in a greedy and local manner. This approach might provide

* Supported in part by the Israel Science Foundation, grant 664/05.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 331–345, 2008.

reasonable performance when clients demands are small with respect to the base stations capacity (as is the case for voice traffic), and all clients are considered equally profitable. However, as clients' demands increase, the resource utilization degrades considerably. This degradation was the main concern of a recent work by Amzallag et al. [1], who propose several global mechanisms for determining the assignment of clients to base stations, while providing guarantees as to their performance. However, these mechanisms are based on a centralized approach: effectively, it is assumed that information of the entire network is gathered by a central server, which locally finds an assignment of clients, and then distributed back to the base stations. This approach suffers from the usual drawbacks of a centralized approach, such as inability to scale to large numbers.

In this paper we present an efficient *distributed* algorithm that computes an assignment. In our algorithm, clients and base stations communicate locally, and after a polylogarithmic number of rounds of communication agree upon an assignment, without resorting to a centralized algorithm with global knowledge. Our algorithm is robust and handles both various base stations capacities, as well as clients' heterogeneous demands, and variable profits. We give worst-case guarantees on the performance of our algorithm (with high probability), and show its approximation ratio is arbitrarily close to the best known ratio of centralized solution. To state our results more precisely, let us first formalize the problem and the computational model.

1.1 Problem Statement and Model

We consider the following model. An instance of the *Cellular Coverage problem* (CC) consists of the following components.

- A bipartite graph $G = (I, J, E)$ where $I = \{1, 2, \ldots, m\}$ is a set of *base stations* and $J = \{1, 2, \ldots, n\}$ is a set of clients. An edge (i, j) represents the fact that client j can receive service from base station j.
- A *capacity* function c that maps each base station i to a non-negative integer $c(i)$ called the capacity of i.
- A *demand* function d that maps each client j to a non-negative integer $d(j)$ called the demand of j.
- A *profit* function p that maps each client j to a non-negative integer $p(j)$ called the profit of j.

The output of CC is a partial assignment of clients to base stations, where a client may be assigned only to one of its neighboring base stations, and such that the total demand of clients assigned to a base station does not exceed its capacity. The goal is to maximize the sum of profits of assigned clients.

Given a constant $r \leq 1$, an instance of CC is said to be *r-restricted* if for every $(i, j) \in E$, we have $d(j) \leq r \cdot c(i)$, i.e., no client demands more than an r-fraction of the capacity of a base station from which it may receive service. The r-CC problem is the CC problem where all instances are r-restricted.

Let (I, J, E, c, d, p) be an instance of CC. For every base station i we let $N(i) \subseteq J$ denote the set of clients which can be covered by i, and for every client

j we let $N(j) \subseteq I$ denote the set of base stations which can potentially cover j. For any set of clients or base stations A, we let $N(A) = \bigcup_{v \in A} N(v)$. Given any function f (e.g., the demand, profit, or capacity), we let $f(A) = \sum_{v \in A} f(v)$. Given any subset of clients $S \subseteq J$, we define $\overline{S} = J \setminus S$.

Let x be a mapping that assigns clients to base stations, and let clients(x) denote the set of clients assigned by x to some base station. For a base station $i \in I$ let $\text{load}_x(i) = \sum_{j\,:\,x(j)=i} d(j)$, i.e., $\text{load}_x(i)$ is the sum of all demands assigned to i. We further let $\text{res}_x(i)$ denote the residual capacity of i, i.e., $\text{res}_x(i) = c(i) - \text{load}_x(i)$. For a client $j \in J \setminus \text{clients}(x)$ and a base station $i \in N(j)$ we say that j *is eligible for* i if $\text{res}_x(i) \geq d(j)$.

Model of Computation. We consider the standard synchronous message passing distributed model of computation (cf. the CONGEST model of [2]). Briefly, the system is modeled as an undirected graph, where nodes represent processing entities and edges represent communication links. Execution proceeds in synchronous rounds, each round consists of three substeps: first, each node may send a message over each of its incident links; then nodes receive all messages sent to them in that round; and finally some local computation is carried out. The length of every message is restricted to $O(\log n)$ bits, where n is the number of nodes in the system. Nodes may have unique identifiers of $O(\log n)$ bits. Note that in our model, the communication graph is identical to the input graph of CC.

1.2 Our Results

We present a distributed randomized algorithm for the r-CC problem, which, given any $\gamma \in (0, 1]$, runs in time $O(\gamma^{-2} \log^3 n)$, and guarantees, with high probability, to produce an assignment with overall profit at least a $\frac{1-r}{2-r}(1-\gamma)$ fraction of the optimal profit possible.

We note that our running time is affected by the best running time of a distributed algorithm finding a maximal matching, which we use as a black box. The best algorithm up to date, due to [3], has expected running time $O(\log n)$, which is reflected by one of the logarithmic factors of our running time. Any improvement in such an algorithm to time $O(T)$ would immediately imply a running time of $O(\gamma^{-2} T \log^2 n)$ for our algorithm.

1.3 Previous Work

There has been extensive work done on cell selection, and client assignment to base stations in the networking community, focusing on aspects of channel allocation, power control, handoff protocols, and cell site selection (e.g., [4,5,6,7]).

Our proposed model was studied in the offline setting in [1]. They study two types of setting, where the first allows the coverage of a client by at most one base station, and the other allows covering a client by more than one base station, whereas its profit is obtained only if its entire demand is satisfied. They refer to the former as the *cover-by-one* paradigm, and to the latter as the *cover-by-many* paradigm. They present a local ratio algorithm for the r-CC problem using the cover-by-many paradigm which is guaranteed to produce a $(1-r)$-approximate solution. This algorithm is based upon a simpler algorithm using the cover-by-one paradigm which is guaranteed to produce a $\frac{1-r}{2-r}$-approximate solution, and this is with respect to the best possible cover-by-many solution.

The CC problem using the cover-by-one paradigm is also closely related to the multiple knapsack problem with assignment constraints, for which a special case where clients demands equal their profits is considered in [8]. They present several approximation algorithms for this problem, starting from a randomized LP-rounding algorithm which produces a $\frac{1}{2}$-approximate solution, through an algorithm employing a sequential use of the FPTAS for solving a single knapsack problem which produces a $(\frac{1}{2}-\varepsilon)$-approximate solution, and finally a greedy algorithm which is guaranteed to produce a $\frac{1}{3}$-approximate solution. Another related problem is the general assignment problem considered in the offline settings in [9,10,11] (see also references therein).

Although the offline problem, and its various variants, has received considerable amount of attention in recent years, we are not aware of any attempts to solve any of the above problems in a distributive manner. Some very restricted cases of the problem, namely, where all the capacities and all the demands are the same, can be viewed as matching problems, and hence distributed algorithms for solving them are available (see [12] for an overview of these results). Specifically, for the subcase where clients profits are arbitrary, the problem reduces to finding a maximum weight matching in the underlying bipartite graph, whereas in the subcase where all clients profits are the same, the goal is to find a maximum cardinality matching. For both these problems the best solutions are due to [12]. For the former problem they show how to obtain a $(\frac{1}{2}-\varepsilon)$-approximate solution, whereas for the latter problem, they guarantee a $(1-\varepsilon)$-approximate solution. Another closely related problem is that of finding a maximal matching in an unweighted graph for which there exists a distributed algorithm [3]. All of the above algorithms are randomized, and their expected running time is logarithmic in the number of nodes.

Paper Organization. In Section 2 we discuss the centralized version of the problem. In Section 3 we present the details of our distributed algorithm, and analyze its performance guarantee and running time. In Section 4 we present some extensions of our results, and finally, in Section 5 we conclude and discuss some open questions.

2 A Centralized Approach

In this section we explain a centralized algorithm for CC which we later implement in a distributed model. The idea is to use the *local ratio* approach [13,14], which in our case boils down to an extremely simple greedy algorithm: compute, for each client, its profit-to-demand ratio; scan clients in decreasing order of this ratio, and for each client in turn, assign it to a base station if possible, or discard it and continue. However, to facilitate the analysis, we present this algorithm in recursive form in Algorithm 1.

To specify the algorithm, we need the following concept. Given an assignment x, let $J' \subseteq J$ be a set of clients such that $J' \supseteq \text{clients}(x)$. We say that x is an *α-cover* w.r.t. J' if the following condition holds: if $\text{load}_x(i) < \alpha \cdot c(i)$ for a base station i, then $N(i) \cap J' \subseteq \text{clients}(x)$. In other words, a client from J' may not be assigned by an α-cover only if the load of each of its neighbors is at least an α fraction of its capacity.

The key step in Algorithm 1 below (Step 1) is to extend the assignment returned by the recursive call of Step 1. The algorithm maintains the invariant that the returned assignment is an α-cover w.r.t. J. Whenever the recursive call of Step 1 returns, the assignment is extended using the clients in J'' to ensure that the invariant holds true.

Algorithm 1 — cCC(I, J, c, d, p)

```
 1. if $J = \emptyset$ then return empty assignment.
 2. $J' = \{j \in J \mid p(j) = 0\}$
 3. if $J' \neq \emptyset$ then
 4.     return cCC$(I, J \setminus J', c, d, p)$
 5. else
 6.     $\delta = \min_{j \in J} \left\{ \frac{p(j)}{d(j)} \right\}$
 7.     for all $j$, define $p_1(j) = \delta \cdot d(j)$
 8.     $x \leftarrow$ cCC$(I, J, c, d, p - p_1)$
 9.     $J'' = \{j \in J \mid p(j) = p_1(j)\}$
10.     using clients from $J''$, extend $x$ to an $\alpha$-cover w.r.t. $J$
11.     return $x$
12. end if
```

The key to the analysis of the algorithm is the following result (see also [1,8]).

Lemma 1. *Assume there exists some $\delta \in \mathbb{R}^+$ such that $p(j) = \delta \cdot d(j)$ for every client j. Consider any assignment x. If x is an α-cover w.r.t. J, then* $p(\text{clients}(x)) \geq (\frac{\alpha}{1+\alpha}) \cdot p(\text{clients}(y))$ *for any feasible assignment y.*

Proof. Let $S = \text{clients}(x)$, and let $Y = \text{clients}(y)$. Then

$$\begin{aligned} p(Y) &= p(Y \cap S) + p(Y \cap \overline{S}) \\ &= \delta \left[d(Y \cap S) + d(Y \cap \overline{S}) \right] \\ &\leq \delta \left[d(S) + c(N(\overline{S})) \right] \\ &\leq \delta \left[d(S) + d(S)/\alpha \right] \\ &= \frac{\alpha + 1}{\alpha} \cdot p(S) \ , \end{aligned}$$

where the first inequality follows from the feasibility of y and the definition of $N(\overline{S})$ (see Figure 1), and the second inequality follows from our assumption that x is an α-cover w.r.t. J. □

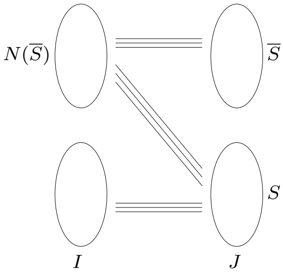


Fig. 1. Depiction of a solution S that uses an α fraction of the capacity of $N(\overline{S})$

We note that the above lemma actually bounds the profit of an α-cover even with respect to *fractional* assignments, where a client may be covered by several base stations (so long as the profit is obtained only from fully covered clients).

The following theorem shows that Algorithm cCC produces an $\frac{\alpha}{\alpha+1}$-approximation, assuming one can extend a given solution to an α-cover.[1]

Theorem 1. *Algorithm* cCC *returns an* $\frac{\alpha}{\alpha+1}$*-approximation.*

Proof. The proof is by induction on the number of recursive calls. The base case is trivial. For the inductive step, we need to consider two cases. For the cover returned in Step 1, by the induction hypothesis it is an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. $J \setminus J'$, and since all clients in J' have zero profit, it is also an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. J. For the cover returned in Step 1, note that by the induction hypothesis, the solution returned by the recursive call in Step 1 is an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. profit function $p - p_1$. Since every client $j \in J''$ satisfies $p(j) - p_1(j) = 0$, it follows that any extension of this solution using clients from J'' is also an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. to $p - p_1$. Since the algorithm extends this solution to an α-cover by adding clients from J'', and p_1 is proportional to the demand, by Lemma 1 we have that the extended α-cover is an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. p_1. By the Local-Ratio Lemma (see, e.g., [14]), it follows that this solution is an $\frac{\alpha}{\alpha+1}$-approximation w.r.t. p, thus completing the proof. □

A closer examination of the local-ratio framework presented above shows that what the algorithm essentially does is to traverse the clients in non-decreasing order of their profit-to-demand ratio, while ensuring that any point, the current solution is an α-cover w.r.t. clients considered so far.

In what follows, we build upon the above framework, and show that given any $\gamma \in (0, 1]$, one can emulate distributively the above approach, while losing a mere $(1 - \gamma)$ factor in the approximation guarantee. Furthermore, we show that this can be obtained in time $O(\gamma^{-2} \log^3 n)$.

[1] In [1] they extend the solution to a maximal solution (w.r.t. set inclusion), which implies a $(1 - r)$-cover.

3 A Distributed Approach

In this section we present a distributed algorithm for the CC problem. We first give an overview of our approach, and then turn to provide the details of our algorithm.

3.1 Overview

Conceptually, the algorithm is derived by a series of transformations that allows us to represent any instance of CC as a multiple instances of maximal matching, which can be solved efficiently in a distributed model. Specifically, our generalization proceeds as follows.

Unit demand, unit capacity, unit profit. First, consider the case where all demands, capacities and profits are one unit each. In this case, CC is exactly equivalent to the problem of maximum matching, which for any $\varepsilon \in (0, 1])$ can be solved in $O(\frac{\log n}{\varepsilon})$ rounds with approximation ratio $(1 - \varepsilon)$ [12].

Unit demand, different capacities, unit profit. Next, suppose that all demands and profits are equal, but capacities may vary. This case is easy to solve as follows: Each base station i of capacity c can be viewed as c unit-capacity "virtual" base stations $i_1, \ldots, i_c$; then maximum matching can be applied to the graph consisting of the original clients and virtual base stations, where each client j originally connected to a base station i is now connected to all the induced virtual base stations $i_1, \ldots, i_c$. Some care needs to be exercised to show that this emulation can be carried out using $O(\log n)$ bit messages without any increase in the running time.

Different demands, different capacities, profit equals demand. The next generalization is to consider the case where base stations have arbitrary capacities, and clients have arbitrary demands, but the profit from each client is proportional to its demand. To solve this case, we use a scaling-like method: we round each demand to the nearest power (from above) of $1 + \varepsilon$, where $\varepsilon > 0$ is a given parameter. This rounding reduces the number of different demands, and as we show, only the $O(\log_{(1+\varepsilon)} n) = O(\frac{\log n}{\varepsilon})$ topmost different demands need be considered. For each fixed demand, we are back in the previous case. The penalty of rounding the profits is a $(1 + \varepsilon)$ factor degradation in the approximation ratio. Some additional complications due to interference between the different demands degrade the approximation ratio to $\frac{1-r}{2-r}$, where r is the maximal demand-to-capacity ratio in the given instance. As mentioned above, the running time increases by a factor of $O(\frac{\log n}{\varepsilon})$.

Different demands, different capacities, different profit. This last generalization is taken care of by applying the local ratio method presented in Section 2, which means that we need to go over the different profit-to-demand ratios in decreasing order. To avoid too many such ratios, we again use the trick of rounding to the nearest power of $(1 + \varepsilon)$, but this time we round the profits, resulting in an additional degradation of $(1 + \varepsilon)$ factor in the approximation ratio, and an additional factor of $O(\frac{\log n}{\varepsilon})$ in the running time.

3.2 Partitioning the Clients

Recall that all demands and capacities are integral, and assume that the minimum demand is 1. Let $\varepsilon \in (0, 1]$.

First Cut: Partition by Cost-Effectiveness. We consider a partition of the clients into sets $J_0, J_1, \ldots$, such that a client j is in J_k iff $\frac{p(j)}{d(j)} \in [(1+\varepsilon)^k, (1+\varepsilon)^{k+1})$. If these ratios are polynomially bounded, then we have $O(\log_{(1+\varepsilon)} n) = O(\frac{1}{\varepsilon} \log n)$ such sets. Denote the number of these sets by W. For every client j, we let its $(1+\varepsilon)$*-rounded profit* be defined by

$$p_{(1+\varepsilon)}(j) = \min_{k \in \mathbb{N}} \left\{ (1+\varepsilon)^k d(j) \mid (1+\varepsilon)^k d(j) \geq p(j) \right\} .$$

The following lemma relates the value of any solution to an instance of the r-CC problem, to the value of the same solution when considering the instance with $(1+\varepsilon)$-rounded profits.

Lemma 2. *Given some input $\mathcal{I} = (I, J, E, c, d, p)$ to the r-CC problem, and some $\varepsilon > 0$, consider the instance $\mathcal{I}' = (I, J, E, c, d, p_{(1+\varepsilon)})$, and let x be any assignment of clients. It follows that* $\text{clients}(x)$ *is a feasible solution to $\mathcal{I}$ iff it is a feasible solution to $\mathcal{I}'$, and* $p_{(1+\varepsilon)}(\text{clients}(x)) \leq (1+\varepsilon)\, p(\text{clients}(x))$.

Proof. The first part of the claim follows from the fact that feasibility is not affected by the change in the profit function, since it relies solely on the underlying topology of G, along with the base stations' capacities and clients' demands. For the second part, by the definition of $p_{(1+\varepsilon)}$ it follows that for every client j, $p_{(1+\varepsilon)}(j) \leq (1+\varepsilon)\, p(j)$, and therefore by considering sets of clients, the claim follows. □

The following corollary is an immediate consequence of Lemma 2.

Corollary 1. *For every $\beta \leq 1$, and any instance $\mathcal{I} = (I, J, E, c, d, p)$, if a feasible assignment x is a β-approximate solution with respect to profit function $p_{(1+\varepsilon)}$, then it is a $\frac{\beta}{(1+\varepsilon)}$-approximate solution with respect to profit function p.*

Proof. Consider the instance $\mathcal{I}' = (G, c, d, p_{(1+\varepsilon)})$. First note that by Lemma 2, S is a feasible solution to $\mathcal{I}$. Furthermore, for any optimal solution S^* to $\mathcal{I}$, S^* is also a feasible solution to $\mathcal{I}'$, and since for any j we have $p_{(1+\varepsilon)}(j) \geq p(j)$ it follows that

$$\text{OPT}(\mathcal{I}) \leq \text{OPT}(\mathcal{I}') \leq \frac{1}{\beta} \cdot p_{(1+\varepsilon)}(S) \leq \frac{1+\varepsilon}{\beta} \cdot p(S) \ ,$$

as required, where the last inequality follows from lemma 2. □

Corollary 1 ensures that by assuming that in every set J_k, the profits are of the same proportion as the demands, does not cause us to lose more than a $\frac{1}{(1+\varepsilon)}$ factor of the original profit obtained from the same solution.

We henceforth assume that the actual profit of every client j is its $(1+\varepsilon)$-rounded profit. It follows that in every J_k all clients have profits which are proportional to the demand. Note that in such a case, the order implied on the set of clients by their profit-to-demand ratio is exactly the same as the order in which cCC considers the clients, also assuming $(1+\varepsilon)$-rounded profits.

Second Cut: Partition by Demand. For every k, we consider a subpartition of the set J_k into subsets $J_k^0, J_k^1, \ldots$ such that a client $j \in J_k$ is in J_k^ℓ if $d(j) \in [(1+\varepsilon)^\ell, (1+\varepsilon)^{\ell+1})$. For every k we let r_k denote the maximal ℓ such that $J_k^\ell \neq \emptyset$. We further let $J'_k = \bigcup_{\ell \geq r_k - 3\log_{(1+\varepsilon)} n} J_k^\ell$.

3.3 A Distributed Algorithm

We now turn to describe our distributed algorithm, dCC. Let $\alpha = \frac{1-r}{1+\varepsilon}$. The goal of our algorithm is to produce an assignment that is an α-cover with respect to $J' = \bigcup_k J'_k$. Using the results presented in Section 2 this would serve as a first component in proving our approximation guarantee. We later show that by restricting our attention to J' we lose a marginal factor in the approximation ratio.

The algorithm, whose formal description in given in Algorithm 2, works as follows. It traverses the subsets J_k in decreasing order of k. For each k, it computes an α-cover with respect to J'_k (this is done by using Algorithm MC which we discuss in the following section). This enables us to show that dCC also produces an α-cover with respect to J'. For clarity we first analyze the performance of Algorithm dCC assuming that MC indeed produces an α cover with respect to J'_k, and then discuss the details of Algorithm MC. The following lemma shows that this assumption on MC suffices in order for dCC to produce an α-cover with respect to J'.

Algorithm 2 — dCC(I, J, c, d, p)

```
$R = J$ {uncovered eligible clients}
$x \leftarrow$ empty assignment
for every $i \in I$, let $\mathrm{res}_x(i) = c(i)$ {the residual capacity}
**for** $k = W$ downto 0 **do**
    $(x_k, \mathrm{res}_x) \leftarrow \mathrm{MC}(k, I, J_k \cap R, \mathrm{res}_x, d)$ {compute an $\alpha$-cover w.r.t. $J'_k$}
    update $x$ according to $x_k$
    remove all clients matched in $x_k$ from $R$
    remove all ineligible clients from $R$
**end for**

**return** $x$
```

Lemma 3. *Assume that for every k,* MC *produces an α-cover with respect to J'_k. For every k, the cover produced by* DCC *after the end of the kth iteration, is an α-cover with respect to $J'_{\geq k} = \bigcup_{t \geq k} J'_k$.*

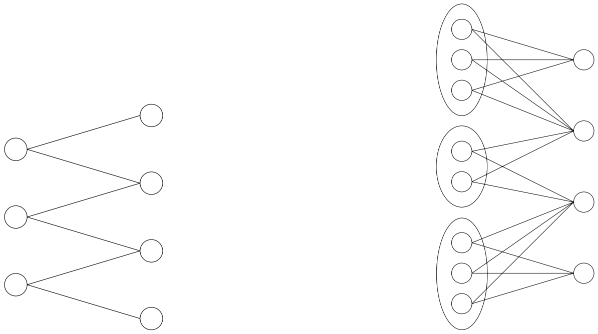
Proof. The claim follows from the fact that in every iteration, the residual capacity never increases, which implies that for any k, if we extend an α-cover for $J'_{\geq k}$, and ensure the extension is an α-cover for J'_{k-1}, then we obtain an α-cover for $J'_{\geq k-1}$. □

3.4 Covering Equally Cost-Effective Clients

In order to describe our algorithm, we need the following notion. Consider any $\varepsilon \in (0, 1)$, and an instance $\mathcal{I} = (I, J, E, c, d, p)$ to the CC problem. Assume there exist some μ such that for all clients j, $d(j) \in [\frac{\mu}{1+\varepsilon}, \mu]$. We consider the *virtual base-stations instance* $\mathrm{VG}(\mathcal{I}) = (I', J, E', c', d, p)$, where every base station $i \in I$ is replaced by $\lfloor c(i)/\mu \rfloor$ base stations in I', each with capacity μ. We refer to these new base stations as *copies* of i. E' contains all virtual edges implied by the above swap, i.e., for every $(i, j) \in E$, we have an edge (i', j) for every copy i' of i. Given such an instance $\mathrm{VG}(\mathcal{I})$, note that every copy has sufficient capacity to cover any single client, and at most one such client. We may therefore assume without loss of generality that all demands are unit demands, and all capacities (of the copies) are unit capacities. It follows that any matching in $\mathrm{VG}(\mathcal{I})$ induces a feasible assignment of clients to base stations. See Figure 2 for an outline of a virtual instance corresponding to an original instance.

Given J_k, the goal of algorithm MC, whose formal description appears in Algorithm 3, is to produce an assignment that is an α-cover with respect to J'_k.

In what follows, we refer to MM as any distributed algorithm for finding a maximal matching in an unweighted bipartite graph. As mentioned earlier, the currently best algorithm for this problem is due to [3], which finds a maximal matching (with high probability) in expected logarithmic time.



(a) Original instance.

(b) Virtual base-stations instance.

Fig. 2. The original graph representing $\mathcal{I}$ and its corresponding virtual base-stations graph representing $\mathrm{VG}(\mathcal{I})$. The clients are placed on the right

Intuitively, the algorithm works as follows. For every base station i, the base station traverses $3\log_{(1+\varepsilon)} n$ subsets J_k^ℓ in decreasing order of ℓ, starting from the maximal ℓ for which it has eligible neighbors in. For every such J_k^ℓ, the algorithm considers its corresponding virtual base-stations instance G_k^ℓ while taking into account only eligible clients. It then computes distributively a maximal matching in the above graph using algorithm MM. Any matched client is assigned to its matched base station, and each base station updates its residual capacity accordingly.

We first show that the algorithm computes a $\frac{(1-r)}{1+\varepsilon}$-cover with respect to J'_k. In the sequel we show that by considering only the topmost $3\log n$ subsets we are able to obtain polylogarithmic running time in exchange for a marginal drop in the approximation ratio.

Lemma 4. *Algorithm* MC *computes a feasible* $\frac{1-r}{1+\varepsilon}$*-cover with respect to* J'_k*.*

Proof. We first note that the algorithm produces a feasible cover with respect to J'_k. To see this, note that any client j is assigned to a base station i only via the matching produced by MM. Since in the virtual base stations graph used by MM, we have $\lfloor \mathrm{res}_{x_k}(i)/d_\ell^i \rfloor$ copies of base station i, each with capacity d_ℓ^i, and every one of its neighbors in this round has demand at most d_ℓ^i (by Step 3), we are guaranteed that every base station has sufficient capacity to cover its matched clients in every round (since residual capacities are updated at the end of every round). We now turn to show the solution is indeed a $\frac{1-r}{1+\varepsilon}$-cover with respect to J'_k.

Consider any uncovered client $j \in J'_k$, and let $\ell \geq r_k - 3\log_{(1+\varepsilon)} n$ be such that $j \in J_k^\ell$. All we need to show is that for every $i \in N(j)$, i has used at least

Algorithm 3 — MC(k, I, J_k, c, d)

```
1.  $x_k \leftarrow$ empty assignment
2.  for every $i \in I$, $\mathrm{res}_{x_k}(i) = c(i)$ {the residual capacity}
3.  for every $i \in I$, let $r_k^i = \max\{\ell \mid \exists$ eligible $j \in N(i) \cap J_k^\ell\}$ {every base station picks its highest relevant level}
4.  every base station $i$ does: {we only consider the topmost $3\log_{(1+\varepsilon)} n$ subsets}
5.  for $\ell = r_k^i$ downto $r_k^i - 3\log_{(1+\varepsilon)} n$ do
6.      $i$ announces to its eligible neighbors in $J_k^\ell$ about the round
7.      $d_\ell^i \leftarrow$ maximal demand of an eligible client in $J_k^\ell \cap N(i)$
8.      $i$ uses $\lfloor \mathrm{res}_{x_k}(i)/d_\ell^i \rfloor$ copies of itself in the virtual graph $G_k^\ell$
9.      update $x_k$ according to MM$(G_k^\ell)$ {performed in parallel}
10.     update $\mathrm{res}_{x_k}(i)$ {update the residual capacity according to the demand of matched clients}
11. end for
12.
13. return $(x_k, \mathrm{res}_{x_k})$
```

a $\frac{1-r}{1+\varepsilon}$-fraction of its capacity. Let i be any base station in $N(j)$. Note first that by maximality of r_k we have $r_k^i \leq r_k$, which implies that $\ell \geq r_k^i - 3\log_{(1+\varepsilon)} n$.

If i did not participate in a round corresponding to ℓ, this can only be because at that time, $\mathrm{res}_{x_k}(i) < d_\ell^i$.[2] Since $d_\ell^i \leq r \cdot c(i)$, this implies that $\mathrm{res}_{x_k}(i) < r \cdot c(i)$. It follows that in this case we are done.

Assume i did participate in a round corresponding to ℓ, and consider the copies of i in the virtual base stations graph. By Step 3, there are $\lfloor \mathrm{res}_{x_k}(i)/d_\ell^i \rfloor$ copies of i in this graph, and they were all matched by MM since otherwise, we could have matched j, whose demand is at most d_ℓ^i, contradicting maximality of the output of MM.

The unused capacity due to rounding down the number of copies of i in the virtual graph implies the base station left out less than $d_\ell^i \leq r \cdot c(i)$ of its capacity from being used in this round. It follows that it has dedicated (and partly used) at least a $(1-r)$-fraction of its capacity for covering clients in all rounds up to (and including) round ℓ. Every client j covered by i in any such round $\ell' \leq \ell$, is matched to a copy of the base station, which represents a capacity of $d_{\ell'}^i \in [d(j), (1+\varepsilon)d(j))$. It follows that we effectively use up at least a $\frac{1}{1+\varepsilon}$-fraction of the capacity dedicated for covering clients in all rounds up to (and including) round ℓ. Combining the above we obtain that base station i has used at least a $\frac{1-r}{1+\varepsilon}$ of its capacity, as required. □

Note that algorithm MC performs $3\log_{(1+\varepsilon)} n$ rounds, in each of which it executes Algorithm MM.

3.5 Wrapping Up

In this section we show how to combine the results presented in the previous sections, to obtain the following:

Theorem 2. *For every $\gamma \in (\frac{1}{n^2}, 1]$, algorithm* DCC *produces a $\frac{1-r}{2-r}(1-\gamma)$-approximate solution to r-CC, with high probability, in time $O(\gamma^{-2}\log^3 n)$.*

First we note that by combining Lemma 4 and Lemma 3, the solution produced by DCC is a $\frac{1-r}{1+\varepsilon}$-cover w.r.t. $J' = \bigcup_k J_k'$.

Let p_M be the maximal profit of any client in J. The following lemma shows that every $j \in J \setminus J'$, has very small profit.

Lemma 5. *Every $j \in J_k \setminus J_k'$ satisfies $p(j) \leq \frac{p_M}{n^3}$.*

Proof. Let $\ell < r_k - 3\log_{(1+\varepsilon)} n$ be such that $j \in J_k^\ell$. It follows that the demand of j is at most $(1+\varepsilon)^{r_k - 3\log_{(1+\varepsilon)} n} = \frac{(1+\varepsilon)^{r_k}}{n^3}$. It follows that $p(j) \leq \frac{p(j')}{n^3}$, where $j' \in J_k^{r_k}$. Note that by the definition of r_k, such a client j' exists. It follows that $p(j) \leq \frac{p_M}{n^3}$, as required. □

We henceforth refer to a client j for which $p(j) \geq \frac{p_M}{n^3}$ as a *fat* client. Lemma 5 ensures that all fat clients are in J'. The following lemma shows that by ignoring non-fat clients, we lose only a negligible fraction of the possible profit.

[2] This is also the case when $\ell > r_k^i$.

Lemma 6. *Let* OPT *denote a solution to some instance $\mathcal{I}$ of r-CC, and let* OPT′ *denote a solution to the same instance, restricted solely to fat clients. Then* $p(\mathrm{OPT}) \leq (1 + \frac{1}{n^2})p(\mathrm{OPT}')$.

Proof. Let p_M denote the maximal profit of any client in J. Since clearly $p(\mathrm{OPT}) \geq p(\mathrm{OPT}') \geq p_M$, it follows that

$$\begin{aligned} p(\mathrm{OPT}) &= p(\mathrm{OPT} \cap J_B) + p(\mathrm{OPT} \cap \overline{J_B}) \\ &\leq p(\mathrm{OPT}') + n \cdot \frac{p_M}{n^3} \\ &= p(\mathrm{OPT}') + \frac{p_M}{n^2} \\ &\leq p(\mathrm{OPT}')(1 + \frac{1}{n^2}), \end{aligned}$$

as required. □

The above lemmas ensure that any α-cover w.r.t. J' guarantees an approximation factor of $\frac{\alpha}{1+\alpha}(\frac{1}{1+1/n^2}) \approx \frac{\alpha}{1+\alpha} \cdot (1 - \frac{1}{n^2})$. Since we assume clients have $(1+\varepsilon)$-rounded profits, by Corollary 1 we lose at most an additional $\frac{1}{1+\varepsilon}$ factor in the approximation factor.

Given any $\gamma \in (\frac{1}{n^2}, 1]$, by considering an appropriate constant $\varepsilon = \varepsilon(\gamma) \in (0,1]$, we can guarantee that the $\frac{1-r}{1+\varepsilon}$-cover produced by dCC is a $\frac{1-r}{2-r}(1-\gamma)$-approximate solution, thus completing the proof of Theorem 2.

As for the running time of algorithm dCC, we use the randomized algorithm of [3]. (This is the only place where randomization is used in our algorithm.) Running that algorithm for $c \log n$ rounds results in a maximal matching with probability at least $(1 - \frac{1}{n^{\Omega(c)}})$. Since the total number of times our algorithm invokes MM is at most n, by the Union Bound it follows that by choosing a sufficiently large constant c, we can guarantee, with high probability, that all executions of MM in Algorithm MC produce a maximal matching.

It follows that if $\max_j \{p(j)/d(j)\}$ is polynomially bounded, then our algorithm runs for $O(\gamma^{-2} \log^3 n)$ rounds, and produces a $\frac{1-r}{2-r}(1-\gamma)$-approximate solution with high probability.

A note on very small and very large values of γ. Note that for $\gamma < 1/n$, we can send the entire network information to every base station in time $O(\gamma^{-2}) = O(n^2)$, in which case every base station may calculate in a centralized manner a deterministic approximate solution. If we are given $\gamma > 1$, we use the algorithm with $\gamma = 1$.

4 Extensions

In this section we present several extensions of our results, whose proofs will appear in the full version.

Clients with Different Classes of Service and Location-Dependent Demands. Our algorithm can be extended to the model where every client j has a specific class of service q_j, and the profit from satisfying a demand d is $q_j d$. Moreover, the client may have a different demand from each possible base station (this may be the case when the requested service is location-dependent): namely, the demand client j has from base station i is $d(j, i)$, and the profit obtained from assigning client j to base station i is $p(j, i) = q_j \cdot d(j, i)$. We can show that even in this general setting, our algorithm provides the same approximation guarantees (with high probability), as well as having the same running time. We note that we still insist that on the condition that a profit is obtained from a client only if its demand is met in full (by one of its neighboring base stations).

Absence of Global Bounds. Our results extend to the model where base stations have no a priori knowledge of the maximal density of the instance (corresponding to the value W in our analysis), and perform on a merely local information basis. In this extension every base station takes on a myopic view of the network, and considers the partition corresponding solely to its neighboring clients. We can show that even in such an asynchronous environment, a simple extension of our algorithm provides the same approximation factors and time complexity.

5 Conclusions and Open Questions

In this work we presented a randomized distributed algorithm for the cellular coverage problem, such that for every $\gamma \in (0, 1]$ our algorithm guarantees to produce a $\frac{1-r}{2-r}(1 - \gamma)$-approximate solution with high probability, in time $O(\gamma^{-2} \log^3 n)$.

There are several interesting questions that arise from our work. First, our work provides a distributed emulation of a centralized local ratio algorithm. It is of great interest to see if similar emulations can be obtained to other problems, where local ratio algorithms provide good approximations. The main elements that seem to facilitate such an emulation are an ordering (or partitioning) of the input implied by the profit decomposition, and a notion of maximality that can be maintained locally. When considering the CC problem, a major goal is to try and improve the running time of a distributed algorithm for solving r-CC. There is currently no reason to believe that a good cover cannot be obtained in logarithmic time. Furthermore, it is interesting to see if there exists a distributed algorithm which makes use of the cover-by-many paradigm, which was shown to provide better solutions than the cover-by-one paradigm. For the more general formulation of the CC problem, it is not evident that one cannot obtain an approximation guarantee that is independent of r. In this respect, we conjecture that even for $r = 1$ the problem admits to constant approximation.

References

1. Amzallag, D., Bar-Yehuda, R., Raz, D., Scalosub, G.: Cell selection in 4G cellular networks. In: Proceedings of IEEE INFOCOM 2008, The 27th Annual Joint Conference of the IEEE Computer and Communications Societies, pp. 700–708 (2008)
2. Peleg, D.: Distributed computing: a locality-sensitive approach. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)
3. Israeli, A., Itai, A.: A fast and simple randomized parallel algorithm for maximal matching. Information Processing Letters 22(2), 77–80 (1986)
4. Hanly, S.V.: An algorithm for combined cell-site selection and power control to maximize cellular spread spectrum capacity. IEEE Journal on Selected Areas in Communications 13(7), 1332–1340 (1995)
5. Mathar, R., Schmeink, M.: Integrated optimal cell site selection and frequency allocation for cellular radio networks. Telecommunication Systems 21, 339–347 (2002)
6. Sang, A., Wang, X., Madihian, M., Gitlin, R.D.: Coordinated load balancing, handoff/cell-site selection, and scheduling in multi-cell packet data systems. In: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MOBICOM), pp. 302–314 (2004)
7. Amzallag, D., Naor, J., Raz, D.: Coping with interference: From maximum coverage to planning cellular networks. In: Proceedings of the 4th Workshop on Approximation and Online Algorithms (WAOA), pp. 29–42 (2006)
8. Dawande, M., Kalagnanam, J., Keskinocak, P., Salman, F.S., Ravi, R.: Approximation algorithms for the multiple knapsack problem with assignment restrictions. Journal of Combinatorial Optimization 4(2), 171–186 (2000)
9. Shmoys, D.B., Tardos, É.: An approximation algorithm for the generalized assignment problem. Mathematical Programming 62, 461–474 (1993)
10. Chekuri, C., Khanna, S.: A PTAS for the multiple knapsack problem. In: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 213–222 (2000)
11. Cohen, R., Katzir, L., Raz, D.: An efficient approximation for the generalized assignment problem. Information Processing Letters 100(4), 162–166 (2006)
12. Lotker, Z., Patt-Shamir, B., Pettie, S.: Improved distributed approximate matching. In: Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 129–136 (2008)
13. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Annals of Discrete Mathematics 25, 27–46 (1985)
14. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Shieber, B.: A unified approach to approximating resource allocation and scheduling. Journal of the ACM 48(5), 1069–1090 (2001)

Fast Geometric Routing with Concurrent Face Traversal

Thomas Clouser, Mark Miyashita, and Mikhail Nesterenko*

Department of Computer Science, Kent State University, Kent, OH 44242, USA
{tclouser,mmiyashi,mikhail}@cs.kent.edu

Abstract. We present a concurrent face routing *CFR* algorithm. We formally prove that the worst case latency of our algorithm is asymptotically optimal. Our simulation results demonstrate that, on average, *CFR* significantly outperforms the best known geometric routing algorithms in the path stretch: the speed of message delivery. Its performance approaches the shortest possible path. *CFR* maintains its advantage over the other algorithms in pure form as well as in combination with greedy routing; on planar as well as on non-planar graphs.

Keywords: Geometric routing, ad hoc wireless routing.

1 Introduction

Geometric routing is an elegant approach to data dissemination in resource-constrained and large-scale ad hoc networks. Geometric routing is attractive because it does not require nodes to maintain, or messages to carry, extensive state or routing information. This lack of routing infrastructure makes such algorithms a popular initialization or fallback option for other routing schemes. Therefore, geometric routing optimization is of interest to the broad community of wireless sensor network designers.

In geometric routing, each node knows its own and its neighbors' coordinates. Using low-cost GPS receivers or location estimation algorithms [1,2], wireless sensor nodes can learn their relative location with respect to the other nodes and then use this information to make routing decisions. The message source node knows the coordinates of the destination node. These coordinates may be obtained from a location service [3,4]. The information that the message can carry does not depend on the network size. Each forwarding node does not maintain any extensive routing data or keep any information about forwarded messages between message transmissions.

Greedy routing [5] is an elementary approach to geometric routing where the node selects the neighbor that is the closest to the destination and forwards the message there. The process repeats until the destination is reached. Greedy routing fails if the node is a *local minimum*: it does not have neighbors that are closer to the destination than itself. Alternatively, in *compass routing* [6], a node selects the neighbor whose direction has the smallest angle to the direction of the destination. This kind of compass routing is prone to livelocks.

One way to circumvent these delivery problems in geometric routing is to flood a region of the network with messages [5,7,8,9,10]. This is useful for *geocasting* [11]

* This author was supported in part by DARPA contract OSU-RF #F33615-01-C-1901 and by NSF CAREER Award 0347485.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 346–362, 2008.

where each node in a certain region of the network needs to receive a message. However, for point-to-point communication flooding may not be efficient.

The *face routing* variants of geometric routing are designed to guarantee message delivery without incurring the message overhead associated with flooding. A source-destination line intersects a finite number of faces of a planar graph. A message may reach the destination by sequentially traversing these faces. In the algorithms published thus far, the faces are traversed sequentially. *GFG*/*GPSR* [12,13] combines greedy and face routing. Greedy routing is used for speed, and face routing helps to recover from local minima. Datta et al [14] propose a number of optimizations to face traversal. Kuhn et al [15,16,17] propose a worst case asymptotically optimal geometric routing algorithm *GOAFR+*. They compare the performance of multiple geometric routing algorithms and demonstrate that in the average case *GOAFR+* also performs the best. Kim et al [18] discuss challenges of geometric routing. Frey and Stojmenovic [19] address some of these challenges and discuss different approaches to geometric routing. Stojmenovic [20] provides a comprehensive taxonomy of geometric routing algorithms.

One of the shortcomings of traditional geometric routing is the need to planarize the graph. This can be done effectively only for unit-disk graphs. However, a unit-disk graph is a poor approximation for most radio networks where radio propagation patterns are not as regular as assumed in unit-disk graphs. Some researchers [21,22] explore a more realistic model of *quasi unit disk graphs*. Nesterenko and Vora [23] propose a technique of traversing voids in non-planar graphs similar to face traversal. This traversal may be combined with greedy routing similar to *GFG*. Barrière et al [21], Kim et al [24], Leong et al [25], and Kuhn et al [22] propose alternative ways of performing geometric routing over non-planar graphs.

Kuhn et al [15,17] conduct extensive evaluation of geometric routing algorithms' performance. They compare the ratio of the path selected by a routing algorithm to the optimal path depending on the graph density. Their findings indicate that at low and high density the performance of most algorithms, especially if combined with greedy routing, approaches optimal. In sparse graphs, due to the limited number of available routes, a geometric routing algorithm is bound to select a route that is close to optimal. In dense graphs, an algorithm nearly always runs in greedy mode which tends to select a nearly optimal route as well. Kuhn et al identified a *critical density* range between 3 and 7 nodes per unit-disk where the paths selected by geometric routing algorithms may substantially differ from the shortest paths and where performance optimization has the greatest impact.

Despite their individual differences, the foundation of most geometric routing algorithms is face traversal. In such traversal, a message is routed around a face. However, the resultant route may vary greatly depending on the choice of traversal direction and the point at which the message switches between adjacent faces. The imbalance is usually exacerbated if the message has to traverse the external face of the graph. However, if the message traverses the faces sequentially, exploring the faces to find a shorter route may result in lengthening to route itself. Hence, traditional geometric routing algorithms are inherently limited in the amount of route optimization they can achieve.

In this paper, we present an algorithm that accelerates the message propagation by sending messages to concurrently traverse faces adjacent to the source-destination line.

We call this algorithm concurrent face routing (*CFR*). When one of the messages encounters a face that is closer to the destination, the message spawns two messages to traverse the new face and continues traversing the old face. *CFR* ensures that all faces are explored and none of the adjacent edges is traversed more than once. The node memory and message-size requirements for *CFR* are the same as for the other geometric routing algorithms. We show that the latency of *CFR* is asymptotically optimal in the worst case. That is, there is no geometric routing algorithm that can deliver a message faster than *CFR*. Moreover, our simulation demonstrates that, on average, *CFR* significantly outperforms other geometric routing algorithms in the critical density region. This average case advantage is preserved if *CFR* is combined with greedy routing or if it runs on non-planar graphs.

The rest of the paper is organized as follows. We introduce our notation in Sect. 2. We then describe *CFR*, formally prove it correct and determine its worst case message complexity in Sect. 3. In Sect. 4, we discuss how the algorithm can be adapted for greedy routing and for use in non-planar graphs. We evaluate the performance of our algorithm and its modifications in Sect. 5 and conclude the paper in Sect. 6.

2 Preliminaries

2.1 Graphs

We model the network as a connected geometric graph $G = (V, E)$. The set of *nodes* (*vertices*) V are embedded in a Euclidean plane and are connected by *edges* E. The graph is *planar* if its edges intersect only at vertices. A *void* is a region on the plane such that any two points in this region can be connected by a curve that does not intersect any of the edges in the graph. Every finite graph has one infinite *external* void. The other voids are internal. A void of a planar graph is a *face*.

2.2 Face Traversal

Each message is a *token*, as its payload is irrelevant to its routing. *Right-hand-rule* face traversal proceeds as follows. If a token arrives to node a from its neighbor b, a examines its neighborhood to find the node c whose edge (a, c) is the next edge after (a, b) in a clockwise manner. Node a forwards the token to c. This mechanism results in the token traversing an internal face in the counter-clockwise direction, or traversing the external face in the clockwise direction. *Left-hand-rule* traversal is similar, except the next-hop neighbor is searched in the opposite direction.

A source node s has a message to transmit to a destination node d. Node s is aware of the Euclidean coordinates of d. Node s attaches its own coordinates as well as those of d to the messages. Thus, every node receiving the message learns about the sd-line that connects the source and the destination. Depending on whether the token is routed using right- or left-hand-rule, it is denoted as R or L. Each node n knows the coordinates of its *neighbors*: the nodes adjacent to n in G. A *juncture* is a node whose adjacent edge intersects the sd-line. A node itself lying on the sd-line is also a juncture. Thus, the source and destination nodes are junctures themselves. Two faces are *adjacent* if their borders share a juncture. A single node may be a juncture to multiple faces if more than one of its adjacent edges intersect the sd-line.

To simplify the algorithm presentation, we use anthropomorphic terms when referring to the nodes of the network such as "know", "learn" or "forget".

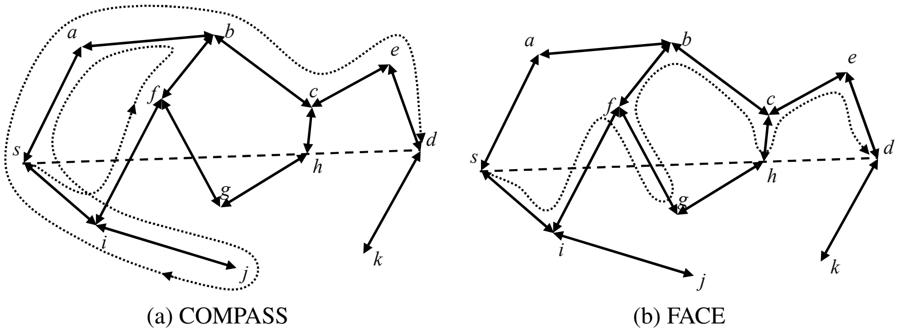
2.3 Performance Metrics

The *message cost* of an algorithm is the largest number of messages that is sent in a single computation calculated in terms of the network graph parameters. A *latency* is the shortest path the message in the algorithm takes to reach the destination. Equivalently, latency is the number of hops the message traverses from the source to destination in accordance with the algorithm. Essentially, message cost captures the expense of communication while the latency captures its speed. For sequential traversal algorithms, such as traditional geometric routing algorithms, where there is always a single message in transit, the two metrics are the same. Note also that the latency of a certain algorithm selects is not necessarily the *optimum* or the shortest path between the source and the destination. A *path stretch* is the ratio between the latency of the algorithm and the shortest path in the graph.

2.4 Existing Face Traversal Mechanisms

One of the first known face routing algorithms that guarantees delivery is Compass Routing II [6]. In this paper we refer to it as *COMPASS*. In *COMPASS*, the token finds the juncture that is closest to the destination. For this, the token traverses the entire face and returns to the initial point of entry. The token is then routed to the discovered closest juncture. There, the token changes faces and the process repeats. Refer to Fig. 1a for an example route selected by *COMPASS*. The message complexity of *COMPASS* is $3|E|$ which is in $O(|E|)$.

In *FACE* [12,14], the token changes faces as soon as it finds the first juncture (refer to Fig. 1b). In degenerate cases, *FACE* allows the token to traverse the same edge multiple times. Hence, its worst case message complexity is in $O(|V|^2)$. It is worse than that of *COMPASS*. However, *FACE* tends to perform better in practice. Both algorithms may



(a) COMPASS

(b) FACE

Fig. 1. Example operation of existing planar face traversal algorithms

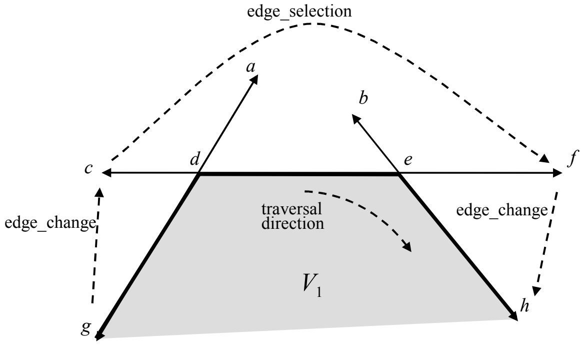


Fig. 2. Traversing non-planar voids

select a route that is far from optimum. The selected route may be particularly long if the token has to traverse the external phase as in the above examples. *OAFR* [17] mitigates long route selection by defining an ellipse around the source-destination pair that the message should not cross. If the message traverses a face and reaches the boundary of the ellipse, the message changes the traversal direction. *OAFR* has the best worst case efficiency for a sequential face traversal algorithm to date. Its path stretch is in $O(\rho^2)$, where ρ is the length of the optimum path.

Algorithms *COMPASS*, *FACE* and *OAFR* operate only on planar graphs. Obtaining such a graph from a graph induced by a general radio network may be problematic. There are several attempts to allow geometric routing on arbitrary non-planar graphs [14,24,25,23]. In particular, Nesterenko and Vora [23] propose to traverse non-planar voids similar to faces. In general the edges that are adjacent to voids do not intersect at the incident vertices. However, the idea is to have the message follow the segments of the edges that are adjacent to the void. Refer to Fig. 2 for illustration.

Each pair of nodes u and v adjacent to an edge (u, v) keeps the information about which edges intersect (u, v) and how to route to the nodes adjacent to these edges. Suppose node g receives the token that traverses void V_1. Node g forwards the token to c in an *edge_change* message. Recall that in a non-planar graph, edges do not have to intersect at nodes. Edges (g, a) and (c, f) intersect at point d. The objective of nodes c and f is to select an edge that intersects (c, f) as close to d as possible. At first c selects an edge and forwards the token with its selection to f in an *edge_change* message. Node f consults its own data, selects edge (b, h) and forwards the token to one of the nodes adjacent to this edge. Thus, the message can completely traverse the void.

Once the void traversal is designed, the various techniques of void-change and exploration can generate the non-planar equivalents of *COMPASS*, *FACE* and *OAFR*.

2.5 Combining Greedy Routing and Face Traversal

GFG/*GPSR* [12,13] improves the quality of route selection by combining face routing with greedy routing. *GOAFR+* [15] does the same for *OAFR*. *GOAFR+* achieves remarkable characteristics. It retains the asymptotic worst case optimality of *OAFR* and achieves the best average case path stretch known to date.

In the combination of greedy routing and face traversal the token has two traversal modes: greedy and face. The token starts in the greedy mode but switches to face mode if it encounters a local minimum (a node with no neighbors closer to the destination). The token continues in the face mode until it finds a node that is closer to the destination than this local minimum. Then the token switches to greedy mode again until another local minimum is discovered.

2.6 Execution Model

To present our algorithm, we place several assumptions on the execution model. We assume that each node can send only one message at a time. The node does not have control as to when the sent message is actually transmitted. After the node appends the message to the send queue SQ, the message may be sent at arbitrary time. Each channel has zero capacity; that is, the sent message leaves SQ of the sender and instantaneously appears at the receiver. Message transmission is reliable (i.e. there is no message loss). The node may examine and modify SQ. We assume that SQ manipulation, including its modification and message transmission, is done *atomically*. We assume that the execution of the algorithm is a sequence of atomic actions. The system is *asynchronous* in the sense that the difference between algorithm execution speed at each node is arbitrary.

3 *CFR* Description, Correctness Proof and Performance Bound Computation

3.1 Description

The pseudocode of *CFR* is shown in Fig. 3. Refer to the pictures in Fig. 4 for the illustration of the algorithm's operation. In the figure, we show three snapshots of a single computation. Thin solid lines denote particular tokens. The tokens are numbered. To reduce clutter in the pictures, we only reproduce token numbers. Thus, token t_5 is only shown as 5. Some tokens are destroyed before they leave their originating node. See for example t_5 or t_9. We denote such tokens by short arrows. In the picture, the face names are for illustration only, the global face names are not available to the incident nodes. The token carries its traversal direction: L or R. When a node receives a token, it can locally determine which adjacent face the token traverses on the basis of its sender and its traversal direction. For example, when node a receives L token t_1 from node s, a knows that t_1 traverses the adjacent face F. Two tokens at a node *match* if they traverse the same face in the opposite directions and at least one of them did not originate in this node. For example, t_6 and t_9 at g as well as t_3 and t_5 at f match. However, t_{11} and t_{12} at h do not match because h originated both of these tokens.

A juncture node can locally determine if an adjacent face locally intersects the sd-line. For example, s knows that F intersects the sd-line while H does not. If a token arrives at a juncture and the token traverses a face that locally intersects the sd-line the juncture node injects a pair of tokens into each other neighboring face that intersects the sd-line. For example, when f receives t_2 traversing F that locally intersects the sd-line, f sends t_5 and t_6 to traverse H, and t_7 and t_8 to traverse G. Similarly when h

```
node $s$
/* let $F$ be a face bordering $s$
    and intersecting the $sd$-line */
add $L(s, d, F)$ to $SQ$
add $R(s, d, F)$ to $SQ$

node $n$
if receive $L(s, d, F)$ then
    if $R(s, d, F) \in SQ$ then
        /* found matching token */
        delete $R(s, d, F)$ from $SQ$
    else
        if $n = d$ then
            deliver $L(s, d, F)$
        if $n$ is a juncture and
          $F$ locally intersects the $sd$-line then
            foreach $F' \neq F$ that locally
                          intersects the $sd$-line do
                add $L(s, d, F')$ to $SQ$
                add $R(s, d, F')$ to $SQ$
        add $L(s, d, F)$ to $SQ$
if receive $R(s, d, F)$ then
    /* handle similar to $L(s, d, F)$ */
```

Fig. 3. Pseudocode of *CFR* at each node

receives t_7, it sends t_{11} and t_{12} to traverse H. A juncture node injects the new tokens only if the token it receives is traversing the face that locally intersects the sd-line. For example, when juncture node c receives t_{14} from e, it just forwards the token to b without injecting tokens into G.

If the destination node receives the token, even though the node delivers it, it processes the token further as an ordinary node. That is, node d forwards the token and injects tokens in adjacent faces if necessary.

3.2 Example Operation

Let us now consider example operation of CFR in the computation in Fig. 4 in detail. Node s initiates the transmission by sending tokens t_1 and t_2 to traverse face F. When t_2 reaches juncture node i, i injects t_3 and t_4 into H and forwards t_2 to f. Node f is also a juncture. Thus, besides forwarding t_2 to b, it injects t_5 and t_6 into H as well as t_7 and t_8 into G. Token t_2 meets a matching token t_1 at b and both tokens are destroyed. This completes the traversal of F. Tokens t_7 and t_8 traverse G and meet in c, where they destroy each other. In the process t_7 reaches all the remaining juncture nodes: g, h and c where the tokens are injected in the adjacent faces. Specifically, t_7 causes the injection of t_9 and t_{10} at g, t_{11} and t_{12} at h and t_{13} and t_{14} at c. All tokens are injected into the external face H. The tokens traversing H find matching tokens and are quickly eliminated at f, g, h and c. Tokens t_4 and t_{14} complete the traversal of H. They arrive at a which destroys them. On its way t_{14} visits d, which delivers it.

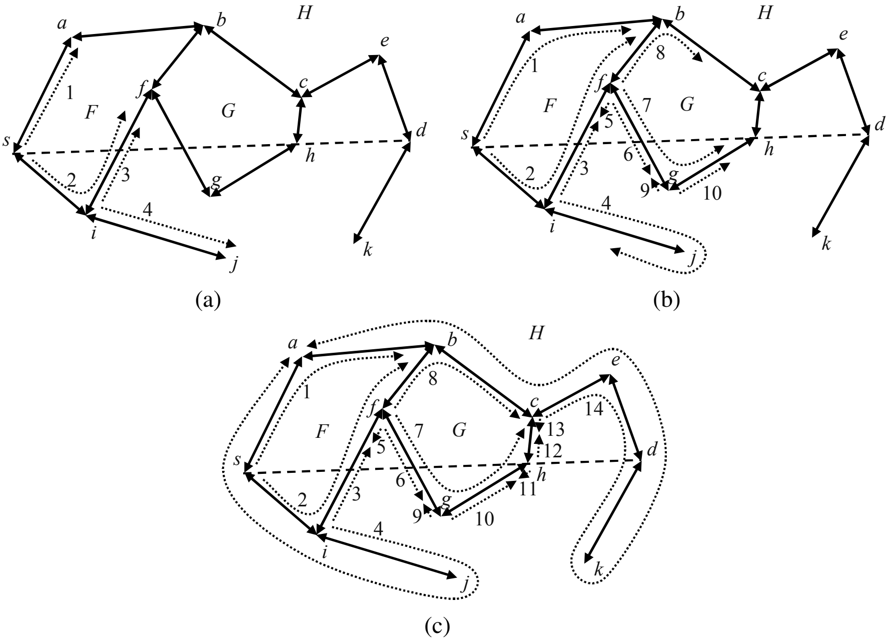


Fig. 4. Example of *CFR* operation on a planar graph

3.3 Correctness Proof

Lemma 1. *For each node n bordering a face F that intersects the sd-line one of the following happens exactly once: either (1) n receives token $T(s, d, F)$ where T is either R or L and forwards it or (2) n has a token, receives a matching token and destroys them both.*

Proof (of lemma). According to the algorithm, a token visits a node and proceeds to the next node along the face, or two matching tokens meet at a node and disappear. Thus, to prove the lemma, we have to show that each node bordering face F is reached and that it is visited only once. A sequence of adjacent nodes of the face is a *visited segment* if each node has been visited at least once. A *border* of a visited segment is a visited node whose neighbor is not visited. By the design of the algorithm, a border node always has a token to send to its neighbor that is not visited. As we assume reliable message transmission, eventually the non-visited neighbor joins the visited segment. Thus, every node in a face with a visited segment is eventually visited.

The face bordering s has at least one visited segment: the one that contains s itself. Thus, every node in this face will eventually be visited. As graph G is connected, there is a sequence of adjacent faces intersecting the sd-line from the face bordering s to the face bordering d. Adjacent faces share a juncture node. Due to the algorithm design, when a juncture is visited in one face that intersects the sd-line, the juncture injects a pair of tokens in every adjacent face. That is, visiting a juncture node creates a visited segment in all adjacent faces. By induction, all nodes in the sequence of adjacent faces are visited, including the destination node.

Let us discuss whether a token may penetrate a visited segment and arrive at an interior (non-border) node. The computation of *CFR* starts with a single visited segment consisting of the source node. Thus, initially, there are no tokens inside any of the visited segments. Assume there are no internal tokens in this computation up to some step x within the visited segment. Let us consider the next step. The token may penetrate the visited segment only through a border node or through an interior junction node. A token may arrive at a border node b only from the border node of another visited segment of the same face. Because b is a border node, it already holds the token of the opposite traversal direction. These two tokens are matching. Thus, b destroys both tokens and the received token does not propagate to the interior nodes. Let us consider a juncture node j. Because j is interior to the visited segment, it was visited earlier. When a juncture node receives a token, it creates a pair of tokens in all adjacent faces. That is, once a juncture is visited, it becomes visited in all adjacent faces at once. Since we assumed that there are no internal tokens up to step x, j cannot receive a token. By induction, a token may not penetrate a visited segment. That is, each node bordering a face is visited at most once. This completes the proof of the lemma. □

The below theorem follows from Lemma 1.

Theorem 1. *Algorithm CFR guarantees the delivery of a message from s to d.*

According to Lemma 1, the total number of messages sent in a computation is equal to the sum of the incident edges of the faces intersecting the sd-line. An edge can be incident to at most two faces. That is, the total number of messages sent throughout the computation is at most $2|E|$. Hence, the following corollary.

Corollary 1. *The worst case message complexity of* CFR *is $O(|E|)$.*

Theorem 2. *The latency of* CFR *is asymptotically optimal and is within $O(\rho^2)$ where ρ is the number of hops in the shortest path in between the source and destination in the planar subgraph of G.*

Proof (of theorem). The theorem's proof parallels the optimality proof of GOAFR [15]. Let us consider the upper bound on the latency first. Kuhn et al argue (see [15, Lemma 5.4]) that to derive a bound it is sufficient to consider a bounded degree traversal graph. If the degree of the graph is unbounded, a bounded degree connected dominating set subgraph can always be locally constructed. Since it takes just one hop to reach this subgraph from any point in the graph, the path length over general graph is only 2 hops more than the length of the path over this subgraph. Let k be the maximum node degree in the traversal graph.

Since the graph to be traversed is a unit-disk graph, if ρ is the number of hops in the shortest path between s and d, then the Euclidean distance between the two points is no more than ρ. Let us consider a disk $D(d, \rho)$ with radius ρ centered in d. Since the shortest path between s and d is no longer than ρ, this path lies completely inside the disk. The shortest path intersects sd-line at least twice: at the source and destination node. Let us consider two consequent points of intersection. Refer to Fig. 5 for illustration.

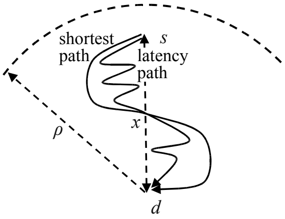


Fig. 5. Illustration for the proof of optimality of *CFR*

Since the graph is planar, the segment of the path between these points, includes the borders of all faces that intersect the sd-line and lie on the same side of the line as the shortest path segment. Since *CFR* traverses these faces, there is a path selected by *CFR* whose segment is completely enclosed by sd-line on one side and this shortest path segment on the other. Examining all such segments of the shortest path, we observe that there is a path of *CFR* that is completely enclosed in the disk $D(d, \rho)$.

Let us estimate the length of this path. Kuhn et al argue (see [15, Figure 2]), that the whole plane can be covered by disks of a diameter of one unit by placing them on a square grid with sides $1/\sqrt{2}$. Let us determine how many such squares cover $D(d, \rho)$. Each square that intersects $D(d, \rho)$ lies completely within $D(d, \rho{+}1)$. Thus, the number of such squares is no more than

$$\frac{\pi(\rho+1)^2}{(1/\sqrt{2})^2} = 2\pi(\rho+1)^2$$

Recall that the graph is unit-disk and all nodes within the unit distance are connected. The graph is of degree k. Thus, the maximum number of nodes in a single disk of diameter one, is k. Therefore, the number of nodes inside $D(d, \rho)$ is no more than $2k\pi(\rho+1)^2$.

There is a path, selected by *CFR* that lies completely inside $D(d, \rho)$. According to Lemma 1 a message of *CFR* can visit the same node at most k times. Thus, the length of this path of *CFR* is no more than $2k^2\pi(\rho+1)^2$ which is in $O(\rho^2)$.

The asymptotic optimality of *CFR* follows from the lower bound established by Kuhn et al [16, Theorem 5.1]. □

4 *CFR* Application and Extensions

4.1 Combining with Greedy Routing, Using Various Traversal Types

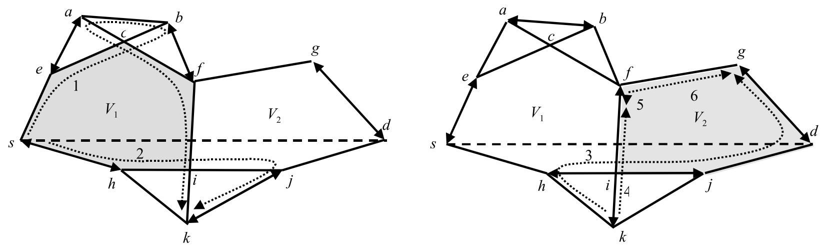
For efficiency, a single direction face traversal may be combined with greedy routing as in *GFG* or *GOAFR+*. Algorithm *CFR* can be used in a similar combination. We call the combined algorithm *GCFR*. The message starts in greedy mode and switches to *CFR* once it reaches a local minimum. Because multiple messages traverse the graph simultaneously, unlike *GFG*, once the message switches to face traversal in *GCFR*, it continues in this mode until the destination is reached.

4.2 Using Non-planar Graphs

CFR can be adapted to concurrent void traversal [23]. The resultant algorithm is *CVR*. *CVR* can also be combined with greedy routing to form *GCVR*. Before we describe the necessary changes let us recall how void traversal operates. Void traversal is performed over segments of edges adjacent to the void, rather than over complete edges. After getting the message, two nodes c and f (see Fig. 2 again), adjacent to the edge (c, f) that contains the segment (d, e), jointly determine the edge whose intersection point produces the shortest segment in the traversal direction. Then, the token is forwarded to one of the nodes adjacent to the new edge (b, h). In the example node f forwards the token to h. In a non-planar graph f and h may be more than one hop apart.

Similar operations happen during the concurrent traversal in *CVR*. However, care must be taken to ensure that mates find each other. In particular a mate traversing the same face might be traveling along the path connecting f and h. Thus, h and f have to agree on the forwarding path and the tokens have to carry enough information to recognize their mates. Another complication to be resolved is the treatment of junctures. For *CVR*, a juncture is the node incident to the edge whose segment intersects the sd-line. Unlike planar graphs, the segments can intersect at points other than nodes. Thus, the segment intersection point itself may potentially lie on the sd-line. This case generates multiple junctures. However, the mates generated by these junctures meet and destroy each other.

Refer to Fig. 6 for an illustration of *CVR* operation. To simplify the presentation we show the traversal of the two adjacent voids V_1 and V_2 separately in Figures 6a and 6b respectively. As before, to avoid cluttering the picture, we only show the token numbers. We explain the traversal of V_1 in detail. The traversal starts when s sends two tokens t_1 and t_2 in the opposite directions around V_1. When t_1 arrives at e, the nodes incident to edge (e, b) have to determine the edge that intersects (e, b) closest to the beginning of the segment. In this case the beginning of the segment is node e itself. Node e sends t_1 to b and the two nodes determine that the appropriate edge is (a, f). Therefore, b forwards t_1 to a which is one of the nodes incident to (a, f). Node a forwards t_1 to f. Node f is a juncture. Hence, f injects a pair of tokens: t_5 and t_6 into V_2. After that, f forwards t_2 to k. Node k is also a juncture. Hence, k injects another pair of tokens: t_3 and t_4 into V_2. Meanwhile, t_1 reaches h. To determine the segment of (h, j) that is adjacent to V_1, h forwards t_1 to j. The intersecting edge correctly determined, j forwards the message



(a) traversing V_1 (b) traversing V_2

Fig. 6. Example of *CVR* operation on a non-planar graph

to k where it meets its mate — t_2. This concludes the traversal of V_1. The traversal of V_2 is completed similarly.

5 Performance Evaluation

5.1 Simulation Environment

To evaluate the performance of *CFR* we recreated the simulation environment used by Kuhn et al [15,17]. For the simulation, we used the graphs formed by uniformly placing the nodes at random on a 20×20 unit square field. The number of nodes depended on the selected density. The edges of the graph were selected according to the unit-disk model: two nodes are connected if and only if they are within the unit-distance of each other. For each graph, a single source and destination pair was randomly selected. We used 21 different density levels. To validate our environment we measured the same preliminary graph parameters as in Kuhn et al [15, Fig. 3],[17, Fig. 3]. For each density level we carried out $2,000$ measurements. Our results are plotted in Fig. 7. They concur with the previous studies.

5.2 Evaluation Description

We implemented *CFR* and compared its performance against the major known geometric routing algorithms. We took $2,000$ measurements at each graph density level. Refer to Fig. 8 for an illustration of the resultant graphs and algorithm path selections.

Let us first compare the speed of communication demonstrated by the routing algorithms. In Fig. 9 we plot the path stretch achieved by the algorithms in pure form and in combination with greedy routing. Figure 9a indicates that pure *CFR* outperforms all the other algorithms. In the critical range, the path stretch that the pure *CFR* provides is up to five times better than the next best algorithm's — *OAFR*. Let us consider the combination of greedy and face routing. Recall that, unlike the other algorithms, after

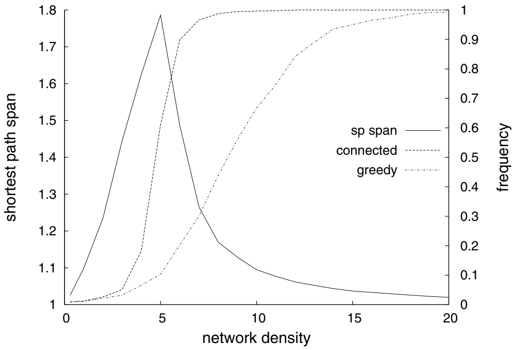
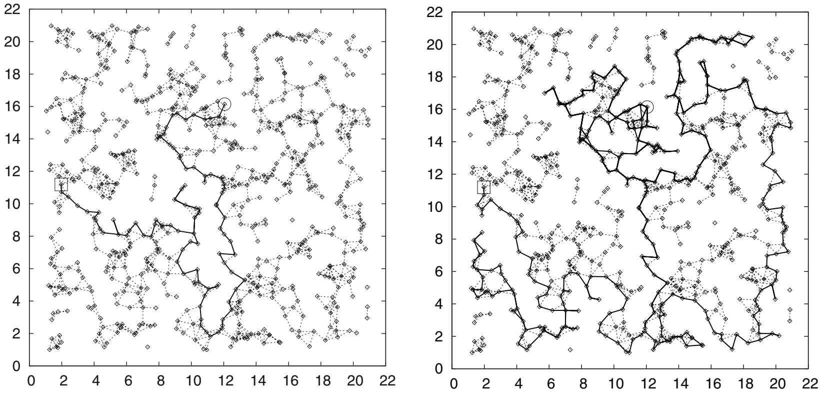


Fig. 7. Graph parameters depending on its density. Shortest path span (ratio between Euclidean and path distance), ratio of connected graphs, and rate of success of pure greedy routing. Last two plotted against the right y axis.



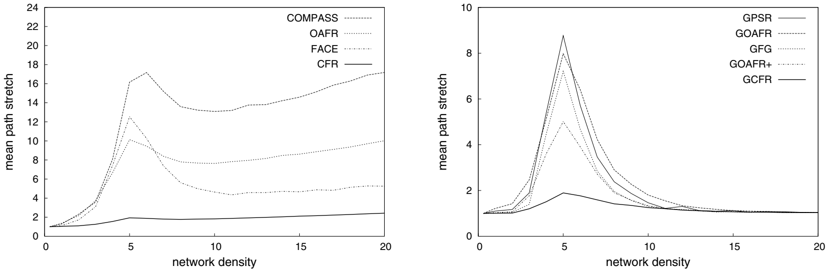
(a) CFR, 102 hops

(b) OAFR, 1051 hops

Fig. 8. Latency paths selected by CFR and OAFR (shown in solid lines). The source and destination nodes are marked by a circle and a square respectively. Graph density is 5 nodes per unit disk.

switching from greedy to face traversal mode, *GCFR* does not switch back to greedy again. Thus, *GCFR* may miss on an efficient path selected by greedy routing. However, as the graph density increases, the greedily routed message may not encounter a local minimum altogether. Therefore, the number of such mode switches decreases and this potential disadvantage of *GCFR* is offset. As Fig. 9b indicates, the path stretch produced by *GCFR* in the critical region is still over 2.5 times better than the next best algorithm.

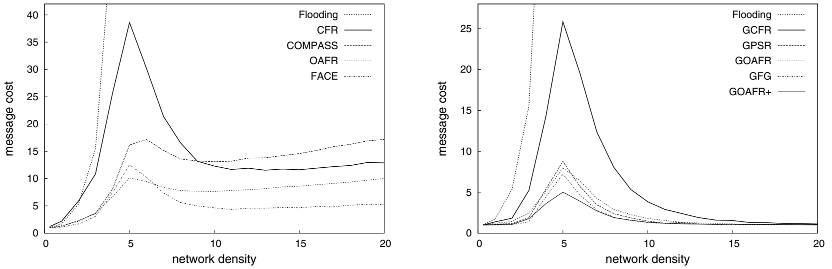
Let us now consider the message cost of communication of the algorithms. In Fig. 10 we show the message cost normalized to the shortest path while in Fig. 11 the cost is normalized to flooding (i.e. every node sends exactly one message). The first presentation indicates the cost compared to the distance from source to destination, the second — compared to the whole system participation in the route discovery. The latter metric gives the perspective of cost of geometric routing compared to flooding-based routing



(a) Pure geometric routing.

(b) Greedy and geometric combined.

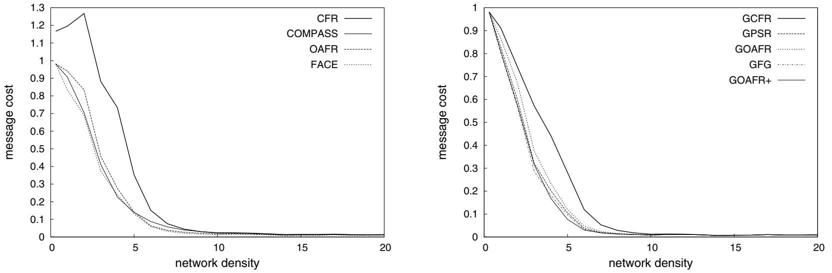
Fig. 9. Mean path stretch (ratio of the path selected by the algorithm to the shortest unit-disk graph path) of geometric routing algorithms on planar graphs depending on the density (nodes per unit disk) of the unit disk graph.



(a) Pure geometric routing.

(b) Greedy and geometric combined.

Fig. 10. Mean message cost normalized to shortest path of geometric routing algorithms on unit-disk graphs depending on density (nodes per unit disk) of the unit disk graph



(a) Pure geometric routing.

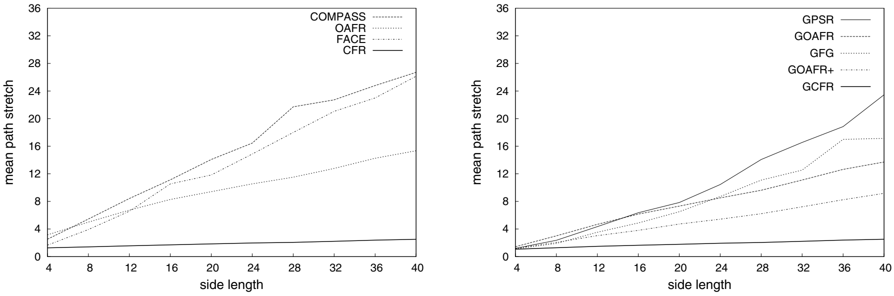
(b) Greedy and geometric combined.

Fig. 11. Mean message cost normalized to flooding of geometric routing algorithms on planar graphs depending on density (nodes per unit disk) of the unit disk graph

algorithms [5,7,8,9,10]. Figure 10 shows that *CFR* and *GCFR* use more messages than other geometric routing algorithms. However, Fig. 11 shows that message cost of *CFR* and *GCFR* are comparable to the other algorithms.

To study the effect of graph scale on the performance of geometric algorithms, we constructed the simulation scenario similar to that of Kuhn et al [17, Fig. 10]. We fixed the density of the graph near the critical value — at 4.5; and varied the field size. Specifically, we selected 10 different lengths of the side of the square field from 4 to 40 units. The number of nodes in the field was selected to match the required density of 4.5. We took 3, 000 measurements for each side length. The results of the simulation are shown in Fig. 12. Our simulation indicates that the path stretch achieved by *CFR* and *GCFR* is lower than that of the other routing algorithms at any scale. This is true for pure geometric routing and its combination with greedy routing. Moreover, as graph scale increases, compared to the other routing algorithms, *CFR* and *GCFR* exhibit significantly slower rate of path stretch increase.

To demonstrate the viability of *CFR* on non-planar graphs, we implemented *CVR* and *GCVR* and compared their performance against conventional *VOID* and *GVG*. For

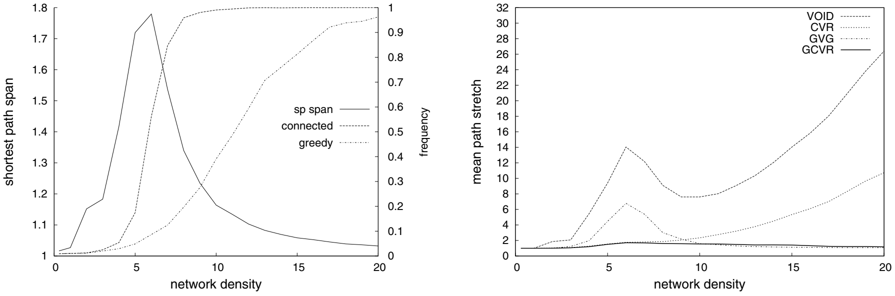


(a) Pure geometric.

(b) Greedy and geometric combined.

Fig. 12. Mean path stretch of routing algorithms on planar subgraphs of unit-disk graphs depending on graph scale with average graph density of 4.5. The graphs are constructed on square fields with side lengths from 4 to 40 units.

these experiments we also used a 20×20 units square field randomly filled by the nodes with randomly selected source and destination pairs. However, the network was modeled as a quasi unit-disk graph [21,22]. Specifically, two vertices of u and v: i) were definitely adjacent if $|u, v| \leq d = 0.75$; ii) were adjacent with probability $p = 0.5$ if $d < |u, v| \leq 1$; iii) definitely not adjacent if $|u, v| > 1$. We selected 21 density levels and carried out $2,000$ trials for each density level. Due to the limitations of double precision floating point calculations, some of the trials did not succeed: due to computation errors, the adjacent nodes may not agree on the edge intersection location. To ensure successful runs, for each graph we globally pre-computed all intersection points. The results are shown in Fig. 13. Our results indicate that *CFR* retains its latency advantages over the other algorithms in non-planar graphs.



(a) Graph parameters depending on its density. Shortest path span, ratio of connected graphs, and rate of success of greedy routing.

(b) Mean path stretch.

Fig. 13. Performance evaluation of geometric routing algorithms on non-planar quasi unit-disk graphs. The distance of definite connectivity is 0.75 unit; possible connectivity between 0.75 and 1 unit; no connectivity above 1 unit.

6 Conclusion

The CFR algorithm presented in this paper improves both the bounds and the practical performance of geometric routing algorithms. Moreover, CFR addresses one of the major drawbacks of geometric routing: its inconsistency due to selection of disadvantageous routes. The proposed technique is simple to implement. The authors are hopeful that it will quickly find its way into practical implementations of geometric routing algorithms.

References

1. Bulusu, N., Heidemann, J., Estrin, D., Tran, T.: Self-configuring localization systems: Design and experimental evaluation. ACM Transactions on Embedded Computing Systems 3(1), 24–60 (2004)
2. Hightower, J., Borriello, G.: Location systems for ubiquitous computing. IEEE Computer 34(8), 57–66 (2001)
3. Abraham, I., Dolev, D., Malkhi, D.: LLS: a locality aware location service for mobile ad hoc networks. In: Proceedings of the Joint Workshop on Foundations of Mobile Computing (DIALM-POMC), Philadelphia, USA, pp. 75–84 (2004)
4. Li, J., Jannotti, J., Couto, D.D., Karger, D., Morris, R.: A scalable location service for geographic ad hoc routing. In: Proceedings of the 6th Annual International Conf. on Mobile Computing and Networking (MobiCom), pp. 120–130 (2000)
5. Finn, G.: Routing and addressing problems in large metropolitan-scale internetworks. Technical Report ISI/RR-87-180 (1987)
6. Kranakis, E., Singh, H., Urrutia, J.: Compass routing on geometric networks. In: Proc. 11 th Canadian Conf. on Computational Geometry, Vancouver, pp. 51–54 (1999)
7. Ko, V., Vaidya, N.: Location-aided routing LAR in mobile ad-hoc networks. In: Proceedings of the 4th Annual ACM/IEEE International Conf. on Mobile Computing and Networking (MobiCom), pp. 66–75 (1998)
8. Ko, Y.B., Vaidya, N.: GeoTORA: A protocol for geocasting in mobile ad hoc networks. In: 8th International Conf. on Network Protocols (ICNP), Osaka, Japan, pp. 240–251 (2000)
9. Seada, K., Helmy, A.: Efficient geocasting with perfect delivery in wireless networks. In: IEEE Wireless Communications and Networking Conference (WNCN 2004), p. 6 (2004)
10. Wu, J., Dai, F.: A generic broadcast protocol in ad hoc networks based on self-pruning. In: 17th International Parallel and Distributed Processing Symposium (IPDPS), April 22–26, 2003, p. 29. IEEE Computer Society, Los Alamitos (2003)
11. Ko, Y.B., Vaidya, N.: Geocasting in mobile ad hoc networks: Location-based multicast algorithms. In: 2nd Workshop on Mobile Computing Systems and Applications (WMCSA), New Orleans, LA, pp. 101–110. IEEE Computer Society, Los Alamitos (1999)
12. Bose, P., Morin, P., Stojmenovic, I., Urrutia, J.: Routing with guaranteed delivery in ad hoc wireless networks. The Journal of Mobile Communication, Computation and Information 7(6), 48–55 (2001)
13. Karp, B., Kung, H.: GPSR: Greedy perimeter stateless routing for wireless networks. In: Proceedings of the Sixth Annual ACM/IEEE International Conf. on Mobille Computing and Networking (MobiCom), pp. 243–254. ACM Press, New York (2000)
14. Datta, S., Stojmenovic, I., Wu, J.: Internal node and shortcut based routing with guaranteed delivery in wireless networks. Cluster Computing 5(2), 169–178 (2002)
15. Kuhn, F., Wattenhofer, R., Zhang, Y., Zollinger, A.: Geometric ad-hoc routing: Of theory and practice. In: 22nd ACM Symp. on the Principles of Distributed Computing (PODC) (2003)

16. Kuhn, F., Wattenhofer, R., Zollinger, A.: Asymptotically optimal geometric mobile ad-hoc routing. In: 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM), Atlanta, Georgia, USA (2002)
17. Kuhn, F., Wattenhofer, R., Zollinger, A.: Worst-case optimal and average-case efficient geometric ad-hoc routing. In: 4th International Symp. on Mobile Ad Hoc Networking and Computing (MobiHoc), Annapolis, Maryland, USA, pp. 267–278 (2003)
18. Kim, Y.J., Govindan, R., Karp, B., Shenker, S.: On the pitfalls of geographic face routing. In: 3rd ACM/SIGMOBILE International Workshop on Foundations of Mobile Computing (DIAL-M-POMC), pp. 34–43 (2005)
19. Frey, H., Stojmenovic, I.: On delivery guarantees of face and combined greedy-face routing in ad hoc and sensor networks. In: Proceedings of the 12th Annual International Conf. on Mobile Computing and Networking, MOBICOM 2006, Los Angeles, CA, USA, September 23-29, 2006, pp. 390–401. ACM, New York (2006)
20. Giordano, S., Stojmenovic, I., Blazevic, L.: Position based routing algorithms for ad hoc networks - a taxonomy. In: Ad Hoc Wireless NetWorking, pp. 103–136 (2004)
21. Barrière, L., Fraigniaud, P., Narayanan, L., Opatrny, J.: Robust position-based routing in wireless ad hoc networks with irregular transmission ranges. Wireless Communications and Mobile Computing 3(2), 141–153 (2003)
22. Kuhn, F., Wattenhofer, R., Zollinger, A.: Ad-hoc networks beyond unit disk graphs. In: Joint Workshop on Foundations of Mobile Computing (DialM-POMC), San Diego, CA, USA, pp. 69–78 (2003)
23. Vora, A., Nesterenko, M.: Void traversal for guaranteed delivery in geometric routing. In: The 2nd IEEE International Conf. on Mobile Ad-hoc and Sensor Systems (MASS 2005), pp. 63–67 (2005)
24. Kim, Y.J., Govindan, R., Karp, B., Shenker, S.: Geographic routing made practical. In: 2nd Symp. on Networked Systems Design and Implementation (NSDI), Boston, MA, USA (2005)
25. Leong, B., Liskov, B., Morris, R.: Geographic routing without planarization. In: 3rd Symp. on Networked Systems Design and Implementation (NSDI), San Jose, CA, USA (2006)

Optimal Deterministic Remote Clock Estimation in Real-Time Systems[*]

Heinrich Moser and Ulrich Schmid

Technische Universität Wien
Embedded Computing Systems Group (E182/2)
A-1040 Vienna, Austria
{moser,s}@ecs.tuwien.ac.at

Abstract. In an OPODIS'06 paper, we laid down the foundations of a real-time distributed computing model (RT-Model) with non-zero duration computing steps, which reconciles correctness proofs and real-time schedulability analysis of distributed algorithms. By applying the RT-Model to the well-known drift-free internal clock synchronization problem, we proved that classic zero step-time analysis sometimes provides too optimistic results. The present paper provides a first step towards worst-case optimal deterministic clock synchronization with drifting clocks in real-time systems, which is an open problem even in classic distributed computing. We define and prove correct an optimal remote clock estimation algorithm, which is a pivotal function in both external and internal clock synchronization, and determine a matching lower bound for the achievable maximum clock reading error in the RT-Model. Moreover, we show how to combine our optimal clock estimation algorithm with existing clock synchronization algorithms.

Keywords: Distributed algorithms, real-time systems, computing models, optimal clock synchronization, remote clock estimation.

1 Introduction

Clock synchronization [1,2,3] is an important and well-studied problem in distributed systems, where processors are equipped with imperfect hardware clocks and connected through a message-passing network. The problem is parameterized by the achievable worst-case synchronization *precision* and comes in two flavors: The goal of *external clock synchronization* is to synchronize all clocks to the clock of a dedicated source processor, whereas *internal clock synchronization* aims at mutually synchronizing all the clocks to each other.

Our research aims at optimal deterministic clock synchronization in distributed *real-time* systems. It differs from traditional distributed computing research by not abstracting away queueing phenomenons and scheduling, but rather quantifying their impact on "classic" results. As a first step towards this goal,

[*] Our research is supported by the Austrian Science Foundation (FWF) under grants P17757 and P20529.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 363–387, 2008.

we revisited the well-known drift- and failure-free internal clock synchronization problem [4] in [5,6]: Utilizing a novel distributed computing model (RT-Model) based on non-zero-duration computing steps, which both facilitates real-time schedulability analysis and retains compatibility with classic distributed computing analysis techniques and results, we proved that the best precision achievable in the real-time computing model is the same as in the classic computing model, namely, $(1 - \frac{1}{n})\varepsilon$, where n is the number of processors and ε is the message delay uncertainty. It turned out, however, that any optimal clock synchronization algorithm has a time complexity of $\Omega(n)$ in the real-time computing model. Since a $\mathrm{O}(1)$ algorithm achieving optimal precision is known for the classic computing model [4], it became apparent that distributed computing results are sometimes too optimistic in the real-time systems context.

The present paper provides a first step towards deterministic optimal-precision[1] clock synchronization in real-time systems with clocks that have non-zero drift, i.e., that do not progress exactly as real-time does. More specifically, we restrict our attention to a (deceptively simple) subproblem of clock synchronization, namely, *remote clock estimation*, in the real-time computing model. Informally, a remote clock estimation algorithm allows processor p to estimate the local clock at processor q at some real-time t, with a known maximum clock reading error. In fact, it is well-known that any existing clock synchronization algorithm can be reviewed in terms of a generic structure [11], which consists of (1) detecting the need for resynchronization, (2) estimating the remote clock values, (3) computing a (fault-tolerant) clock adjustment value, and (4) adjusting the local clock accordingly. Our results on (continuous) remote clock estimation are hence pivotal building blocks for finding and analyzing optimal algorithms for both external and internal clock synchronization in real-time systems.

Related work: Optimal-precision clock synchronization with drifting clocks is an open problem even in classic distributed computing: No optimal algorithms, and only trivial or quite specialized lower bounds are known by now. For example, [12] provides a lower bound for the restricted class of function-based internal clock synchronization algorithms, and [13] provides a lower bound for the precision achievable in a "single-shot" version of clock synchronization ("tick synchronization") in the semi-synchronous model, with inter-step-times $\in [c, 1]$ that can be thought of as being caused by drifting clocks. Optimal results are only available with respect to given message patterns ("passive" clock synchronization) [14,15]; unfortunately, optimal message patterns and hence optimal "active" clock synchronization algorithms cannot be inferred from this research.

Remote clock estimation itself is also handled/analyzed sub-optimally or abstracted away entirely in the wealth of existing research on clock synchronization:

[1] We restrict our attention to worst-case optimality throughout this paper, i.e., we only care about minimizing the maximum precision in the worst execution (rather than in every execution). This is appropriate in the hard real-time systems context, where only the worst case performance matters. For the same reason, we consider deterministic algorithms only; probabilistic clock synchronization [7,8], statistically optimal estimations [9,10] and similar topics are hence out of scope.

Most papers on clock synchronization employ trivial clock estimation algorithms only, based on a one-way or round-trip time-transfer via messages [9], and provide a fairly coarse analysis that (at best) incorporates clock drift [7] and clock granularity [16]. Alternatively, as in [17,12], remote clock estimation is considered an implementation issue and just incorporated via the a-priori given maximum clock reading error. Hence, to the best of our knowledge, optimal deterministic clock estimation has not been addressed in the existing literature.

Contributions: The present paper aims at optimal deterministic clock estimation and related lower bounds in the real-time systems context, where the issue of queueing and scheduling is added to the picture: Utilizing an extension of the real-time computing model (Sect. 2) introduced in [18], which allows to model executions in real-time systems with drifting clocks, we provide an optimal solution for the problem of how to continuously estimate a source processor's clock (Sect. 3). The algorithm is complemented by a matching lower bound on the achievable maximum clock reading error (Sect. 4). Our results precisely quantify the effect of system parameters such as clock drift, message delay uncertainty and step duration on optimal clock estimation. Finally, we show how to incorporate our optimal clock estimation algorithm in existing clock synchronization algorithms (Sect. 5). Some conclusions (Sect. 6) eventually complete our paper.

2 The Real-Time Computing Model

Our system model is based on an extension of the simple real-time distributed computing model introduced in [5], which reconciled the distributed computing and the real-time systems perspective by just replacing instantaneous computing steps with computing steps of non-zero duration. The extended version of the real-time model, introduced in [18], is based on a "microscopic view" of executions termed *state-transition traces*, which allows to define a total order on the *global states* taken on during an execution; this feature is mandatory for properly modeling drifting clocks. Due to lack of space, we will only restate the most important definitions and properties of the extended real-time computing model here; consult [18] for all the details (including a relation to existing computing models).

2.1 Jobs and Messages

We consider a network of n failure-free *processors*, which communicate by passing unique messages. Each processor p is equipped with a CPU, some local memory, a hardware clock $HC_p(t)$, and reliable links to all other processors.

The CPU is running an algorithm, specified as a mapping from processor indices to a set of initial states and a transition function. The *transition function* takes the processor index p, one incoming message, the receiver processor's current local state and hardware clock reading as input, and yields a sequence of *states* and *messages to be sent* (termed *state transition sequence* in the sequel), e.g. $[oldstate, int.st._1, int.st._2, \text{msg. } m \text{ to } q, \text{msg. } m' \text{ to } q', int.st._3, newstate]$, as output. Note that a message to be sent is specified as a pair consisting of the message itself and the destination processor. Since more than one atomic state transition might be required in response to a message, the example above contains three *intermediate states*.

Every message reception causes the receiver processor to change its state and send out all messages according to the state transition sequence provided by the transition function. Such a *computing step* will be called a *job* in the following, and is executed non-preemptively within a system-wide lower bound $\mu^- \geq 0$ and upper bound $\mu^+ < \infty$. We assume that the hardware clock can only be read at the beginning of a job. Note that these assumptions are not overly restrictive, since a job models a (possibly complex) computing step rather than a task in the real-time computing model.

Jobs can be triggered by ordinary messages, timer messages and input messages: *Ordinary messages* are transmitted over the links. The *message delay* δ is the difference between the real time of the *start of the job* sending the message and the real time of the arrival of the message at the receiver. There is a lower bound $\delta^- \geq 0$ and an upper bound $\delta^+ < \infty$ on the message delay of every ordinary message. Since the message delay *uncertainty* is a frequently used value, we will abbreviate it with $\varepsilon := \delta^+ - \delta^-$.

To capture timing differences between sending a single message versus (single-hop) multicasting or broadcasting, the interval boundaries δ^-, δ^+, μ^- and μ^+ can be either constants (e.g. in the case of hardware broadcast) or non-decreasing functions $\{0, \ldots, n-1\} \rightarrow \mathbb{R}^+$, representing a mapping from the number of destination processors to which ordinary messages are sent during a job to the actual message or processing delay bound. The following example shall clarify this: If a job sends ℓ messages to ℓ recipients (with the same or with different content),[2] that job's duration lies between $\mu^-_{(\ell)}$ and $\mu^+_{(\ell)}$ time units. Each of the ℓ messages takes between $\delta^-_{(\ell)}$ and $\delta^+_{(\ell)}$ time units to arrive at the recipient. The delays of these messages need not be the same, however.

Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. Formally, $\forall i, j \geq 1 : f_{(i+j)} \leq f_{(i)} + f_{(j)}$ (for $f = \delta^-$, δ^+, μ^- and μ^+). In addition, we assume that the message delay uncertainty $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ is also non-decreasing and, therefore, $\varepsilon_{(1)}$ is the minimum uncertainty. This assumption is reasonable, as sending more messages usually increases the uncertainty rather than lowering it.

Timer messages are used for modeling time(r)-driven executions in our message-driven setting: A processor setting a timer is modeled as sending a timer message (to itself), and timer expiration is represented by the reception of a timer message. Note that timer messages do not need to obey the message delay bounds, since they are received when the hardware clock reaches (or has already reached) the time specified in the timer message.

Input messages arrive from outside the system. In this paper, input messages are used solely for booting up the system, i.e., for triggering the first job in an execution.

[2] As the message size is not bounded, we can assume that at most one message is sent to the same processor in a job. Hence, there is a one-to-one correspondence between messages and destination processors in each job.

Messages arriving while the processor is busy with some job are *queued*. When and in which order messages collected in the queue are processed is specified by some *scheduling policy*, which is, in general, independent of the algorithm. Formally, a scheduling policy is specified as a mapping from the current queue state (= a sequence of messages), the hardware clock reading, and the current local processor state onto a single message from that message sequence. The scheduling policy is used to select a new message from the queue whenever processing of a job has been completed. We ensure liveness by assuming that the scheduling policy is *non-idling*. To make our results as strong as possible, we will allow the adversary to control the scheduling policy in the algorithm proofs, but we will assume an algorithm-controlled scheduler in the lower bound proof.

Figure 1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. It shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dotted arrows. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not be known to the algorithm. Bounds on the queuing delay ω and the end-to-end delay Δ, however, are *not* parameters of the system model. Rather, those bounds (if they exist) must be derived from the system parameters (n, $[\delta^-, \delta^+]$, $[\mu^-, \mu^+]$) and the message pattern of the algorithm, by performing a real-time schedulability analysis, cp. Sect. 3.4.

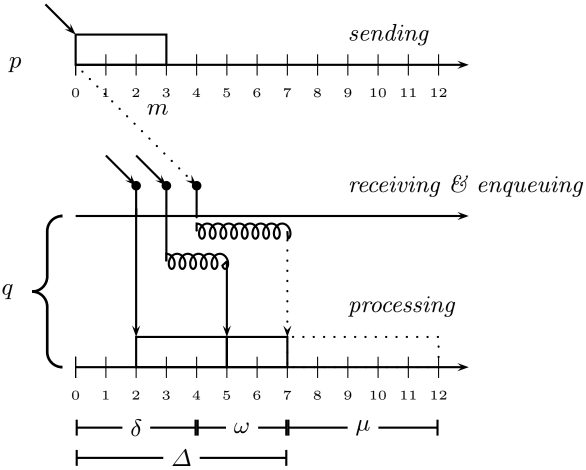


Fig. 1. Timing parameters for some message m

2.2 Hardware Clocks

The hardware clock of any processor p starts with some arbitrary initial value $HC_p(0)$ and then progresses with a bounded drift rate of ρ_p, i.e., t real-time units correspond to at least $(1-\rho_p)t$ and at most $(1+\rho_p)t$ clock-time units. Formally, for all p, $t > t' \geq 0$:

$$(t-t')(1-\rho_p) \leq HC_p(t) - HC_p(t') \leq (t-t')(1+\rho_p)$$

When talking about *time units* in this paper, we mean *real-time units*, unless otherwise noted.

2.3 Real-Time Runs

A *real-time run* (rt-run) is a sequence of receive events and jobs.

A *receive event* R for a message arriving at p at real-time t is a triple consisting of the processor index $proc(R) = p$, the message $msg(R)$, and the arrival real-time $time(R) = t$. Note that t is the receiving/enqueuing time in Fig. 1.

A *job* J starting at real-time t on p is a 6-tuple, consisting of the processor index $proc(J) = p$, the message being processed $msg(J)$, the start time $begin(J) = t$, the job processing time $duration(J)$, the hardware clock reading $HC(J) = HC_p(t)$ when the job starts, and the state transition sequence $trans(J) = [oldstate, \ldots, newstate]$. Let $end(J) = begin(J) + duration(J)$.

Figure 1 provides an example of an rt-run, containing three receive events and three jobs on the second processor q. For example, the dotted job on processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \ldots, newstate])$, with m being the message received during the receive event $(q, m, 4)$. An rt-run is called *admissible*, if all its message delays (measured from the start of the job to the corresponding receive event) stay within $[\delta^-_{(\ell)}, \delta^+_{(\ell)}]$, the duration of all jobs sending ℓ messages is within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$, and the ordering of receive events and jobs does not violate causality (cf. the well-known *happens-before* relation [19]).

If a timer is set during some job J for some time $T < HC_{proc(J)}(end(J))$, the timer message will arrive at time $end(J)$, when J has completed.

2.4 State Transition Traces

The *global state* of a system is composed of the local state s_p of every processor p and the set of not yet processed messages. Rt-runs do not allow a well-defined notion of global states, since they do not fix the exact time of state transitions in a job. This problem is fixed by introducing the "microscopic view" of *state-transition traces* (st-traces) [18], which assign real times to all atomic state transitions.

The following example should provide the required background for understanding the usage of this method, see Appendix A in [20] for more information.

Example 1. Let J with $trans(J) = [oldstate, \text{msg. } m \text{ to } q, int.st._1, newstate]$ be a job in a real-time run ru. If tr is an st-trace of ru, it contains the following *state transition events* (st-events) ev', ev'' and ev''':

- $ev' = (send : t', p, m)$
- $ev'' = (transition : t'', p, oldstate, int.st._1)$
- $ev''' = (transition : t''', p, int.st._1, newstate)$

with $begin(J) \le t' \le t'' \le t''' \le end(J)$. For every time t, there is at least one global state g in tr. Note carefully that tr may contain more than one g with $time(g) = t$. For example, if $t'' = t'''$ in the previous example, three different global states at time t'' would be present in the st-trace, with $s_p(g)$ representing p's state as *oldstate*, $int.st._1$ or *newstate*. Nevertheless, in every st-trace, all st-events and global states are totally ordered by some relation $\prec$.

The time t of a *send* st-event must be such that message causality is not violated, i.e., $t \le begin(J)$, with J being the job processing the message in ru. Recall, however, that the message delay δ is measured from the start of the job sending the message to the receive event in ru. This convention allows a complete schedulability analysis to be done on top of the rt-run, without the need to consider the actual time of state transitions.

3 Optimal Remote Clock Estimation

In this and the following section, we assume a failure-free two-processor system with drifting clocks. Note that remote clock reading is trivially unsolvable in case of just a single crash failure.

3.1 Interval-Based Notation

Often, remote clock estimations are represented by a tuple $(value, margin)$, with *value* representing the expected value of the remote clock and *margin* the absolute deviation from the remote clock's real value, i.e., $remote\_clock \in [value - margin, value + margin]$. With non-drifting clocks, this works well [4,5]. However, consider the two cases in Fig. 2, in which p tries to guess src's value at time t_r by evaluating a timestamped message with delay $\in [\delta^-, \delta^+]$ and clocks with maximum drift ρ_{src} and ρ_p.

In the first case, *src* is a processor with a slow clock and the message is fast; in the second case, *src*'s clock is fast but the message is slow. Thus, at time t_r, *src*'s hardware clock reads $HC_{src}(t_s) + \delta^-(1 - \rho_{src})$ in the first and $HC_{src}(t_s) + \delta^+(1 + \rho_{src})$ in the second case. In the drift-free case ($\rho_{src} = 0$), p can assume that *src*'s clock progressed by $\frac{\delta^- + \delta^+}{2} = \delta^- + \frac{\varepsilon}{2}$ and add this value to $HC_{src}(t_s)$, which is contained in the message. This results in a good estimation of $HC_{src}(t_r)$: It matches the expected value of $HC_{src}(t_r)$, provided message delays are uniformly distributed, with a maximum error margin of $\pm\varepsilon/2$.

In the drifting case, the arithmetic mean of $\delta^-(1 - \rho_{src})$ (= the progress of *src* in the first case) and $\delta^+(1 + \rho_{src})$ (in the second case) is $\delta^- + \frac{\varepsilon}{2}(1 + \rho_{src})$, which is larger than $\delta^- + \frac{\varepsilon}{2}$. Thus, p can either estimate *src*'s clock to be

- $HC_{src}(t_s) + \delta^- + \frac{\varepsilon}{2}(1 + \rho_{src})$, which makes for a nice symmetric error margin of $\pm(\delta^- \rho_{src} + \frac{\varepsilon}{2}(1 + \rho_{src}))$, or

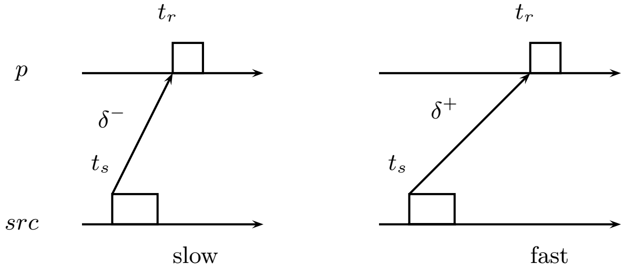


Fig. 2. p receiving a timestamped message from src

- $HC_{src}(t_s) + \delta^- + \frac{\varepsilon}{2}$, which is the expected value, but which has asymmetric error margins $[-(\frac{\varepsilon}{2} + \delta^- \rho_{src}), +(\frac{\varepsilon}{2} + \delta^+ \rho_{src})]$.

To avoid this problem, we assume that p outputs two values est^- and est^+, such that src's real value is guaranteed to be $\in [est^-, est^+]$. Since we want to prove invariants on $[est^-, est^+]$, although there might not be a computation event at every time t, we define $est_p^-(g)$ and $est_p^+(g)$ at some global state g on processor p as functions of the current hardware clock reading, $HC_p(time(g))$, and the current local state $s_p(g)$ of p. Hence, the remote clock estimation problem is formally defined as follows:

Definition 1 (Continuous clock estimation within Γ). *Let src (*source*) and p be processors. Eventually, p must continuously estimate the hardware clock value of src with a* maximum clock reading error Γ. *Formally, for all st-traces tr:*

$$\exists ev_{stable} \in tr : \forall g \succ ev_{stable} :$$
$$HC_{src}(time(g)) \in [est_p^-(g), est_p^+(g)] \wedge est_p^+(g) - est_p^-(g) \leq \Gamma$$

3.2 System Model

The clock estimation algorithm in Sect. 3.3 will continuously send messages from src to p as fast as possible. The following parameters specify the underlying system:

- $[\delta^-, \delta^+]$: Bounds on the message delay.
- $[\mu_{(0)}^-, \mu_{(0)}^+]$: Bounds on the length of a job processing an incoming message, without sending any (non-timer) messages. In the algorithm of Sect. 3.3, all jobs on p fall into this category.
- $[\mu_{(1)}^-, \mu_{(1)}^+]$: Bounds on the length of a job processing an incoming message and sending one message to the other processor. In our algorithm, all jobs on src fall into this category; any such job is triggered by a timer message (or an input message, in case of the first job).

- ρ_p and ρ_{src}: Bounds on the drift of p and src, respectively. We assume $0 \le \rho < 1$, for both $\rho = \rho_p$ and $\rho = \rho_{src}$.

To circumvent pathological cases, we need to assume that

$$\mu^-_{(1)} \ge \mu^+_{(0)} \ . \tag{1}$$

Otherwise, the adversary could create an rt-run in which the "receiving" computing steps at p take longer than the "sending" computing steps at src, causing p's message queue to grow without bound. Note that (1) can also be interpreted as a bandwidth requirement: The maximum data rate of src must not exceed the available processing bandwidth at p (including communication).

3.3 The Algorithm

Consider the algorithm in Fig. 3, which lets src send timestamped messages to p as fast as possible. Processor p determines an estimate for src's clock by using the most recent message from src: While the formula used for the lower error margin est^- is straightforward (est^- increases with a factor ≤ 1 due to p's drift), the fact that the upper error margin est^+ stays constant as soon as the last message from src is older than $(\mu^+_{(0)} + \mu^+_{(1)})(1 - \rho_p)$ might seem counterintuitive, because it means that, as the last message from src gets older, the clock reading error $est^+ - est^-$ of the estimate becomes *smaller* than it was immediately after receiving the message.

The explanation for this phenomenon is that, in a system with reliable links, a lot of information can be gained from *not* receiving a message. As we will show

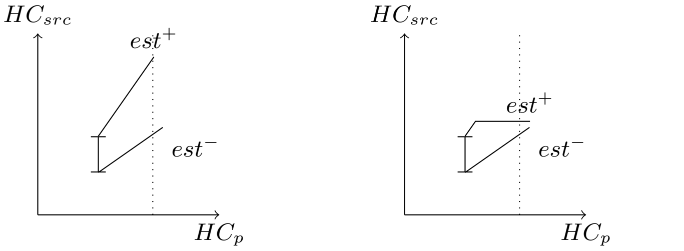
```
Processor src
upon booting or upon receiving ‘‘send now’’:
  var my_hc = current_hc()   /* can only be read at the beginning of the job */
  send my_hc to $p$
  set ‘‘send now’’ timer for my_hc        /* will arrive at $end(current\_job)$ */

Processor p
var rcv_hc ← $-\infty$            /* local time of reception */
var send_hc ← $-\infty$           /* remote time of sending */

function $age$ = current_hc() − rcv_hc
public function $est^-$ = send_hc + $(1 - \rho_{src})\left(\delta^- + age/(1 + \rho_p)\right)$
public function $est^+$ = send_hc + $(1 + \rho_{src})\left(\delta^+ + \min\{\mu^+_{(0)} + \mu^+_{(1)}, age/(1 - \rho_p)\}\right)$

upon receiving a HC value $HC_{src}$ from $src$:
  var my_hc = current_hc()
  if $HC_{src}$ > send_hc:
    rcv_hc ← my_hc; send_hc ← $HC_{src}$        /* one atomic step */
```

Fig. 3. Remote clock estimation algorithm



(a) Naive estimation

(b) Considering future messages

Fig. 4. p's estimate of src's hardware clock

in the next section, the end-to-end delay Δ, i.e., the message delay plus the queuing delay, of every *relevant* message is $\in [\delta^-, \delta^+]$ in the model of Sect. 3.2. If the last message m from src is $\mu^+_{(1)} + x$ time units old (for some $x > 0$, plus $\mu^+_{(0)}$ for processing on the receiver side, plus some drift factor), we know that this message cannot have had an end-to-end delay Δ_m of δ^+. Otherwise, the next message m' from src should have arrived by now. Actually, we know that Δ_m must be within $[\delta^-, \delta^+ - x]$, which is much more accurate than our original assumption of $[\delta^-, \delta^+]$. Clearly, the better p's estimate for Δ_m is, the better p's estimate of src's hardware clock can be.

As can be inferred from Fig. 4, the maximum clock reading error is reached when the message is $(\mu^+_{(0)} + \mu^+_{(1)})(1 - \rho_p)$ hardware clock time units old:

$$\begin{aligned}\Gamma = \max\{est^+ - est^-\} = (1 + \rho_{src})\left(\delta^+ + (\mu^+_{(0)} + \mu^+_{(1)})(1 - \rho_p)/(1 - \rho_p)\right)\\ - (1 - \rho_{src})\left(\delta^- + (\mu^+_{(0)} + \mu^+_{(1)})(1 - \rho_p)/(1 + \rho_p)\right)\end{aligned}$$

Note that $(\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}(1-\rho_{src})$ can be rewritten as $(\mu^+_{(0)} + \mu^+_{(1)})(1-\rho_{src}-2\rho_p)$ $+\,\nu$, with

$$\nu = 2(\mu^+_{(0)} + \mu^+_{(1)})\rho_p \frac{\rho_p + \rho_{src}}{1 + \rho_p} \tag{2}$$

denoting a very small term in the order of $\mathrm{O}(\mu^+\rho^2)$,[3] which is usually negligible. Thus, we have a maximum clock reading error of

$$\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu\ . \tag{3}$$

3.4 Schedulability Analysis

Applying the system model restrictions from Sect. 3.2 to the algorithm allows us to make some general observations:

[3] We use $\mu^+ = \max\{\mu^+_{(0)}, \mu^+_{(1)}\}$ and $\rho = \max\{\rho_{src}, \rho_p\}$ as abbreviations here.

Observation 1. *Every timer set during some job starts processing at the end of that job. Formally, for all timer messages m:* $(m \in \mathit{trans}(J)) \Rightarrow \exists J' : (\mathit{begin}(J') = \mathit{end}(J) \wedge \mathit{msg}(J') = m)$.

Observation 2. *src sends an infinite number of messages to p. The begin times of the jobs sending these messages are between* $\mu^-_{(1)}$ *and* $\mu^+_{(1)}$ *time units apart.*

Given only FIFO links and a FIFO scheduling policy, a simple analysis would show that the end-to-end delay Δ_m, i.e., message (transmission) delay plus queuing delay, is within $[\delta^-, \delta^+]$, for all ordinary messages m. However, in the general setting with non-FIFO links and an arbitrary scheduling policy, it could, for example, be the case that a slow (message delay δ^+) message is "overtaken" by a fast message that was sent later but arrives earlier. If this fast message causes the slow one to be queued, the bound of δ^+ is exceeded. We can, however, solve this problem by filtering (line 10 of the algorithm) "irrelevant" messages, which have been overtaken by faster messages and, thus, might have had a longer end-to-end delay than δ^+.

Of course, one obvious solution would be to put that "filter" inside the scheduler, preventing these irrelevant messages from being enqueued and allowing us to derive the bound $\Delta_m \in [\delta^-, \delta^+]$ by some very simple observations. However, the remainder of this section will demonstrate that this is not necessary: By showing that the bound is satisfied even if every message gets queued and filtering is done within the algorithm, we increase the coverage of our result to systems without low-level admission control.

Let $i \geq 1$ denote the i-th non-timer message sent from *src* (to p). We will show, by induction on i, that a certain bound holds for all messages. This generic bound will allow us to derive the upper bound of δ^+ for the end-to-end delay of relevant messages. First, we need a few definitions:

- J_i: The sending job of message i (on processor *src*).
- J'_i: The processing job of message i (on processor p).
- $\mathcal{F}_i := \{r : \mathit{begin}(J'_r) < \mathit{begin}(J'_i) \wedge r > i\}$: The set of "*fast*" messages $r > i$, that were processed (on p) before i. Informally speaking, this is the set of messages that have overtaken message i. Note that these messages are not necessarily *received* earlier than i, but *processed* earlier.
- $f(i) := \mathit{begin}(J_i) + \delta^+ + \sum_{j \in \mathcal{F}_i \cup \{i\}} \mu^j_{(0)}$. This is an upper bound on the "*finishing*" real time by which all messages $\leq i$ have been processed. $\mu^j_{(0)}$ denotes the actual processing time $\in [\mu^-_{(0)}, \mu^+_{(0)}]$ of message j $(= \mathit{duration}(J'_j))$.

Observe that $f(i) \geq f(i-1)$, since $\mathit{begin}(J_i)$ increases by at least $\mu^-_{(1)}$, whereas at most one message (whose processing takes at most $\mu^+_{(0)}$) "leaves" the set $\mathcal{F}_i \cup \{i\}$.

Lemma 1. *For all i holds: No later than* $f(i)$, *all* J'_j, $1 \leq j \leq i$, *finished processing; formally,* $\mathit{end}(J'_j) \leq f(i)$.

Proof. By induction. For the induction start $i = 0$, the statement is void since there is no job to complete ($f(0)$ can be defined arbitrarily). For the induction step, we can assume that the condition holds for $i - 1 \geq 0$, i.e., that

$$\forall 1 \leq j \leq i - 1 : end(J'_j) \leq f(i - 1) \ . \tag{4}$$

Assume by contradiction that the condition does not hold for i, i.e., that there is some $j \leq i$ such that $end(J'_j) > f(i)$. Since $f(i) \geq f(i - 1)$, choosing some $j < i$ immediately leads to a contradiction with (4). Thus,

$$end(J'_i) > f(i) \ . \tag{5}$$

Assume that $begin(J'_i) \leq begin(J'_{i-1})$. Since $end(J'_i) \leq end(J'_{i-1}) \leq f(i-1) \leq f(i)$ by (4), this leads to a contradiction with (5). Thus, $begin(J'_i) > begin(J'_{i-1})$.

Since J'_i starts later than J'_{i-1}, $\mathcal{F}_{i-1} \subseteq \mathcal{F}_i$ (since $i \notin \mathcal{F}_{i-1}$, and, thus, all $r \in \mathcal{F}_{i-1}$, $r > i - 1$, are also in $\mathcal{F}_i$). Partition $\mathcal{F}_i$ into $\mathcal{F}^{old} = \mathcal{F}_{i-1}$ and $\mathcal{F}^{new} = \mathcal{F}_i \setminus \mathcal{F}_{i-1}$. Note that $f(i) \geq f(i - 1) + \mu^-_{(1)} + \mu^i_{(0)} - \mu^{i-1}_{(0)} + \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$.

Let $t = f(i) - \mu^i_{(0)} - \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$. Note that $t \geq f(i - 1)$, which means that all messages J'_j, $j < i$, and all messages $\in \mathcal{F}^{old}$ have been processed by that time, and that $t \geq begin(J_i) + \delta^+$, which means that message i has arrived by time t. There are two cases, both contradicting (5):

1. There is some idle period in between t and $f(i)$: Since i has arrived by time t, this means that i has already been processed by time $f(i)$, due to our non-idling scheduler.
2. There is no idle period in between t and $f(i)$. Thus, we have a busy period of length $f(i) - t = \mu^i_{(0)} + \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$, which is only used to process messages from $\mathcal{F}^{new}$ and message i (all other messages are done by $f(i - 1)$ due to the induction assumption). This also implies that i gets processed by $f(i)$. □

We call i a "relevant" message if $\mathcal{F}_i = \emptyset$. Thus, the following lemma follows immediately:

Lemma 2. *The end-to-end delay Δ of all relevant messages is* $\in [\delta^-, \delta^+]$.

3.5 Proof of Correctness

Fix some rt-run *ru* and st-trace *tr* and let ev_{stable} be the *transition* st-event of the first relevant message from *src* to *p*. It will be shown that after ev_{stable}, *src*'s hardware clock stays within *p*'s values of est^- and est^+.

Fix some global state $g \succ ev_{stable}$: Let m be the last relevant message from *src* to p fully processed before g, i.e., whose *transition* st-event $\prec g$, with t_j being the time that the job processing the message starts and t_s being the starting time of the sending job of that message. Since $g \succ ev_{stable}$, such a message m must exist. Observe that line 10 in the algorithm ensures that only relevant messages cause a state transition in p.

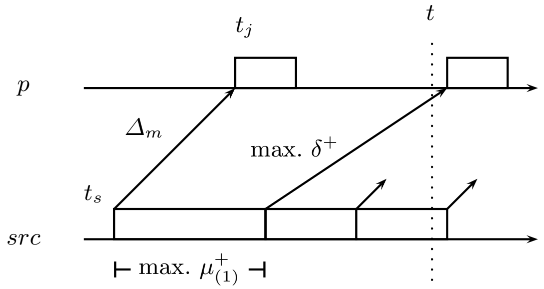


Fig. 5. Two consecutive messages from *src* to *p*

Let $t = time(g)$ and $\Delta_m = t_j - t_s$ (cf. Fig. 5). Note that Δ_m corresponds to δ_m, the message delay, plus any queuing delay m may experience. (For simplicity, Fig. 5 shows a case without queuing.) Due to Lemma 2, we know that Δ_m is bounded by $[\delta^-, \delta^+]$. In addition, we define the following *drift factors*:

$$dr_p = \frac{HC_p(t) - HC_p(t_j)}{t - t_j} \tag{6a}$$

$$dr_{src} = \frac{HC_{src}(t) - HC_{src}(t_s)}{t - t_s} \tag{6b}$$

Clearly, $dr_{src} \in [1 - \rho_{src}, 1 + \rho_{src}]$ and $dr_p \in [1 - \rho_p, 1 + \rho_p]$. These definitions allow us to derive the following by applying (6a) and the definition of Δ_m:

$$\begin{aligned} HC_{src}(t) &= HC_{src}(t_s) + (t - t_s)dr_{src} = HC_{src}(t_s) + ((t - t_j) + (t_j - t_s))dr_{src} \\ &= HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{dr_p} + \Delta_m\right) dr_{src} \enspace . \end{aligned}$$

Since $HC_{src}(t)$ can never become less than the minimum of this expression,

$$\begin{aligned} HC_{src}(t) &\geq \min_{\substack{dr_p \\ dr_{src} \\ \Delta_m}} \left\{ HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{dr_p} + \Delta_m\right) dr_{src} \right\} \\ &= HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{1 + \rho_p} + \delta^-\right)(1 - \rho_{src}) \\ &= s_p(g).send\_hc + \left(age_p(g)/(1 + \rho_p) + \delta^-\right)(1 - \rho_{src}) \enspace . \end{aligned}$$

Hence, we have:

Lemma 3. *$HC_{src}(t) \geq est^-_p(g)$.*

Doing the same for the maximum of the above expression yields a similar result:

Lemma 4. *$HC_{src}(t) \leq s_p(g).send\_hc + (age_p(g)/(1 - \rho_p) + \delta^+)(1 + \rho_{src})$.*

This value is still greater than est^+. Thus, we have to use another approach to prove our upper bound on HC_{src}: First, we note that the real time between t_s and t is bounded:

Lemma 5. $t - t_s \leq \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}$.

Proof. We will again use the numbering of messages as in Sect. 3.4. Recall Fig. 5 and assume by contradiction that $i = m$ was sent earlier, i.e., that $t_s < t - \delta^+ - \mu^+_{(0)} - \mu^+_{(1)}$. Since the (real-time) delay between two consecutive messages sent from src to p is at most $\mu^+_{(1)}$ (cf. Observation 2), $t'_s < t - \delta^+ - \mu^+_{(0)}$ holds for t'_s, the begin time of the job sending $i+1$. Since i is a relevant message, $i+1$ must be processed later than i.

Consider $\mathcal{F}_{i+1}$, the set of messages sent after message $i+1$ but processed earlier; $\mathcal{F}_{i+1}$ might also be $\emptyset$, if $i+1$ is a relevant message. Let J'_{i+1} be the job processing message $i+1$ and let $\mathcal{J} = \mathcal{F}_{i+1} \cup \{i+1\}$. By Lemma 1 we know that $end(J'_{i+1}) \leq f(i+1) = t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu^j_{(0)}$.

Let x be the first message $\in \mathcal{J}$ that will be processed at p. Clearly, x must be a relevant message. Otherwise, there would be some $y > x \geq i+1$ such that J'_y is processed before J'_x. However, if $begin(J'_y) < begin(J'_x) \leq begin(J'_{i+1})$, then $y \in \mathcal{J}$, contradicting the assumption that x is the first message $\in \mathcal{J}$ that will be processed.

We know that all of $\mathcal{J}$ have been processed before $end(J'_{i+1}) \leq t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu^j_{(0)}$ and that processing all of $\mathcal{J}$ takes at least $\sum_{j \in \mathcal{J}} \mu^j_{(0)}$ time units. Thus, at $t'_s + \delta^+$, at least one of $\mathcal{J}$ starts processing, is currently being processed or has already been processed. Since J'_x is the first such job, $begin(J'_x) \leq t'_s + \delta^+$.

Recalling $t'_s < t - \delta^+ - \mu^+_{(0)}$ from the beginning of the proof leads to $begin(J'_x) < t - \mu^+_{(0)}$. Since x is a relevant message and processing x takes at most $\mu^+_{(0)}$ time units, its *transition* st-event is no later than at $t' < t$. This contradicts our assumption that $m = i$ is the last relevant message from src fully processed by p before g. □

Combining Lemma 5 with (6b) results in

$$\frac{HC_{src}(t) - HC_{src}(t_s)}{dr_{src}} \leq \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}$$

and hence

$$\begin{aligned} HC_{src}(t) &\leq HC_{src}(t_s) + (\delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) dr_{src} \\ &\leq \max_{dr_{src}} \left\{ HC_{src}(t_s) + (\delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) dr_{src} \right\} \\ &= HC_{src}(t_s) + (\delta^+ + \mu^+_{(0)} + \mu^+_{(1)})(1 + \rho_{src}) \\ &= s_p(g).send\_hc + (\delta^+ + \mu^+_{(0)} + \mu^+_{(1)})(1 + \rho_{src}) \end{aligned}$$

which, combined with Lemma 4 and the definition of est^+, yields the following result:

Lemma 6. *$HC_{src}(t) \leq est_p^+(g)$.*

Combining Lemma 3 and 6 finally yields Theorem 1, which proves that the algorithm in Fig. 3 indeed solves the remote clock estimation problem according to Definition 1.

Theorem 1 (Correctness). *For all global states $g \succ ev_{stable}$, where ev_{stable} is the transition st-event of the first message from src to p, it holds that $HC_{src}(time(g)) \in [est_p^-(g), est_p^+(g)]$. The maximum clock reading error $\Gamma = \max\{est^+ - est^-\}$ is*

$$\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu,$$

with the usually negligible term $\nu = \mathrm{O}(\mu^+\rho^2)$ given by (2).

4 Lower Bound

In this section, we will show that the upper bound on Γ determined in Theorem 1 is tight, i.e., that the algorithm in Fig. 3 is optimal w.r.t. the maximum clock reading error.

4.1 System Model

For the lower bound proof, we require that $\delta^+(1-\rho) \geq \delta^-(1+\rho)$ and that $\mu^+_{(\ell)}(1-\rho) \geq \mu^-_{(\ell)}(1+\rho)$, for $\ell \in \{0,1\}$ and $\rho \in \{\rho_{src}, \rho_p\}$. These lower bounds on the message and processing delay uncertainties prevent the processors from using their communication subsystems or their schedulers to simulate a clock that has a lower drift rate than their hardware clocks.

To simplify the presentation, we will make three additional assumptions. In Sect. 4.3, we will briefly discuss the consequences of dropping these.

1. $\delta^- \geq \mu^+_{(0)}$. This allows the adversary to choose a scenario where no *send* and/or *transition* st-event in a job occurs earlier than $\mu^+_{(0)}(1-\rho)$ hardware clock time units after the beginning of the job.
2. We assume that the algorithm knows when it has stabilized, i.e., that p switches a Boolean register *stable* (initially false) when the algorithm has stabilized. In the algorithm in Fig. 3, p would set its *stable* register after completing the processing of the first relevant message from *src*.
3. There is at least one message from *src* arriving at p after p has set its *stable* register.

4.2 Proof

Assume by contradiction that there exists some deterministic algorithm $\mathcal{A}$ that allows processor p to continuously estimate processor *src*'s hardware clock with a maximum clock reading error $\max\{est^+ - est^-\} < \Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) +$

$2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu$. Using an adaption of the well-known shifting and drift scaling techniques to st-traces, which is technically quite intricate due to the multiple state transitions involved in a job, we show that there are indistinguishable rt-runs of $\mathcal{A}$ that cause a clock reading error of at least Γ.

Definition 2. *Since our proof uses an indistinguishability argument, we will use the notation $p : tr[ev_A, ev_\Omega] \approx tr'[ev'_A, ev'_\Omega]$ to denote that, for processor p, st-trace tr from st-event ev_A to ev_Ω is indistinguishable from st-trace tr' from st-event ev'_A to ev'_Ω, where ev_A, ev_Ω, ev'_A and ev'_Ω all occur on processor p. Intuitively, this means that p cannot detect a difference between the two st-trace segments.*

Let $(ev_1, ev_2, \ldots, ev_\eta)$ and $(ev'_1, ev'_2, \ldots, ev'_{\eta'})$ be the restrictions of st-traces tr and tr' to send and transition st-events occurring on processor p, beginning with $ev_A = ev_1$ and $ev'_A = ev'_1$, and ending with $ev_\Omega = ev_\eta$ and $ev'_\Omega = ev'_{\eta'}$. Indistinguishability means that $\eta = \eta'$ and $ev_i = ev'_i$ for all $i, 1 \le i \le \eta$, except for the real time of the events, i.e., $time(ev_i) = time(ev'_i)$ is not *required. In fact, indistinguishability is even possible if the st-trace segments are of different real time length, i.e., if $time(ev_\Omega) - time(ev_A) \neq time(ev'_\Omega) - time(ev'_A)$. However, $HC_p^{tr}(time(ev_i)) = HC_p^{tr'}(time(ev'_i))$ must of course be satisfied, i.e., the hardware clock values of all matching st-events must be equal.*

The notations $tr[t_A, ev_\Omega]$, $tr[ev_A, t_\Omega]$ and $tr[t_A, t_\Omega]$ are sometimes used as short forms for $tr[ev_A, ev_\Omega]$, with ev_A being the first st-event with $time(ev_A) \ge t_A$ and ev_Ω being the last st-event with $time(ev_\Omega) \le t_\Omega$. Parenthesis are used to denote $<$ instead of $\le$, e.g. $tr[0, t_\Omega)$.

Likewise, global states are sometimes used as boundaries: $tr[g_A, \ldots]$ and $tr[\ldots, g_\Omega]$ actually mean the first st-event on p succeeding g_A and the last st-event on p preceding g_Ω. Clearly, $s_p(g_\Omega) = s_p(g'_\Omega)$ if $p : tr[\ldots, g_\Omega] \approx tr'[\ldots, g'_\Omega]$.

Note: Since $est^{-/+}$ can be a function of the state of p and the current hardware clock value, it does not suffice to show $s_p(g_1) = s_p(g_2)$ in some indistinguishable st-traces tr_1 and tr_2. If we want to prove that $est^{-/+}$ is equal in both st-traces, we also need to show that $HC_p^{tr_1}(time(g_1)) = HC_p^{tr_2}(time(g_2))$, which is more difficult in our setting than in a drift-free environment.

Let tr_1 be an st-trace of some rt-run ru_1 of $\mathcal{A}$ where the adversary makes the following choices:

- Both processors boot (i.e., receive an initial input message, if required) at time $t = 0$.
- $HC_p(0) = 0$, $HC_{src}(0) = 0$.
- Every message from src takes δ^+ time units.
- Every message to src takes δ^- time units.
- Every job sending ℓ message takes $\mu^+_{(\ell)}$ time units.
- No *transition* or *send* st-event occurs earlier than $\mu^+_{(0)}(1-\rho)$ hardware clock time units after the beginning of the job ($\rho = \rho_p$ for p and $\rho = \rho_{src}$ for src).
- src's clock has a drift factor of $1 + \rho_{src}$
- p's clock has a drift factor of $1 - \rho_p$.

Since $\mathcal{A}$ is a correct algorithm, the execution will eventually become stable. Let $ev_{sta,1}$ be the *transition* st-event at which p switches its *stable* register in tr_1. Let m be an arbitrary message from src to p, sent by a job starting at time t_s and arriving through a receive event at time t_r, with $t_r > time(ev_{sta,1})$. By assumption (cf. Sect. 4.1), such a message exists.

Let tr_2 be an st-trace of another rt-run ru_2 of $\mathcal{A}$ where the adversary behaves exactly as specified for tr_1 (using the same scheduling policy), with the following differences (cf. Fig. 6):

- src boots at time $t = \varepsilon$ (instead of 0).
- $HC_{src}(\varepsilon) = 0$ (instead of $HC_{src}(0) = 0$).
- Every message from src takes δ^- time units (instead of δ^+).
- Every message to src takes δ^+ time units (instead of δ^-).
- After $t_s + \varepsilon$, src's clock has a drift factor of $1 - \rho_{src}$.
- After t_r, p's clock has a drift factor of $1 + \rho_p$.
- After t_r, on p, every job sending ℓ messages takes $\mu^+_{(\ell)}\frac{1-\rho_p}{1+\rho_p}$ time units (instead of $\mu^+_{(\ell)}$). Note that $\mu^+_{(\ell)}\frac{1-\rho_p}{1+\rho_p} \in [\mu^-_{(\ell)}, \mu^+_{(\ell)}]$ (cf. Sect. 4.1). Likewise, *send* and *transition* st-events occur no earlier than $\mu^+_{(0)}\frac{1-\rho_p}{1+\rho_p}$ time units (and hence no earlier than $\mu^+_{(0)}(1-\rho_p)$ hardware clock time units, as in tr_1) after the beginning of their job.[4]

Lemma 7. *$p : tr_1[0, t_r] \approx tr_2[0, t_r]$ and $src : tr_1[0, t_s] \approx tr_2[\varepsilon, t_s + \varepsilon]$.*

Proof. The lemma follows directly from the following observations:

- The initial states are the same in ru_1 and ru_2.
- All st-events within that time occur at the same hardware clock time and in the same order (on each processor).

A formal proof can be obtained by induction on the st-events of ru_1 or ru_2, using these properties, or by adapting any of the well-known "shifting argument" proofs. □

Since $time(ev_{sta,1}) \leq t_r$, this lemma also implies[5] the existence of a corresponding st-event $ev_{sta,2}$ in tr_2, in which p sets its *stable* register.

Lemma 8. *For all $t_1, t_2 \geq t_r : HC_p^{tr_1}(t_1) = HC_p^{tr_2}(t_2) \Leftrightarrow t_2 = (t_1 - t_r)\frac{1-\rho_p}{1+\rho_p} + t_r$.*

Proof. The proof follows directly from the drift factors of p in tr_1 and tr_2, i.e., for all $t_1, t_2 \geq t_r$: $HC_p^{tr_1}(t_1) = t_1(1-\rho_p)$ and $HC_p^{tr_2}(t_2) = t_r(1-\rho_p) + (t_2 - t_r)(1+\rho_p)$.

[4] If there is a job J starting before but ending after t_r, its duration is weighted proportionally, i.e., $duration(J) = (\mu^+_{(\ell)} - x) + x\frac{1-\rho_p}{1+\rho_p}$, with $x = end(J) - t_r$. The same is done with the minimum offset for *send* and *transition* st-events in a job.

[5] This could not be inferred that easily if the algorithm did not know when it had stabilized.

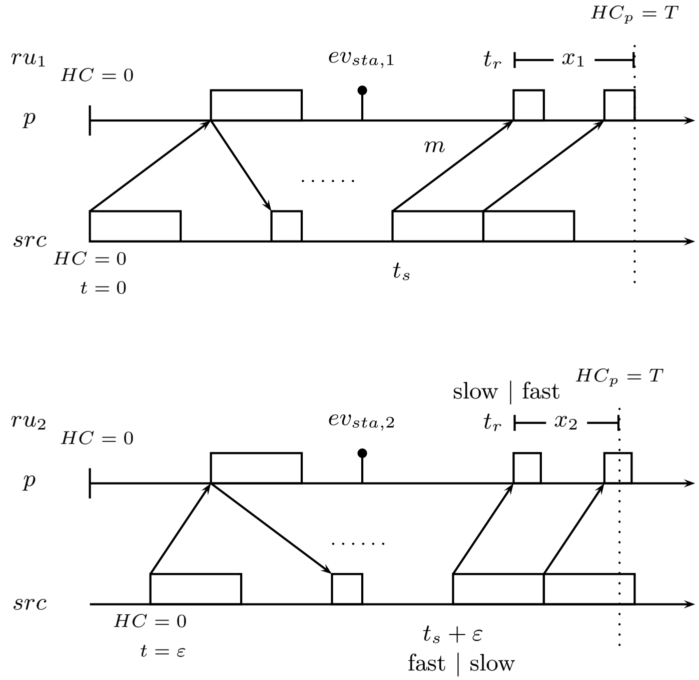


Fig. 6. ru_1 and ru_2 (timer messages not shown); $x_1 = \mu^+_{(0)} + \mu^+_{(1)}$; $x_2 = x_1 \frac{1-\rho_p}{1+\rho_p}$

Let g_1 and g_2 be defined as follows:

- g_1 is the first global state in tr_1 at time $t_r + \mu^+_{(0)} + \mu^+_{(1)}$, i.e., the global state preceding the first st-event (if any) happening at $t_r + \mu^+_{(0)} + \mu^+_{(1)}$.
- g_2 is the first global state in tr_2 at time $t_r + (\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}$.

Clearly, by Lemma 8, p's hardware clock values at g_1 and g_2 are equal (denoted T and represented by the dotted line in Fig. 6).

Lemma 9. $p : tr_1[0, g_1] \approx tr_2[0, g_2]$

Proof. By Lemma 7, tr_1 and tr_2 are indistinguishable for src until t_s and $t_s + \varepsilon$, respectively. Since src starts a job of duration $\mu^+_{(1)}$ in ru_1 at time t_s, a corresponding job is started in ru_2 at time $t_s + \varepsilon$. Both jobs send the same message m to p. Since our system model does not allow preemption, src's next message to p can be sent no earlier than $t_s + \mu^+_{(1)}$ (tr_1) and $t_s + \varepsilon + \mu^+_{(1)}$ (tr_2). Thus, by the definition of message (transmission) delays in ru_1 and ru_2, the earliest time that p can receive another message from src (after the reception of m) is $t_r + \mu^+_{(1)}$ (in both tr_1 and tr_2, cf. Fig. 6).

Thus, the only jobs occurring at p in ru_1 and ru_2 after the reception of m (at time t_r) and before $t_r + \mu^+_{(1)}$ are jobs caused by timer messages, by message m

or by messages that have been received earlier. These messages, however, cannot "break" the indistinguishability: Since (a) p's hardware clock is speeded up and (b) the processing time of jobs on p are slowed down by the same factor $(\frac{1-\rho_p}{1+\rho_p})$, the hardware clock times of all jobs (starting and ending times) as well as all state transitions are equal in tr_1 and tr_2, as long as no new external message reaches p. Since this does not happen before $t_r + \mu^+_{(1)}$, we can conclude that tr_1 and tr_2 are indistinguishable until hardware clock time $T' := HC^{tr_1}_p(t_r + \mu^+_{(1)})$, at which a message might arrive in ru_1 that did not yet arrive in ru_2 (since, in ru_2, only $t_r + \mu^+_{(1)}\frac{1-\rho_p}{1+\rho_p}$ real time units have passed yet at T'). Thus, $p : tr_1[0, t_r + \mu^+_{(1)}) \approx tr_2[0, t_r + \mu^+_{(1)}\frac{1-\rho_p}{1+\rho_p})$.

If a job (J_1 in tr_1, J_2 in tr_2) is currently running at hardware clock time T', a message reception does not change any (future) state transitions of that job, due to no-preemption. Thus, the indistinguishability continues until $T'' := HC^{tr_1}_p(end(J_1)) = HC^{tr_2}_p(end(J_2))$. (If no job was running at hardware clock time T', let $T'' := T'$, cp. Fig. 6.) At hardware clock time T'', the schedulers of ru_1 and ru_2 might choose different jobs to be executed next (since the message from src arrived at different hardware clock times in ru_1 and ru_2). However, due to our assumption that the adversary causes all state transitions to occur no earlier than $\mu^+_{(0)}(1-\rho_p)$ hardware clock time units after the beginning of the job, the state of p is still equal in ru_1 and ru_2 until hardware clock time $T'' + \mu^+_{(0)}(1-\rho_p)$. As $T'' \geq T'$, this corresponds to some real time of at least $t_r + \mu^+_{(0)} + \mu^+_{(1)}$ in tr_1 and at least $t_r + (\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}$ in tr_2. Since g_1 and g_2 are, by definition, the first global states at these real times, no state transition breaking the indistinguishability can have occurred yet. □

Lemma 10. $HC^{tr_1}_{src}(time(g_1)) - HC^{tr_2}_{src}(time(g_2)) = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu$

Proof.

$$\begin{aligned}
HC^{tr_1}_{src}(time(g_1)) &= HC^{tr_1}_{src}(t_r + \mu^+_{(0)} + \mu^+_{(1)}) = HC^{tr_1}_{src}(t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) \\
&= (t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)})(1+\rho_{src}) \\
&= t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)} + \rho_{src}(t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) \\
HC^{tr_2}_{src}(time(g_2)) &= HC^{tr_2}_{src}\left(t_r + (\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}\right) \\
&= HC^{tr_2}_{src}\left(t_s + \varepsilon + \delta^- + (\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}\right) \\
&= HC^{tr_2}_{src}(\varepsilon) + t_s(1+\rho_{src}) + \left(\delta^- + (\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}\right)(1-\rho_{src})
\end{aligned}$$

Again, $(\mu^+_{(0)} + \mu^+_{(1)})\frac{1-\rho_p}{1+\rho_p}(1-\rho_{src})$ can be rewritten as $(\mu^+_{(0)} + \mu^+_{(1)})(1 - \rho_{src} - 2\rho_p) + \nu$, with ν, defined in (2), denoting a small term in the order of $\mathrm{O}(\mu^+\rho^2)$. Thus,

$$HC_{src}^{tr_2}(time(g_2)) =$$
$$t_s + \delta^- + \mu_{(0)}^+ + \mu_{(1)}^+ + \rho_{src}(t_s - \delta^- - \mu_{(0)}^+ - \mu_{(1)}^+) - 2\rho_p(\mu_{(0)}^+ + \mu_{(1)}^+) + \nu \ . \ \Box$$

We can now prove our lower bound theorem:

Theorem 2. *There is no clock estimation algorithm $\mathcal{A}$ that allows processor p to estimate processor src's clock with a maximum clock reading error of less than $\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu$.*

Proof. Lemmas 8 and 9 have shown that $s_p(g_1) = s_p(g_2)$ and $HC_{src}^{tr_1}(time(g_1)) = HC_{src}^{tr_2}(time(g_2))$. Since $est^{-/+}$ on p is a function of the local state and the hardware clock time, it holds that $est_p^-(g_1) = est_p^-(g_2)$ and $est_p^+(g_1) = est_p^+(g_2)$. By the assumption that $\mathcal{A}$ is a correct algorithm which allows p to estimate src's hardware clock with a maximum clock reading error $< \Gamma$, the following condition must hold: $\mathcal{A}$ always maintains two values est^- and est^+, such that

$$HC_{src}^{tr_1}(time(g_1)) \in [est^-, est^+] \qquad \text{and} \qquad HC_{src}^{tr_2}(time(g_2)) \in [est^-, est^+]$$

with $est^+ - est^- < \Gamma$.

Lemma 10 reveals, however, that $HC_{src}^{tr_1}(time(g_1)) - HC_{src}^{tr_2}(time(g_2)) = \Gamma$, which provides the required contradiction. □

4.3 The System Model Revisited

In Sect. 4.1, three assumptions were introduced, which simplify the lower bound proof. In this section, we will briefly discuss the consequences of dropping these assumptions.

1. If we replace the assumption $\delta^- \geq \mu_{(0)}^+$ by the weaker criterion that no *send* or *transition* st-event occurs before $x \cdot (1 - \rho)$ hardware clock time units, with $0 \leq x < \mu_{(0)}^+$ (thereby restricting the adversary's power in tr_1 and tr_2), the precision lower bound is decreased to $\varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(x + \mu_{(1)}^+) - \nu'$, i.e., $\mu_{(0)}^+$ gets replaced by x. Analogously, ν' equals ν with $\mu_{(0)}^+$ replaced by x.
2. If the algorithm need not know when it has stabilized, we must prove that one can always find two st-traces tr_1 and tr_2 where p has stabilized before t_r, recall Fig. 6. Informally, this can be guaranteed due to the fact that even eventual properties are always satisfied within bounded time in a closed model like the RT-Model (where all delays are bounded), see e.g. [21].
3. *There is at least one message from src arriving at p after p has set its stable register.* If this condition is not satisfied, we have two cases:
   Case 1: After p has set its *stable* register, no more messages are exchanged between p and src. In that case, it is trivial to create an indistinguishable rt-run in which p has a different drift rate. Since no messages are exchanged, neither p nor src ever detects a difference between the two rt-runs and we can choose a global state g arbitrarily far in the future to create an arbitrarily large discrepancy between p's estimate and src's hardware clock.

Case 2: After p has set its *stable* register, only messages from p to *src* are sent. In that case, the proof is quite similar to the one in Sect. 4.2. Since only *src* receives messages here, only *src* can detect a difference between two rt-runs with different drift rates. Consider Fig. 6 with the labels p and *src* reversed. In complete analogy to Lemma 9, we can argue that *src* cannot detect a difference until m', the second message from p, has arrived. For p to change its estimate, this information needs to be transmitted back to p.[6] Therefore we have an additional δ^- for the message transmission plus $\mu^-_{(0)}$ (or x, see Assumption 1) required by p until a state transition in response to this message can be performed. Thus, detecting a change in this case takes at least δ^- time units longer than in the case analyzed in Sect. 4.2, finally leading to the same contradiction.

5 Synchronizing Clocks

In this section, we will move from the two-processor clock estimation problem to its application in external and internal clock synchronization.

Since the problems analyzed in this section involve more than two processors, a job may send (non-timer) messages to more than one recipient. Thus, we will also use subscripts (ℓ) on the message delay bounds $\delta^-_{(\ell)}$ and $\delta^+_{(\ell)}$ here, which give the number of recipients to which the sending job sends a message. As detailed in Sect. 2.1, $\delta^-_{(\ell)}$, $\delta^+_{(\ell)}$ as well as $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ are assumed to be non-decreasing w.r.t. ℓ.

5.1 Admission Control

In the classic zero-step-time computing model usually employed in the analysis of distributed algorithms, a Byzantine faulty processor can send an arbitrary number of messages with arbitrary content to all other processors. This "arbitrary number", which is not an issue when assuming zero step times, could cause problems in the real-time model: It would allow a malicious processor to create a huge number of jobs at any of its peers. Consequently, we must ensure that messages from faulty processors do not endanger the liveness of the algorithm at correct processors.

In the following sections, we assume the presence of an *admission control* component in the scheduler or in the network controller, which restricts the set of messages reaching the message queue.

5.2 External Clock Synchronization

In large-scale distributed systems such as the Internet, hierarchical synchronization algorithms like NTP have proven to be very useful. With respect to smaller

[6] Since *src* detected a difference, the rt-runs are no longer indistinguishable. Thus, messages from *src* to p are possible in this (shifted) rt-run.

networks, our results indicate that it pays off to minimize the dominant factor ε, which is severely increased by multi-hop communication. Thus, direct communication between the source and the "clients" will usually lead to tighter synchronization.

For this section, let n specify the number of processors in the system, ρ_{src} the drift rate of the source processor and ρ_* the drift rate of all other processors. The goal is for each processor $p \neq src$ to estimate *src*'s clock as close as possible. The maximum estimation error is called *accuracy* α here. Note that external clock synchronization obviously implies internal clock synchronization with precision $\pi = 2\alpha$.

Consider a variant of the algorithm presented in Sect. 3, where *src* sends its hardware clock value not only to p but to all of the other $n-1$ processors, and the receiver uses the midpoint of $[est^-, est^+]$ as its estimation of *src*'s clock. Admission control is performed by only accepting messages from *src*. An obvious generalization of the analysis in Sect. 3 shows that, if *src* is correct, the worst case accuracy for any correct receiver p is $\alpha = \Gamma/2$ with

$$\Gamma = \varepsilon_{(\ell)} + \rho_{src}(\delta^-_{(\ell)} + \delta^+_{(\ell)}) + 2(\rho_{src} + \rho_*)(\mu^+_{(0)} + \dot{\mu}) - \nu \ ,$$

(cf. Theorem 1), where ℓ depends on the broadcasting method, $\dot{\mu}$ is the transmission period (see below), and $\nu = \mathrm{O}(\dot{\mu}\rho^2)$ refers again to a usually negligible term. The precision achieved by any two correct receivers p, q is hence $\pi = \Gamma$.

In the real-time computing model, the required broadcasting can actually be implemented in two ways:

(a) *src* uses a single job with broadcasting to distribute its clock value. In this case, the duration of each of its jobs is $\in [\mu^-_{(n-1)}, \mu^+_{(n-1)}]$ and the message delay of each message is $\in [\delta^-_{(n-1)}, \delta^+_{(n-1)}]$. Thus, $\ell = n-1$ and $\dot{\mu} = \mu^+_{(n-1)}$.

(b) *src* sends unicast messages to every client, in a sequence of $n-1$ separate jobs that send only one message, i.e., $\ell = 1$. This reduces the message delay uncertainty from $\varepsilon_{(n-1)}$ to $\varepsilon_{(1)}$, but increases the period $\dot{\mu}$ in which every processor p receives *src*'s message from $\mu^+_{(n-1)}$ to $(n-1) \cdot \mu^+_{(1)}$.

5.3 Fault-Tolerant Internal Clock Synchronization

As outlined in the introduction, remote clock estimation is only a small, albeit important, part of the internal clock synchronization problem. In [17], Fetzer and Christian presented an optimal round- and convergence-function-based solution to this problem. They assume the existence of a generic remote clock reading method, which returns the clock value of a remote clock within some symmetric error. Thus, extending their work is a perfect choice for demonstrating the applicability of our optimal clock estimation result in the context of internal clock synchronization.

The algorithm of [17] works as follows: Periodically, at the same logical time at every processor, the current clock values of all other clocks are estimated. These estimates are passed on to a fault-tolerant *convergence function*, which provides a new local clock value that is immediately applied for adjusting the clock. Provided that all clocks are sufficiently synchronized initially and the resynchronization period is chosen sufficiently large, the algorithm maintains a precision of $4\Lambda + 4\rho r_{max} + 2\rho\beta$, where r_{max} denotes the resulting maximum real-time round duration and β the maximum difference of the resynchronization times of different processors. Λ is the maximum clock reading error margin, i.e., $\Lambda = \Gamma/2$ in our setting.

In Appendix B of [20], we present a detailed analysis of how to combine our clock estimation method with their convergence function, resulting in an internal clock synchronization algorithm that tolerates up to f arbitrary faulty processors, for $n > 3f$. The analysis includes a pseudo-code implementation and a correctness proof, which just establishes conditions that guarantee the preconditions of the proofs in [17]. The result is summarized in the following theorem:

Theorem 3. *For a sufficiently large resynchronization period and sufficiently close initial synchronization, fault-tolerant internal clock synchronization can be maintained within* $\pi = 2\Gamma + 4\rho r_{max} + 2\rho\beta$ *with* $\Gamma = \varepsilon_{(n-1)} + \rho(\delta^-_{(n-1)} + \delta^+_{(n-1)}) + 4\rho\mu^+_{(n-1)} - \mathrm{O}(\mu^+\rho^2)$.

6 Conclusions

We presented an algorithm solving the problem of continuous remote clock estimation in the real-time computing model, which guarantees a maximum clock reading error of $\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu$. Using an elaborate shifting and scaling argument, we also established a matching lower bound for the maximum clock reading error. This result leads to some interesting conclusions, which could aid real-time system designers in fine-tuning their systems:

- ε, the message delay uncertainty, dominates everything else, since it is the only parameter that is not scaled down by some clock drift $\rho \ll 1$. This matches previous results on drift-free clock synchronization [5].
- Both sender and receiver clock drift influence the attainable precision. However, the drift of the source clock has a bigger impact, since it affects not only the term involving the processing times $\mu^+_{(0)} + \mu^+_{(1)}$, but also the (potentially larger) term involving message delays.
- Since the presented algorithm does not send messages from p to *src*, we can conclude that round-trips, which are well-known to improve remote clock estimation in the average case, do not improve the attainable worst case error.
- The lower bound implies that optimal precision induces a very high network load: $\mu^+_{(1)}$, the interval, in which messages from *src* to p are sent, is directly proportional to the maximum clock reading error.

Furthermore, we have shown how our optimal clock reading method can be used to solve the problem of external clock synchronization, and how it can be plugged into an existing internal clock synchronization algorithm.

References

1. Ramanathan, P., Shin, K.G., Butler, R.W.: Fault-tolerant clock synchronization in distributed systems. IEEE Computer 23(10), 33–42 (1990)
2. Simons, B., Lundelius-Welch, J., Lynch, N.: An overview of clock synchronization. In: Simons, B., Spector, A.Z. (eds.) Fault-Tolerant Distributed Computing. LNCS, vol. 448, pp. 84–96. Springer, Heidelberg (1990)
3. Schmid, U. (ed.): Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems. J. Real-Time Systems 12(1–3) (1997)
4. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control 62, 190–204 (1984)
5. Moser, H., Schmid, U.: Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 95–109. Springer, Heidelberg (2006)
6. Moser, H., Schmid, U.: Reconciling distributed computing models and real-time systems. In: Proceedings Work in Progress Session of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), Rio de Janeiro, Brazil, pp. 73–76 (December 2006)
7. Cristian, F.: Probabilistic clock synchronization. Distributed Computing 3(3), 146–158 (1989)
8. Arvind, K.: Probabilistic clock synchronization in distributed systems. IEEE Transactions on Parallel and Distributed Systems 5(5), 474–487 (1994)
9. Ellingson, C., Kulpinski, R.: Dissemination of system time. Communications, IEEE Transactions on [legacy, pre - 1988] 21(5), 605–624 (1973)
10. Moon, S., Skelley, P., Towsley, D.: Estimation and removal of clock skew from network delay measurements. In: IEEE INFOCOM 1999 (March 1999)
11. Schneider, F.B.: A paradigm for reliable clock synchronization. In: Proceedings Advanced Seminar of Local Area Networks, Bandol, France, pp. 85–104 (April 1986)
12. Fetzer, C., Cristian, F.: Lower bounds for function based clock synchronization. In: Proceedings 14th ACM Symposium on Principles of Distributed Computing, Ottawa, CA (August 1995)
13. Mavronicolas, M.: An upper and a lower bound for tick synchronization. In: Proceedings Real-Time Systems Symposium, pp. 246–255 (December 1992)
14. Patt-Shamir, B., Rajsbaum, S.: A theory of clock synchronization (extended abstract). In: STOC 1994: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pp. 810–819. ACM Press, New York (1994)
15. Ostrovsky, R., Patt-Shamir, B.: Optimal and efficient clock synchronization under drifting clocks. In: PODC 1999: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, pp. 3–12. ACM, New York (1999)
16. Schmid, U., Schossmaier, K.: Interval-based clock synchronization. Real-Time Systems 12(2), 173–228 (1997)
17. Fetzer, C., Cristian, F.: An optimal internal clock synchronization algorithm. In: Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS 1995), pp. 187–196 (1995)

18. Moser, H.: Towards a real-time distributed computing model. Research Report 17/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, Theoretical Computer Science, Special Issue (2007)
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
20. Moser, H., Schmid, U.: Optimal deterministic remote clock estimation in real-time systems. Research Report 43/2008, Vienna University of Technology, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria (2008)
21. Robinson, P., Schmid, U.: The Asynchronous Bounded-Cycle Model. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 246–262. Springer, Heidelberg (2008)

Power-Aware Real-Time Scheduling upon Dual CPU Type Multiprocessor Platforms

Joël Goossens, Dragomir Milojevic, and Vincent Nélis*

Université Libre de Bruxelles, Brussels, Belgium
{joel.goossens,dragomir.milojevic,vnelis}@ulb.ac.be

Abstract. Nowadays, most of the energy-aware real-time scheduling algorithms belong to the DVFS (Dynamic Voltage and Frequency Scaling) framework. These DVFS algorithms are usually efficient but, in addition to often consider unrealistic assumptions: they do not take into account the current evolution of the processor energy consumption profiles. In this paper, we propose an alternative to the DVFS framework which preserves energy, while considering the emerging technologies. We introduce a dual CPU type multiprocessor platform model (compatible with any general-purpose processor) and a non-DVFS associated methodology which considerably simplifies the energy-aware real-time scheduling problem, while providing significant energy savings.

1 Introduction

1.1 Context of the Study

Hard real-time systems require both functionality correct executions and results that are produced on time. Control of the traffic (ground or air), control of engines, control of chemical and nuclear power plants are just some examples of such systems. These systems are usually modeled by a set of *recurrent* tasks, each one having a computing requirement and a temporal deadline. These deadlines are due to the critical aspect of the applications and, for some systems, a deadline miss may produce fatal consequences (e.g., the Anti-lock Braking System ABS in car). To ensure that all the task deadlines and requirements are satisfied, Real-Time Operating Systems (RTOS) integrate a specific algorithm (the "scheduler") which assigns *priorities* to the tasks (or to their instances) in order to schedule them upon the available processor(s).

Among these applications which impose temporal constraints on the task completion times, some are running on platforms with a limited power reserve such as battery powered devices. Furthermore, many energy-constrained embedded systems are built upon multiprocessor platforms because of their high-computational requirements. As pointed out in [1], another advantage is that multiprocessor systems are more energy efficient than equally powerful uniprocessor platforms, because raising the frequency of a single processor results in a

* Supported by the Belgian National Science Foundation (FNRS) under a FRIA grant.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 388–407, 2008.

multiplicative increase of the energy consumption while adding processors leads to an *additive* increase. Nowadays, as real-time systems become more complex, they are often implemented using *heterogeneous* multiprocessor architectures (i.e., processors do not necessarily have the same architecture) [2]. Hence, the *energy-aware real-time scheduling problem upon heterogeneous platforms* has recently gained in interest in the real-time research domain.

1.2 Related Work

In the scheduling literature, the multiprocessor energy-aware scheduling problem is usually addressed through Dynamic Voltage and Frequency Scaling (DVFS) framework, which consists in reducing the system energy consumption by maintaining both the supply voltage and the operating clock frequency of the processor(s) as low as possible, while ensuring that all the task deadlines are met. The energy-aware scheduling problem *upon heterogeneous platforms* is usually divided into two sub-problems [3]: *the Task Scheduling and Voltage Scaling problem* (TSVS) and the *Task Mapping Improvement problem* (TMI).

The TSVS problem assumes that a mapping of the tasks upon the processors is known beforehand, where each task is assigned to only one processor. For a given task mapping, this problem consists in *determining the schedule/order of the tasks executions on each CPU and their assignment to the various voltage levels so as to minimize the total energy consumption.* Currently, most of the papers [4,5,6,7,8,9,10] focus on this TSVS sub-problem, while considering different task models. In [11], the authors formulate the task mapping and TSVS problems as a mixed integer linear programming with discrete CPU voltage levels and propose a relaxation heuristic. There are also few papers [12,13,14] which assume that the task mapping and task ordering are known and focus only on the voltage scaling problem.

The TMI problem consists in *iteratively improving the task mapping*, based on the system schedulability and the energy consumption of the produced schedule.

The authors of [3] consider the TSVS and TMI problems together in order to derive a feasible and energy-efficient schedule. Leung et al. [15] formulate the whole problem of task mapping, task ordering and voltage scaling as a mixed integer non-linear programming problem with continuous CPU voltage levels. In [16] Schmitz et al. propose a strategy that also considers task mapping, task ordering and voltage scaling, where the priorities of the tasks are generated using a genetic algorithm and voltage scaling of the tasks is done using [14].

1.3 Drawbacks of the DVFS Framework

Usually, the DVFS algorithms are efficient but their authors often consider unrealistic assumptions. For instance, it is often assumed that decreasing the CPUs operating frequency always reduces their energy consumption. But experimentally, this not necessarily true since several operating frequencies may be associated to the *same* supply voltage. It is also often assumed that the frequency of a processor may be set to any non-integer values, or that the time delay involved by a modification of the supply voltage is negligible. Moreover, DVFS strategies obviously require DVFS-capable processors and, although a lot of such processor architectures have been proposed in past years, only a few of them have been used in embedded systems. One major reason is that many DVFS processors involve large mass production cost including test cost, design cost and the cost of on-chip DC-DC converters [17].

The main drawback of the DVFS algorithms is that most of them *do not take into account the "static part"* of the processors energy consumption (due to the leakage currents and subthreshold currents) and only reduce the "dynamic part" of their consumption (due to the processor activity). However, this static part has become more and more significant as feature sizes has become smaller and threshold levels lower [18]. Since the dynamic part of the CPUs consumption becomes less important as the static part grows, the energy savings provided by the DVFS algorithms become less and less significant. Nowadays, as technologies scale down to the UDSM (ultra deep-submicron), *the static power dissipation becomes a significant component of the total power consumption* (see for instance [19,20] or [21] in Section 1.1.4 pages 18–21), and cannot be neglected anymore.

1.4 This Research

Although this research does not introduce new scheduling algorithms or novel formal proofs, it aims to address new solutions to the emerging problem of static power consumption. Nowadays, *"leakage power dissipation is becoming a huge factor challenging a continuous success of CMOS technology in the semiconductor industry (...) innovations in leakage control and management are urgently needed"* [18]. In this research, we so propose *an alternative to the DVFS framework* for reducing the energy consumption, while taking into account the static consumption. Our proposal:

1. is compatible with any general-purpose processor,
2. provides important energy savings by effectively employing the features of our proposed non-DVFS platform model and the characteristic of the real-time application,
3. considerably simplifies the energy-aware scheduling problem, in regard with the existing solutions cited above.

On the contrary to the existing solutions which are typically computational-intensive and/or provide low energy savings, our methodology is based on some well-know optimization heuristics and may therefore benefit (depending on the selected parameters of the heuristics) from a low complexity while providing significant energy savings.

Paper organization. This paper is structured as follows. Section 2 defines the computational models used throughout the paper. In Sect. 3, we first introduce our non-DVFS methodology and we explain how our proposed platform and methodology reach the three objectives described above. Then, we formalize the considered problem, and we address the multiple issues of optimal and approximated approaches to solve it. Section 4 illustrates the experimental results provided by all the methods discussed in Sect. 3 and finally, Sect. 5 presents our conclusions and future work.

2 Model of Computation

2.1 Hardware Model

In this research, we introduce dual CPU type Multiprocessor System-on-Chip (MPSoC) platform illustrated in Fig. 1 and referred in the following as DMP. MPSoC systems are widely used in the design of high-performance and low-power embedded systems as it has been already suggested in the literature [2]. A DMP is composed of two MPSoC platforms denoted L_{lp} and L_{hp}, containing M_{lp} and M_{hp} processors respectively. Each processor has a small amount of the local memory (L1 cache memory), allowing the processor to operate independently, at least for a certain period of time, before fetching the data from the possibly L2 cache memory and/or distant (off-chip) central memory. All processors in the DMP are identical in terms of their architecture, i.e. their hardware specification are derived from the same Register Transfer Logic (RTL) description in some Hardware Description Language (HDL), such as VHDL or Verilog. However, for the physical synthesis of the processors (integrated circuit manufacturing) that bel ong to the L_{lp} (called the lp-*processors*), a Low-Power, Low-Performance technology library is used. The resulting integrated circuit can be operated with lower power supply voltages and consequently lower operating clock frequency, resulting in lower dynamic power dissipation. Inversely, processors in L_{hp} (called the hp-*processors*) are physically synthesized using High-Power, High-Performance technology library (for the same or different technological node). Such circuit allows higher operating frequency, but will require higher supply voltage, resulting in higher dynamic power dissipation than the lp-processors. In this way, while each MPSoC platform can be seen as *homogeneous* multiprocessor platform, the DMP as a whole appear as a special case of a *heterogeneous* multiprocessor system. Notice that prior researches [22,23,24] demonstrated that, compared to homogeneous ones, heterogeneous architectures deliver higher performance at lower costs in terms of die area and power consumption. Moreover, there is no restriction here on the processors purposes, hence achieving the objective 1.

Different processors, within the same MPSoC platform, communicate between themselves using the Network-on-Chip (NoC) communication paradigm. In recent years, the NoCs have gained many interest in the research community and are more and more used to replace traditional communication solutions based on shared buses and their variates [25,26]. When compared to buses, the NoCs profit from higher bandwidth capacities because of their concurrent operation and much lower and predictable latencies, allowing efficient processor-to-processor communication in our case. NoCs are flexible, scalable and even energy efficient [27,28] for systems containing more than ten nodes (processors and/or memories).

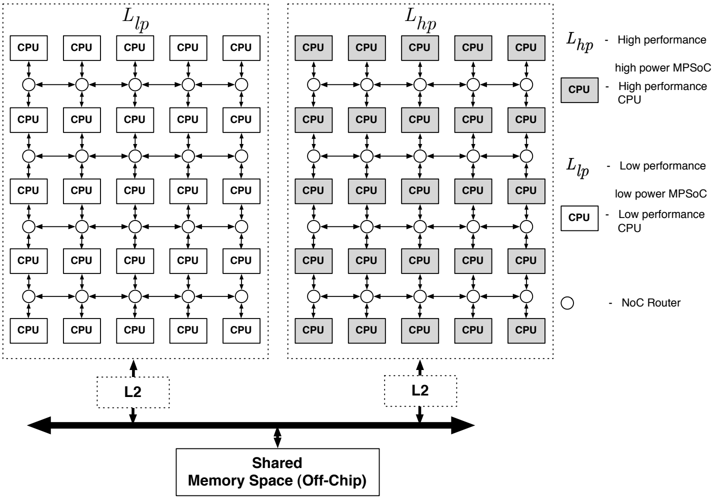

L_{lp}
L_{hp}
CPU
L2
L2
Shared Memory Space (Off-Chip)
L_{hp} - High performance high power MPSoC
CPU - High performance CPU
L_{lp} - Low performance low power MPSoC
CPU - Low performance CPU
○ - NoC Router

Fig. 1. Our considered DMP model with 2 × (5 × 5) processors

In DMP, each MPSoC is built using regular, fully connected mesh NoC. That is: each microprocessor is connected to one router through a Network Interface. Each router is connected to its 4 direct neighbors (note that the routers at the edges are connected, so that they form a torus). Of course other NoC topologies could be explored, for more efficient (lower latency) inter processor communication, but this is out of the scope of the paper. The two MPSoC systems are assumed to be *independent* and they cannot communicate directly. However, since they share the same central memory, the communication is still possible, although not in a very efficient way. But this is not critical in this "first-step" study since we assume in the following that the two MPSoC platforms do not communicate. A thorough reflection about the many possible communication technologies between the two platforms L_{lp} and L_{hp} will be addressed and exploited in our future work.

An x-processor (where x is either lp or hp) is characterized by two parameters: its estimated energy consumption e^{x}_{busy} *while it is busy and is running at its maximal clock frequency*; and its estimated energy consumption e^{x}_{idle} *while it is idle.* Notice that in our study, *all the supplied processors always run at their maximal operating clock frequency and both their supply voltage and clock frequency are never modified.*

2.2 Application Model

We consider in this paper the scheduling of n *sporadic constrained-deadline tasks*, i.e., systems where each task $\tau_i = (C_i^{\mathrm{lp}}, C_i^{\mathrm{hp}}, D_i, T_i)$ is characterized by four parameters – a worst-case execution time at maximal frequency C_i^{lp} and C_i^{hp} upon the lp- and hp-processors, respectively, a minimal inter-arrival delay T_i and a deadline $D_i \leq T_i$ – with the interpretation that the task generates successive *jobs* $\tau_{i,j}$ (with $j = 1, \ldots, \infty$) arriving at times $e_{i,j}$ such that $e_{i,j} \geq e_{i,j-1} + T_i$ (with $e_{i,1} \geq 0$), each such job has an execution requirement of at most C_i^{lp} or C_i^{hp} upon L_{lp} or L_{hp}, respectively, and must be completed at (or before) its deadline noted $D_{i,j} = e_{i,j} + D_i$. The tasks are assumed to be independent, i.e. there is no communication, no precedence constraint and no shared resource (except the processors) between them. We now introduce some preliminary definitions.

Utilization. We define the *utilization* U_i^x of a task τ_i upon an x-processor as the ratio between its worst-case execution time upon an x-processor running at maximal frequency and its period: $U_i^x \overset{\mathrm{def}}{=} \frac{C_i^x}{T_i}$. The *total utilization* $U_{\mathrm{sum}}^x(\tau)$ and the *largest utilization* $U_{\mathrm{max}}^x(\tau)$ of a set of tasks τ allocated to a MPSoC platform L_x are defined as follows: $U_{\mathrm{sum}}^x(\tau) \overset{\mathrm{def}}{=} \sum_{\tau_i \in \tau} U_i^x$ and $U_{\mathrm{max}}^x(\tau) \overset{\mathrm{def}}{=} \max_{\tau_i \in \tau}(U_i^x)$. The utilization U_i^x of τ_i denote the highest proportion of time during which an x-processor could be used to achieve all the execution requirements of τ_i.

Density. We define the *density* δ_i^x of a task τ_i upon an x-processor as the ratio between its worst-case execution time upon an x-processor running at maximal frequency and its deadline: $\delta_i^x \overset{\mathrm{def}}{=} \frac{C_i^x}{D_i}$. The *total density* $\delta_{\mathrm{sum}}^x(\tau)$ and the largest density $\delta_{\mathrm{max}}^x(\tau)$ of a set of tasks τ allocated to a MPSoC platform L_x are defined as follows: $\delta_{\mathrm{sum}}^x(\tau) \overset{\mathrm{def}}{=} \sum_{\tau_i \in \tau} \delta_i^x$ and $\delta_{\mathrm{max}}^x(\tau) \overset{\mathrm{def}}{=} \max_{\tau_i \in \tau}(\delta_i^x)$. We assume that $C_i^{\mathrm{lp}} \geq C_j^{\mathrm{lp}}$ *iff* $C_i^{\mathrm{hp}} \geq C_j^{\mathrm{hp}}$ $\forall$ τ_i, τ_j in order to get a total order on the tasks requirements, and we assume without loss of generality that the tasks are indexed by decreasing order of their density, i.e. $\delta_1^x \geq \delta_2^x \geq \ldots \geq \delta_n^x$, and consequently $\delta_{\mathrm{max}}^x(\tau) = \delta_1^x$. Notice that a task τ_i such that $\delta_i^{\mathrm{lp}} > 1$ *cannot be completed in time* while being executed upon a lp-processor. As a result, such tasks *must mandatorily be assigned to the MPSoC platform* L_{hp} (assuming that $\delta_i^{\mathrm{hp}} \leq 1 \, \forall \tau_i$).

Demand bound function. For any time interval of length t, the demand bound function $\mathrm{DBF}^x(\tau_i, t)$ of a sporadic task τ_i allocated to a MPSoC platform L_x bounds the maximum cumulative execution requirements by jobs of τ_i that both arrive in, and have deadline within, any interval of length t. It has been shown in [29] that

$$\mathrm{DBF}^x(\tau_i, t) = \max\left\{0, (\lfloor \frac{t - D_i}{T_i} \rfloor + 1) \cdot C_i^x \right\} \ .$$

Load. A load parameter, based upon the DBF function, may be defined for any set of tasks τ allocated to a MPSoC platform L_x as follows:

$$\mathrm{LOAD}^x(\tau) \overset{\mathrm{def}}{=} \max_{t>0} \left\{ \frac{\sum_{\tau_i \in \tau} \mathrm{DBF}^x(\tau_i, t)}{t} \right\} \ .$$

As it is mentioned in [30], the DBF and the LOAD functions can be computed *exactly* [29,31], or *approximately* [32,33,34] to any arbitrary degree of accuracy in pseudo-polynomial time or polynomial time, respectively.

2.3 Scheduling Specifications

Both MPSoC platforms L_{lp} and L_{hp} uses its own global multiprocessor scheduling algorithm. "Global" scheduling algorithms, on the contrary to partitioned algorithms, allow different tasks and different instances of the same task (called job in the following) to be executed upon *different* processors. In our study, each task is statically assigned to one of the two MPSoC platforms (L_{lp} or L_{hp}) during the design of the real-time system. Every job can start its execution on any processor of its assigned platform L_x and *may migrate at run-time* to any other processor of L_x (with no loss or penalty) if it gets meanwhile preempted by an higher-priority job. However, tasks and jobs are not allowed to migrate between the two platforms since we consider that there is no communication between them.

We consider in this paper two popular multiprocessor scheduling algorithms: the *preemptive global Deadline Monotonic* and the *preemptive global Earliest Deadline First* [35]. For sake of simplicity, these algorithms will henceforth be referred as global-DM and global-EDF in the remainder of this document. Global-DM assigns a static (i.e., constant) priority to *each task* during the design of the system. The Deadline Monotonic (DM) assigns priority to tasks according to their (relative) deadlines: the smaller the deadline, the greater the priority. On the other hand, EDF assigns priority to *jobs* according to their (absolute) deadlines: the earlier the deadline, the greater the priority. There are other (popular) global scheduling strategies, e.g., PFAIR strategies [36], Least Laxity First scheduler [37] or LLREF [38], but they are limited to *implicit-deadline* tasks (i.e., $D_i = T_i \; \forall i$). As a result, they are not considered in this paper.

3 Off-Line Task Mapping

3.1 Our Methodology

In this study, *we focus on the energy consumption of a* DMP *while executing real-time systems*. Since a task consumes a lower amount of energy while executed upon L_{lp} than upon L_{hp}, the main idea for minimizing the energy consumption of a DMP is to maximize the workload onto L_{lp}. *During the design of the real-time system*, we thus address the problem of *how to partition the set of real-time tasks upon the two platforms* L_{lp} *and* L_{hp} *for the given workload, so that the tasks are completed in time with an energy consumption as low as possible.* The mapping of the tasks that we determine assigns each task to one of the two MPSoC platforms, not to a specific processor. During the execution of the system, the tasks are scheduled upon their assigned platform using a global scheduling algorithm (possibly different on each platform). Notice that a DVFS algorithm considering homogeneous multiprocessor platforms may be used on both L_{lp} and L_{hp} to further reduce their energy consumption.

Once the task mapping is established, *only a subset of the CPUs are supplied in both* L_{lp} *and* L_{hp}, ensuring that all their assigned tasks can be completed in time (i.e. without missing any deadlines) during the system execution. The unnecessary processors in both platforms are not supplied in order to switch off their leakage (i.e. static) consumption. For a given task mapping, the resulting hardware configuration of the DMP is therefore energy-optimized for the targeted real-time application system (objective 2). We can assume that for some kinds of practical applications, these unsupplied CPUs may be turned on later during the system execution in order to replace a defective supplied processor, to handle an emergency, etc.

Moreover, our strategy of switching off some processors may also be very useful to other concerns of the real-time domain, such as the energy-aware scheduling problem of *multi-mode* real-time systems (see e.g., [39] for a detailed description of such systems). The DMP architecture and our task mapping algorithms, enable a task mapping for the set of tasks of *each* mode and then carry out a task mapping switch when a mode change is requested. Since each task mapping has its own associated numbers of supplied (and unsupplied) CPUs in the platforms L_{lp} and L_{hp}, it may be needed to turn on/off some processors during a task mapping switch. For such multi-mode real-time systems, the hardware configuration of the DMP could therefore be energy-optimized for each mode of the application. Although this kind of real-time systems is not studied in this paper, it justifies our hardware architecture and the fact that some CPUs are switched off. The interested reader may refer to [40] for a description of how a transition from one mode to another is ensured.

The schedulability of the two task sets τ^{lp} and τ^{hp} upon L_{lp} and L_{hp} (resp.) is checked thanks to *schedulability tests*, i.e. conditions associated to a scheduling algorithm which indicates (based on the tasks and platform characteristics) whether the given set of tasks can be scheduled upon the given platform without missing any deadline. These conditions must be sufficient or necessary and sufficient. Currently, several *schedulability tests* have already been established for the global scheduling algorithms, the task model and the homogeneous MPSoC platform model considered in this paper. From a practical point of view, the use of a global scheduling algorithm on each platform L_{lp} and L_{hp} involves that both of them may be seen as a single computing element. The task mapping determination problem may therefore be compared to a 2-bin-packing problem (objective 3), and we will show in Sect. 4 that *even some popular bin-packing heuristics may provide a relevant power savings.*

3.2 Notations

Off-line task mapping is thus the process of determining, *during the design of the real-time application system* τ, the partitioning of the tasks set into two subsets of tasks τ^{lp} and τ^{hp} leading to a minimal (or so) consumption. These subsets are such that $\tau^{\mathrm{lp}} \cup \tau^{\mathrm{hp}} = \tau$ and $\tau^{\mathrm{lp}} \cap \tau^{\mathrm{hp}} = \phi$. The subset τ^{lp} contains the tasks which will be executed upon the MPSoC platform L_{lp} and symmetrically, τ^{hp} contains the tasks which will be executed upon L_{hp}. We call such partition $\{\tau^{\mathrm{lp}}, \tau^{\mathrm{hp}}\}$ a *task mapping*.

We denote by S and S' the global scheduling algorithms used on L_{lp} and L_{hp}, respectively (where S and S' are either global-DM or global-EDF), and we will use the notation Y to denote any of these two scheduling algorithms, whatever the platform which uses it. We denote by τ any set of sporadic constrained-deadline tasks. We denote by $\text{CPU}_x^Y(\tau)$ *the function returning a sufficient number of x-processors so that the set of tasks τ can be scheduled by Y without missing any deadline.* Once a task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$ is determined for a given real-time system τ, the sufficient numbers of processors supplied on L_{lp} and L_{hp} are given by $\text{CPU}_{\text{lp}}^S(\tau^{\text{lp}})$ and $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}})$, respectively. Unfortunately, no *necessary and sufficient* schedulability test is known for global-DM and global-EDF in order to determine the *minimal* number of required CPUs to schedule a given sporadic constrained-deadline task system. Fortunately, *sufficient* tests exist.

3.3 Schedulability Test for Global-DM

For the global-DM algorithm, we use the following *sufficient* schedulability condition from [35].

Test 1. *A set of sporadic constrained-deadline tasks τ is guaranteed to be schedulable upon an identical multiprocessor platform L_x composed of m processors using preemptive global-DM if, for every task τ_k,*

$$\frac{\sum_{\tau_i \in \tau \mid D_i < D_k} \alpha_{i,k}^x(\tau)}{1 - \delta_k^x} \le m$$

$$\text{where } \alpha_{i,k}^x(\tau) = \begin{cases} U_i^x \cdot (1 + \frac{T_i - C_i^x}{D_k}) & \text{if } \delta_{\max}^x(\tau) \ge U_i^x \\ U_i^x \cdot (1 + \frac{T_i - C_i^x}{D_k}) + \frac{C_i^x - \delta_{\max}^x(\tau) \cdot T_i}{D_k} & \text{otherwise.} \end{cases}$$

From Test 1, we derive the function $\text{CPU}_x^{\text{DM}}(\tau)$ returning a sufficient number of x-processors in order to schedule the set of tasks τ while meeting all the task deadlines:

$$\text{CPU}_x^{\text{DM}}(\tau) \overset{\text{def}}{=} \max_{\tau_k \in \tau} \left\{ \left\lceil \frac{\sum_{\tau_i \in \tau \mid D_i < D_k} \alpha_{i,k}^x(\tau)}{1 - \delta_k^x} \right\rceil \right\}$$

where $\alpha_{i,k}^x(\tau)$ is defined as previously.

3.4 Schedulability Test for Global-EDF

The global-EDF algorithm has been widely studied in the literature and several (incomparable) sufficient schedulability conditions were already established. In this paper, we use a combination of the most popular ones (Inequalities (1), (2) and (3)) that we have gathered in Test 2. Notice that we have expressed these three inequalities in the number m of required processors.

Test 2. *A set of sporadic constrained-deadline tasks τ is global-EDF schedulable upon an identical multiprocessor platform L_x composed of m processors, provided GFB condition [41]:*

$$m \ge \frac{\delta_{\text{sum}}^x(\tau) - \delta_{\max}^x(\tau)}{1 - \delta_{\max}^x(\tau)} \ , \qquad (1)$$

or BAK condition [35]:

$$m \geq \max_{\tau_k \in \tau} \left\{ \frac{\sum_{\tau_i \in \tau} \min\{1, \beta_i^x\} - \delta_k^x}{1 - \delta_k^x} \right\} \quad , \tag{2}$$

$$\textit{where } \beta_i^x = \begin{cases} U_i^x \cdot (1 + \frac{T_i - D_i}{D_k}) & \textit{if } \delta_k^x \geq U_i^x \\ U_i^x \cdot (1 + \frac{T_i - D_i}{D_k}) + \frac{C_i^x - \delta_k^x \cdot T_i}{D_k} & \textit{if } \delta_k^x < U_i^x \end{cases}$$

or BB condition [30]:

$$m \geq \frac{\text{LOAD}^x(\tau) - 1}{(1 - \delta_{\max}^x(\tau))^2} + 1 \quad , \tag{3}$$

where $\mu^x \stackrel{\text{def}}{=} m - (m-1) \cdot \delta_{\max}^x(\tau)$.

The function $\text{CPU}_x^{\mathsf{EDF}}(\tau)$ returning a sufficient number of x-processors in order to schedule a set of tasks τ is defined as $\text{CPU}_x^{\mathsf{EDF}}(\tau) \stackrel{\text{def}}{=} \min\{\mathsf{GFB}_x(\tau), \mathsf{BAK}_x(\tau), \mathsf{BB}_x(\tau)\}$ where $\mathsf{GFB}_x(\tau)$, $\mathsf{BAK}_x(\tau)$ and $\mathsf{BB}_x(\tau)$ are respectively defined from the three conditions[4] of Test 2:

$$\begin{aligned} \mathsf{GFB}_x(\tau) &\stackrel{\text{def}}{=} \left\lceil \tfrac{\delta_{\text{sum}}^x(\tau) - \delta_{\max}^x(\tau)}{1 - \delta_{\max}^x(\tau)} \right\rceil \\ \mathsf{BAK}_x(\tau) &\stackrel{\text{def}}{=} \max_{\tau_k \in \tau} \left\{ \left\lceil \tfrac{\sum_{\tau_i \in \tau} \min\{1, \beta_i^x\} - \delta_k^x}{1 - \delta_k^x} \right\rceil \right\} \\ \mathsf{BB}_x(\tau) &\stackrel{\text{def}}{=} \left\lceil \tfrac{\text{LOAD}^x(\tau) - 1}{(1 - \delta_{\max}^x(\tau))^2} + 1 \right\rceil \end{aligned}$$

3.5 Energy Consumption Function

For a given set of tasks τ and a given global scheduling algorithm Y, we formulate *the energy consumption of an homogeneous platform L_x while executing τ by Y* as follows:

$$E_x^Y(\tau) \stackrel{\text{def}}{=} U_{\text{sum}}^x(\tau) \cdot e_{\text{busy}}^x + (\text{CPU}_x^Y(\tau) - U_{\text{sum}}^x(\tau)) \cdot e_{\text{idle}}^x \quad . \tag{4}$$

Remember that $\text{CPU}_x^Y(\tau)$ is the number of processors *supplied* in L_x. $U_{\text{sum}}^x(\tau)$ represents an upper bound on the proportion of time during which the supplied CPUs of L_x will be busy, considering an infinite interval of time. Consequently, $(\text{CPU}_x^Y(\tau) - U_{\text{sum}}^x(\tau))$ represents a lower bound on the proportion of time during which the $\text{CPU}_x^Y(\tau)$ supplied processors of L_x will be idle. It results that $E_x^Y(\tau)$ is an upper bound on the energy consumption of the platform L_x while executing τ by Y. For a given DMP and a given task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$, we define the energy consumption of the whole system as follows:

$$E^{S,S'}(\tau^{\text{lp}}, \tau^{\text{hp}}) \stackrel{\text{def}}{=} E_{\text{lp}}^S(\tau^{\text{lp}}) + E_{\text{hp}}^{S'}(\tau^{\text{hp}}) \tag{5}$$

where S and S' denote the schedulers employed by L_{lp} and L_{hp} respectively.

[4] Notice that, for some infeasible real-time systems, one (or more) of these three functions can get negative. A negative value would distort the result from $\text{CPU}_x^{\mathsf{EDF}}(\tau)$ and hence, every function which get negative are ignored while computing the minimum value in $\text{CPU}_x^{\mathsf{EDF}}(\tau)$ (we can also consider that these functions $\mathsf{GFB}_x(\tau)$, $\mathsf{BAK}_x(\tau)$ and $\mathsf{BB}_x(\tau)$ return ∞ if negative).

3.6 Problem Formulation

The problem of minimizing the energy consumption of a DMP while executing a sporadic constrained-deadline task system can be stated as follows.

Formulation of the problem. Let π be a DMP composed of two platforms L_{lp} and L_{hp}, composed of M_{lp} lp-processors and M_{hp} hp-processors, respectively. Let S and S' be the global scheduling algorithms used by L_{lp} and L_{hp} (resp.), for which a sufficient schedulability test based on the number of required CPUs is known. For a given task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$, let $\text{CPU}_{\text{lp}}^{S}(\tau^{\text{lp}})$ and $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}})$ be the functions returning a sufficient number of lp- and hp-processors (resp.) in order to schedule the task sets τ^{lp} and τ^{hp} (resp) by S and S' (resp.) without missing any deadline. Let τ be a given sporadic constrained-deadline task system. The goal of this study is to determine a task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$ (where $\tau^{\text{lp}} \cup \tau^{\text{hp}} = \tau$ and $\tau^{\text{lp}} \cap \tau^{\text{hp}} = \phi$) such that:

1. $\text{CPU}_{\text{lp}}^{S}(\tau^{\text{lp}}) \leq M_{\text{lp}}$
2. $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}}) \leq M_{\text{hp}}$
3. $E^{S,S'}(\tau^{\text{lp}}, \tau^{\text{hp}})$ *is low as possible.*

As it was mentioned above, to find a task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$ may be seen as a bin-packing problem. The two subsets τ^{lp} and τ^{hp} represent two bins. Each task τ_i represents an item whose the weight depends on the container (τ^{lp} or τ^{hp}). The functions $\text{CPU}_{\text{lp}}^{S}(\tau^{\text{lp}})$ and $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}})$ give *the capacity* of each bin and M_{lp} and M_{hp} are their respective *total capacity*. We left open, for future work, the problem of formulating this problem as an *Integer Linear Programming*.

The optimal task mapping (or one of them) may be found by using an exhaustive search, but the worst-case computing complexity of such a search exponentially grows with the number of tasks in the system. As a result, we address in the following section, four popular heuristics for solving the problem described above with an reasonable computing complexity. Notice that all the task mapping methods that we propose are applied *at design time.*

3.7 Approximation Algorithms

A task mapping is denoted in this section as a vector V of n elements, where n is the number of tasks in the system. The sets τ^{lp} and τ^{hp} of the task mapping V are henceforth defined as the set of tasks τ_i for which $V[i] = \text{lp}$ and $V[i] = \text{hp}$, respectively. We define an *admissible task mapping* as a task mapping for which:

1. $\nexists i \in [1, n]$ such that $\delta_i^{\text{lp}} > 1$ and $V[i] = \text{lp}$
2. $\text{CPU}_{\text{lp}}^{S}(\tau^{\text{lp}}) \leq M_{\text{lp}}$
3. $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}}) \leq M_{\text{hp}}$

Random Algorithm (RA). This is the simplest way to get one solution: a task mapping is randomly generated (with probability 0.5 to choose lp or hp for each task), with no guarantee to be admissible. If admissible, the task mapping is returned. Otherwise, the empty task mapping $\{\phi, \phi\}$ is returned.

First-Fit Decreasing Density (FFDD). To approximate (one of) the optimal task mapping(s), we consider the popular *First-Fit Decreasing Density* bin-packing heuristic. *"Decreasing Density"* means that tasks are considered (and then placed into a bin) by decreasing order of their density. Initially the set τ^{lp} is empty and the set τ^{hp} contains every task τ_i such that $\delta_i^{\text{lp}} > 1$. Then, every remaining task τ_j is placed in τ^{lp} if $\text{CPU}_{\text{lp}}^{S}(\tau^{\text{lp}} \cup \{\tau_j\}) \leq M_{\text{lp}}$ or otherwise, in τ^{hp} if $\text{CPU}_{\text{hp}}^{S'}(\tau^{\text{hp}} \cup \{\tau_j\}) \leq M_{\text{hp}}$. If a task cannot be placed in either τ^{lp} or τ^{hp}, FFDD stops and returns the empty task mapping $\{\phi, \phi\}$.

Unfortunately, FFDD and RA are not very efficient (see Sect. 4.3 for details), especially if the number of tasks is high. Hence, we consider in the following two more efficient heuristics: a *Genetic Algorithm* (GA) [42] and a *Simulated Annealing* (SA) [43].

Genetic Algorithm (GA). Due to the space limitation, we do not explain in this paper how a GA works (the reader may consult [42] for details). However, we describe here how we have implemented the different steps of this heuristic. Suppose that we have a target DMP noted π, which runs the global scheduling algorithms S and S' on L_{lp} and L_{hp}, respectively, and a given sporadic constrained-deadline task system τ.

Initialization step. During the initialization process, 50000 task mappings of τ are randomly generated to form an initial population of 50 admissible task mappings. If the algorithm does not find any admissible task mapping after 50000 tentatives, it returns the empty task mapping $\{\phi, \phi\}$. If it finds less than 50 admissible task mappings, it returns the best one that it found. Otherwise, if the population size (50) is reached, the algorithm sequentially repeats the following steps.

Selection step. Every task mapping in the current population is rated by the function $E^{S,S'}(\tau^{\text{lp}}, \tau^{\text{hp}})$. The lower the task mapping rate is, the higher its probability to be selected is. The selection method that we used is the popular and well-studied *roulette wheel* method [42].

Reproduction step. The goal of this step is to generate *a next population of admissible task mappings* from the current population. This is achieved through the *crossover* operator. This operator takes as argument two task mappings V_1 and V_2 (selected in *the current population* through the roulette wheel selection method) and produces two new admissible task mappings v_1 and v_2. The crossover operator works as follows. A task index k is randomly selected in the interval $[1, n-1]$, and it achieves $v_1[i] = V_1[i]$ and $v_2[i] = V_2[i]$ $\forall i$ such that $1 \leq i \leq k$, and $v_1[i] = V_2[i]$ and $v_2[i] = V_1[i]$ $\forall i$ such that $k < i \leq n$. While both v_1 and v_2 are not admissible, a new task index k is randomly selected in $[1, n-1]$ and the crossover operation is repeated. If no admissible v_1 (resp. v_2) is produced when all the possible task indexes have been considered, it achieves $v_1 \leftarrow V_1$ (resp. $v_2 \leftarrow V_2$). The two resulting admissible task mapping v_1 and v_2 are inserted in the next population, and this reproduction step is repeated until this next population reaches the appropriate size (50). This step ultimately results in a next population of admissible task mappings which are different from those in the current population. Generally, the average rate of the task mappings in the next population is better than those of the current population, since the best-rated task mappings of the current population are more likely to be selected for the crossover operation (thanks to the roulette wheel method).

Termination step. The selection and reproduction steps are repeated while the accumulated rate of all the task mappings in the next population is lower than those in the current population. In other words, populations are bred while the population rate decreases. Finally, the task mapping that this heuristic returns is the best-rated one found during the whole process.

Important Notes. During the whole process, all the generated populations are composed of only *admissible* task mappings. However some implementations of GA also deal with non-admissible solutions, because passing through the non-admissible solution space sometimes allows to escape from local minima, or to find "short cuts" to some better-rated admissible solutions. In our study, we know that the best task mapping V_{opt} (in regard with the energy consumption) is $V[i] = \text{lp} \; \forall i$, which is often not admissible (otherwise, it could be found by FFDD). By allowing populations to also contain non-admissible task mappings, we have noted that GA prematurely leaves the admissible solution space, by generating populations in such a way that their task mappings too quickly converge toward V_{opt}. Moreover, since our GA is numerously invoked in our experiments in Sect. 4, all its parameters (i.e. the population size, etc.) are set to relatively low values in order to limit the time needed by our simulations. For the same reason, we did not implement the "mutation" operator which is usually employed by the genetic algorithms (see [42] for details about this operator). Indeed, since only admissible task mappings are allowed in the generated populations, the mutation operator should ensure that the resulting task mapping will be still admissible. For some populations, it may be necessary to consider numerous task mappings before finding one which guarantees this property. Implementing this operator therefore involves a consequent increase of the simulation time, while providing a negligible impact on the results.

Simulated Annealing (SA). It is out of the scope of this paper to describe how a simulated annealing works, and we will only present our parameters. The interested reader may consult [43] for details about this algorithm. In our study, the considered solutions of our SA are all the possible admissible task mappings for the given system τ and the function to be minimized is the function $E^{S,S'}(\tau^{\text{lp}}, \tau^{\text{hp}})$, where S and S' are the global scheduling algorithms used by L_{lp} and L_{hp}, respectively. Here is the list of the SA parameters that we used in our experiments:

The initial admissible task mapping. During the initialization step, task mappings are randomly generated until finding an admissible one. If no admissible task mapping is found after 1000 tentatives, the algorithm stops and returns the empty task mapping $\{\phi, \phi\}$.

The neighborhood of a task mapping V is the set of every *admissible* task mapping V' such that $\exists j \in \{1, 2, \ldots, n\}$ for which $V'[j] \neq V[j]$ and $V'[i] = V[i]$ $\forall i \neq j$. In other words, an admissible task mapping V' is a neighbor of V if it differs from V by only one task assignment.

The temperature T is handled as follows: the initial temperature is set to 1 and at each iteration multiple of 100, the temperature is decreased such that $T \leftarrow 0.95 \times T$.

The acceptance probabilistic function $\text{Prob}(\Delta E, T)$ is defined as follows. ΔE is the *difference between the rate of the current task mapping* V_{cur} *and the rate of its selected neighbor* V_{neighbor}, i.e. $\Delta E \overset{\text{def}}{=} E^{S,S'}(\tau^{\text{lp}}_{\text{neighbor}}, \tau^{\text{hp}}_{\text{neighbor}}) - E^{S,S'}(\tau^{\text{lp}}_{\text{cur}}, \tau^{\text{hp}}_{\text{cur}})$ and $\text{Prob}(\Delta E, T) = e^{\frac{-\Delta E}{T}}$. Notice that the probability of acceptance progressively decreases with the temperature.

The termination condition : the algorithm stops after 1000 iterations or if all the task mappings in the neighborhood of the current one have been refused. The algorithm then returns the best-rated task mapping that it found during the whole process.

Notice that, for the same reasons as for our genetic algorithm, the parameters of our simulated annealing are chosen relatively low, and only admissible solutions are considered.

4 Simulation Results

4.1 Conditions of the Simulation

Since our methodology does not consider DVFS-capable processors, we do not compare the consumption of our proposed strategy with the consumption while using DVFS algorithms. Thereby, the goal of our experiments is *to quantify the average energy savings provided by a* DMP *in regard with the consumption of an homogeneous multiprocessors platform* $\mathcal{P}_{\text{hom}}$ *composed of only* hp-*processors*. To justify the use of only hp-processors in $\mathcal{P}_{\text{hom}}$, we assume that all the real-time systems generated in our simulations have at least one task which is too heavy to be executed upon a lp-processor.

The DMP that we consider in our experiments (noted π) is composed of 10 processors: 5 lp-processors and 5 hp-processors. The lp- and hp-processor cores are the Diamond 108 Mini and the Diamond 570T, respectively. The 108 Mini is an ultra-low power CPU with minimal gate count for lowest silicon cost whereas the core 570T is a extremely high-performance, 2- or 3-issue static superscalar processor [44]. We assume that the lp-processor cores run at 0.6V, are manufactured with a 130nm technology and are built with ARM's Artisan Metro low-power cell libraries. Due to the process technology, the physical cell libraries, and the synthesis options selected, their clock speed are constrained to be well below 100MHz [45] and we assume that their clock speed is of 50MHz in our simulations. On the other hand, we assume that the cores 570T run at

Table 1. Description of the processor cores

CPU characteristics	lp-**proc.**	hp-**proc.**
Operating clock freq. (MHz)	50	250
Static pow. diss. (mW)	0.21	1.41
Dynamic pow. diss. (mW/MHz)	0.04	0.41
Idle/Busy factor	8	8
Speed (MIPS/MHz)	1.34	1.59

1.2V, are manufactured with a 130nm technology and are built with ARM's Artisan SageX cell libraries. These processor cores can typically run at clock rates in excess of 200MHz [45] and we assume in our experiments that they run at 250MHz. Table 1 shows the characteristics of these processors from [45]. The idle/busy factor denotes the consumption ratio between a busy processor and an idle processor. Unfortunately, this factor is not given for the processors considered in this work, but we assume that the factor is comprised between 4 and 11, based on other hardware [46]. We thus choose to set it to 8 in our experiments. Notice that Tab. 1 provides the required values for our processor model: the lp-processors can be modeled as ($e^{\text{lp}}_{\text{busy}} = 0.21 + 50 \times 0.04 = 2.21$, $e^{\text{lp}}_{\text{idle}} = e^{\text{lp}}_{\text{busy}}/8 \approx 0.28$) and the hp-processor as ($e^{\text{hp}}_{\text{busy}} = 103.91$, $e^{\text{hp}}_{\text{idle}} \approx 12.99$). The last line of Tab. 1 gives the speed of the processors, depending on their operating clock frequency. For a given task worst-case execution requirement expressed in number of instructions, the processors speeds allow to determine the worst-case execution time of the tasks upon both the lp- and hp-processors (denoted C_i^{lp} and C_i^{hp} in our task model).

Concerning the scheduling algorithms used by π, our simulations were first carried out while using Global-DM (see Fig. 2) on both L_{lp} and L_{hp} and on $\mathcal{P}_{\text{hom}}$. $\mathcal{P}_{\text{hom}}$ is composed of only hp-processors (the same hp-processors than in π), but its number of processors is not statically fixed. For each generated real-time system τ, the number of hp-processors in $\mathcal{P}_{\text{hom}}$ is set to $\text{CPU}^{\mathsf{DM}}_{\text{hp}}(\tau)$ before computing its energy consumption while executing τ. We achieved this in order to compare our DMP π with an homogeneous platform optimized in its number of available resources. In our second experiment (see Fig. 3), Global-EDF is used by L_{lp}, L_{hp} and $\mathcal{P}_{\text{hom}}$, and the number of processors of $\mathcal{P}_{\text{hom}}$ is therefore set to $\text{CPU}^{\mathsf{EDF}}_{\text{hp}}(\tau)$. Other experiments are still possible (e.g. by considering that different and distinct scheduling algorithms are used by L_{lp}, L_{hp} and $\mathcal{P}_{\text{hom}}$), but we have focused our study on these two experiments due to the space limitation.

To study the energy savings provided by π over $\mathcal{P}_{\text{hom}}$ in the real-time context, we generate a large amount of sporadic constrained-deadline real-time systems. For each generated system τ, a task mapping $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$ is produced by jointly using FFDD, GA and SA algorithms. Then, the energy consumption of π while executing $\{\tau^{\text{lp}}, \tau^{\text{hp}}\}$ is compared to the consumption of $\mathcal{P}_{\text{hom}}$ while executing τ. The consumption of π is determined thanks to Expression (5) and the consumption of $\mathcal{P}_{\text{hom}}$ by Expression (4).

In our simulations, we studied how the tasks characteristics (i.e. the tasks density) affect the energy saved by the use of π *over* $\mathcal{P}_{\text{hom}}$. We distinguish between four groups of real-time tasks: the *HP tasks* ($\delta_i^{\text{lp}} > 1$ and $\delta_i^{\text{hp}} \leq 1$), the *Heavy LP tasks* ($1 \geq \delta_i^{\text{lp}} > 0.65$), the *Middle LP tasks* ($0.65 \geq \delta_i^{\text{lp}} \geq 0.3$) and the *Light LP tasks* ($0.3 > \delta_j^{\text{lp}} > 0$). During our simulations, a real-time system τ of 20 tasks is generated as follows. The first task is always generated in the HP group (in order to justify the fact that $\mathcal{P}_{\text{hom}}$ is composed of only hp-processors). Then, the remaining task are generated from two different selected groups, where the number of tasks belonging to both these groups is given. Consequently, we have six different mixed set of tasks: HP and HLP, HP and MLP, HP and LLP, HLP and MLP, HLP and LLP, and MLP and LLP. Hence, we can then study the variation in the energy savings provided by π when tasks are moved from one group to another, while limiting the number of possible transfer of tasks between the groups.

4.2 Results of Our Simulation

Figures 2 and 3 show our results when the scheduling algorithms Global-DM and Global-EDF (respectively) are used on both platforms L_{lp} and L_{hp} and by $\mathcal{P}_{\text{hom}}$. The X-axis represents all the 6 possible couples of different selected tasks groups. For each one of the six points in the X-axis, the Y-axis represents the number of tasks in the two selected groups. For instance, for $x =$ "MLP and LLP", $y = 14$ means that $(n - y - 1) = 5$ tasks belong to the Middle LP group, $y = 14$ tasks belong to the Light LP group and 1 task is an HP task. That is, $5 + 14 + 1 = 20$ tasks. The Z-axis represents the energy savings. For each couple of (x, y) discrete values, 100 real-time systems of n tasks are generated in regard with these values x and y. The simulator computes the average consumption gain (and the standard deviation) between the execution of these 100 systems upon π (while determining an energy-optimized task mapping for each system) and upon the platform $\mathcal{P}_{\text{hom}}$ (while determining a sufficient number of hp-processors for each system). The upper plane is the sum between the average energy savings and the standard deviation, and the lower plane is the difference between the average energy savings and the standard deviation.

Notice that in both Figs. 2 and 3, there is no energy savings for the greater part of real-systems containing many HP tasks. Actually, most of the real-time

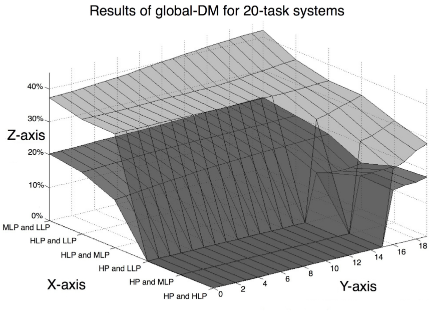


Fig. 2. Results for DM

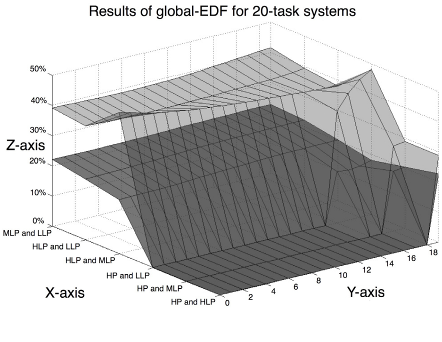


Fig. 3. Results for EDF

systems generated in this space region have a so high workload that the considered heuristics do not find any *admissible* task mapping for the 10 processors of π. Since not enough information is obtained about the energy savings to provide significant statistics, these systems are not taken into account and we set the energy savings of the heuristics to zero. On the other hand, we see that the average energy savings out of this region mainly vary between 20% and 40% for both global-DM and global-EDF.

4.3 Some Statistical Results

Tables 2 and 3 present some statistical numbers, provided by our simulation. Table 2 shows the relative number of cases where the corresponding heuristic is not worst than the others. Notice that the use of a sophisticated heuristic such that GA or SA is clearly urged by these results. Table 3 shows the relative number of cases where a schedulability test for EDF provides a lower *or equal* number of required processors than the others.

5 Conclusion and Future Work

In this paper, we have addressed the energy-aware scheduling problem upon a DMP, a dual CPU type multiprocessor architecture composed of two homogeneous platforms. We exhibited the multiple advantages of such devices from software and practical point of views, and we showed that our DMP architecture, in addition to be compatible with any general-purpose processor, *considerably simplifies the energy-aware scheduling problem to a 2-bins packing problem.* Moreover on the contrary to the most of the existing DVFS solutions, our methodology takes into account the static consumption of the processors and is therefore more appropriate to the emerging processors technologies. Finally, we showed in our experiments that our methodology, by taking as starting point some popular optimization heuristics, provides a relevant energy savings upon a DMP architecture, compared to the energy consumption of an homogeneous multiprocessor platform.

In our future work, we will allow job migrations between the two homogeneous platforms L_{lp} and L_{hp} in order to address a new *slack reclaiming scheme* (see [47] for a definition). This *on-line* mechanism deals with the dynamic workload of the task instances, while the system is running. It profits from the earlier job completions of the lp-tasks by filling the *slack time* with some pending hp-tasks.

Table 2. Heuristics comparison

	DM	**EDF**
RA	0.01 %	0.04 %
FFDD	0 %	0 %
GA	91.65 %	89.32%
SA	8.39 %	10.68 %

Table 3. Tests comparison

	EDF
GFB	99.98 %
BAK	0.0 %
BB	48.21 %

It therefore reduces the workload on the platform L_{hp}, and hence the total energy consumption.

References

1. Baruah, S., Anderson, J.: Energy-aware implementation of hard-real-time systems upon multiprocessor platform. In: Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems, pp. 430–435 (August 2003)
2. Jerraya, A., Wolf, W.: Multiprocessor Systems-on-Chips. Morgan Kaufmann, Elsevier (2005)
3. Veeravalli, B., Goh, L., Viswanathan, S.: Design of fast and efficient energy-aware gradient-based scheduling algorithms heterogeneous embedded multiprocessor systems. IEEE Transactions on Parallel and Distributed Systems 99(1) (2008)
4. Gorjiara, B., Bagherzadeh, N., Chou, P.: An efficient voltage scaling algorithm for complex SoCs with few number of voltage modes. In: Proceedings of 2004 International Symposium on Low Power Electronics and Design, pp. 381–386 (2004)
5. Gorjiara, B., Bagherzadeh, N., Chou, P.: Ultra-fast and efficient algorithm for energy optimization by gradient-based stochastic voltage and task scheduling. ACM Transactions on Design Automation of Electronic Systems 12(4) (September 2007)
6. Gorjiara, B., Chou, P., Bagherzadeh, N., Reshadi, M., Jensen, D.: Fast and efficient voltage scheduling by evolutionary slack distribution. In: Proceedings of Asia and South Pacific Design Automation Conference, pp. 659–662 (January 2004)
7. Gruian, F., Kuchcinski, K.: Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In: Proceedings of Asia and South Pacific Design Automation Conference, pp. 449–455 (January 2001)
8. Liu, Y., Veeravalli, B., Viswanathan, S.: Novel critical-path based low-energy scheduling algorithms for heterogeneous multiprocessor real-time embedded systems. In: 13th International Conference on Parallel and Distributed Systems, pp. 1–8 (2007)
9. Luo, J., Jha, N.K.: Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In: Proceedings of International Conference on Computer-Aided Design, pp. 357–364 (November 2000)
10. Rae, A., Parameswaran, S.: Voltage reduction of application-specific heterogeneous multiprocessor systems for power minimisation. In: Proceedings of Design Automation Conference, pp. 147–152 (2000)
11. Yu, Y., Prasanna, V.K.: Resource allocation for independent real-time tasks in heterogeneous systems for energy minimization. Journal of Information Science and Engineering 19(3) (May 2003)
12. Andrei, A., Schmitz, M.T., Eles, P., Peng, Z., Al-Hashimi, B.M.: Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In: IEEE Proceedings - Computers and Digital Techniques, pp. 28–38 (2005)
13. Zhang, Y., Hu, X., Chen, D.Z.: Task scheduling and voltage selection for energy minimization. In: Proceedings of 39th Design Automation Conference, pp. 183–188 (June 2002)
14. Schmitz, M.T., Al-Hashimi, B.: Considering power variations of DVS processing elements for energy minimisation in distributed systems. In: Proceedings of International Symposium on Systems Synthesis, pp. 250–255 (October 2001)

15. Leung, L.F., Tsui, C.Y., Ki, W.H.: Minimizing energy consumption of multiple-processors-core systems with simultaneous task allocation, scheduling and voltage assignment. In: Proceedings of Asia and South Pacific Design Automation Conference, pp. 647–652 (2004)
16. Schmitz, M., Al-Hashimi, B., Eles, P.: Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. In: Proceedings of Design, Automation and Test in Europe Conference and Exhibition, pp. 514–521 (2002)
17. Oyama, Y., Ishihara, T., Sato, T., Yasuura, H.: A multi-performance processor for low power embedded applications. In: Proc. of COOL Chips X IEEE Symposium on Low-Power and High-Speed Chips, p. 138 (2007)
18. Ekekwe, N., Etienne-Cummings, R.: Power dissipation sources and possible control techniques in ultra deep submicron cmos technologies. Microelectronics Journal 37(9), 851–860 (2006)
19. Taur, Y., Nowark, E.: CMOS devices below 0.1 um: how high will performance go? In: Electron Devices Meeting, Technical Digest, pp. 215–218. International Publication (1997)
20. Mukhopadhyay, S., Roy, K., Mahmoodi-Meimand, H.: Leakage current mechanisms and leakage reduction techniques in deep-submicron CMOS circuits. Proceedings of the IEEE 91(2), 305–327 (2003)
21. Leroy, A.: Optimizing the on-chip communication architecture of low power Systems-on-Chip in Deep Sub-Micron technology. PhD thesis, Université Libre de Bruxelles (2006)
22. Annavaram, M., Grochowski, E., Shen, J.: Mitigating Amdahl's law through EPI throttling. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pp. 298–309 (2005)
23. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 81–92 (2003)
24. Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I.: Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, pp. 64–75 (2004)
25. Benini, L., Micheli, G.D.: Networks on Chips: A New SoC Paradigm. Computer 35(1), 70–78 (2002)
26. Towles, B., Dally, W.J.: Route packets, not wires: On-chip interconnection networks. Design Automation Conference 0, 684–689 (2001)
27. Wolkotte, P.T., Smit, G.J.M., Kavaldjiev, N., Becker, J.E., Becker, J.: Energy model of networks-on-chip and a bus. In: 2005 International Symposium on System-on-Chip, 2005. Proceedings, pp. 82–85 (2005)
28. Bolotin, E., Cidon, I., Ginosar, R., Kolodny, A.: Cost considerations in network on chip. Integr. VLSI J. 38(1), 19–42 (2004)
29. Baruah, S., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th real-time systems symposium, Orlando, Florida, pp. 182–190 (1990)
30. Baruah, S., Baker, T.: Schedulability analysis of global EDF. Real-Time Systems 38(3), 223–235 (2008)
31. Ripoll, I., Crespo, A., Mok, A.K.: Improvement in feasibility testing for real-time tasks. Real-Time Systems 11(1), 19–39 (1996)

32. Baker, T., Fisher, N., Baruah, S.: Algorithms for determining the load of a sporadic task system. Technical Report TR-051201, Department of Computer Science, Florida State University (2005)
33. Fisher, N., Baker, T., Baruah, S.: Algorithms for determining the demand-based load of a sporadic task system. In: Proceedings of the 12th International Conference on Embedded and Real-Time Computing, pp. 135–146 (2006)
34. Fisher, N., Baruah, S., Baker, T.P.: The partitioned scheduling of sporadic tasks according to static-priorities. In: Euromicro Conference on Real-Time Systems, vol. 0, pp. 118–127 (2006)
35. Baker, T.: Multiprocessor EDF and deadline monotonic schedulability analysis. In: Proceedings of the 24th IEEE International Real-Time Systems Symposium, pp. 120–129 (2003)
36. Baruah, S., Cohen, N., Plaxton, C., Varvel, D.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15(6), 600–625 (1996)
37. Dertouzos, M., Mok, A.: Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Transactions on Software Engineering 15(2), 1497–1506 (1989)
38. Cho, H., Ravindran, B., Jensen, E.D.: An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium, pp. 101–110 (2006)
39. Real, J., Crespo, A.: Mode change protocols for real-time systems: A survey and a new proposal. Real-Time Systems 26(2), 161–197 (2004)
40. Nélis, V., Goossens, J.: Mode change protocol for multi-mode real-time systems upon identical multiprocessors. Technical Report arXiv:0809.5238v1, Cornell University (September 2008)
41. Bertogna, M., Cirinei, M., Lipari, G.: Improved schedulability analysis of EDF on multiprocessor platforms. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems, pp. 209–218 (2005)
42. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning, vol. 1. Addison-Wesley Professional, Reading (1989)
43. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
44. D&R Industry Articles: Diamond standard processor core family architecture. Tensilica White Paper (July 2007)
45. Halfhill, T.R.: Tensilica's preconfigured cores: Six embedded-processor cores challenge ARM, ARC, MIPS, and DSPs. Microprocessor Report (2006)
46. Gavrichenkov, I.: Meet intel wolfdale: Core 2 duo e8500, e8400 and e8200 processors review (2008)
47. Aydin, R., Melhem, R., Mossé, D., Mejia-Alvarez, P.: Power-aware scheduling for periodic real-time tasks. IEEE Transactions on Computers 53(5), 584–600 (2004)

Revising Distributed UNITY Programs Is NP-Complete*

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, U.S.A.
{borzoo,sandeep}@cse.msu.edu

Abstract. We focus on automated revision techniques for adding UNITY properties to distributed programs. We show that unlike centralized programs, where multiple safety properties along with one progress property can be simultaneously added in polynomial-time, addition of only one safety or one progress property to distributed programs is NP-complete. We also propose an efficient symbolic heuristic for adding a leads-to property to a distributed program. We demonstrate the application of this heuristic in automated synthesis of recovery paths in fault-tolerant distributed programs.

Keywords: UNITY, Distributed programs, Automated revision, Transformation, Repair, Complexity, Formal methods.

1 Introduction

Program correctness is an important aspect and application of formal methods. There are two ways to achieve correctness when designing programs: *correct-by-verification* and *correct-by-construction*. Applying the former often involves a cycle of design, verification, and subsequently manual repair if the verification step does not succeed. The latter, however, achieves correctness in an automated fashion.

Taking the paradigm of correct-by-construction to extreme leads us to synthesizing programs from their specification. While synthesis from specification is undoubtedly useful, it suffers from lack of *reuse*. In *program revision*, on the other hand, one can transform an input program into an output program that meets additional properties. As a matter of fact, such properties are frequently identified during a system's life cycle in practice due to reasons such as incomplete specification, renovation of specification, and change of environment. As a concrete example, consider the case where a program is diagnosed with a failed property by a model checker. In such a case, access to automated transformation methods that revise the program at hand with respect to the failed property is

* This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 408–427, 2008.

highly advantageous. For such revision to be useful, in addition to satisfaction of new properties, the output program must inevitably preserve existing properties of the input program as well.

In our previous work in this context [8], we focused on revising *centralized programs*, where processes can read and write all program variables in one atomic step, with respect to Unity [7] properties. Our interest in Unity properties is due to the fact that they have been found highly expressive in specifying a large class of programs. In [8], we showed that adding a conjunction of multiple Unity *safety* properties (i.e., unless, stable, and invariant) along with one *progress* property (i.e., leads-to and ensures) can be achieved in polynomial-time. We also showed that the problem becomes NP-complete if we consider simultaneous addition of two progress properties. We emphasize that our revision method in [8] ensures satisfaction of all existing Unity properties of the input program as well.

In this paper, we shift our focus to *distributed programs* where processes can read and write only a subset of program variables. We expect the concept of program revision to play a more crucial role in the context of distributed programs, since non-determinism and race conditions make it significantly difficult to assert program correctness. We find somewhat unexpected results about the complexity of adding Unity properties to distributed programs. In particular, we find that the problem of adding only one Unity safety property or one progress property to a distributed program is NP-complete in the size of the input program's state space.

The knowledge of these complexity bounds is especially important in building tools for incremental synthesis. In particular, the NP-completeness results demonstrate that tools for revising distributed programs must utilize efficient heuristics to expedite the revision algorithm at the cost of *completeness*. Moreover, NP-completeness proofs often identify where the exponential complexity lies in the problem. Thus, thorough analysis of proofs is also crucial in devising efficient heuristics.

With this motivation, in this paper, we also propose an efficient symbolic heuristic that adds a leads-to property to a distributed program. We integrate this heuristic with our tool Sycraft [5] that is designed for adding fault-tolerance to existing distributed programs. Meeting leads-to properties are of special interest in fault-tolerant computing where *recovery* within a finite number of steps is essential. To this end, one can first augment the program with all possible recovery transitions that it can use. This augmented program clearly does not guarantee that it would recover to a set of legitimate states, although there is a potential to reach the legitimate states from states reached in the presence of faults. In particular, it may continue to execute on a cycle that is entirely outside the legitimate states. Thus, we apply our heuristic for adding a leads-to property to modify the augmented program so that from any state reachable in the presence of faults, the program is guaranteed recovery to its legitimate states within a finite number of steps. A by-product of the heuristic for adding leads-to properties is a cycle resolution algorithm. Our experimental results show that this algorithm can also be integrated with state-of-the-art model checkers for assisting in developing programs that are correct-by-construction.

Organization. The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. Then, we formally state the revision problem in Section 3. Section 4 is dedicated to complexity analysis of addition of UNITY safety properties to distributed programs. In Section 5, we present our results on the complexity of addition of UNITY progress properties. We also present our symbolic heuristic and experimental results in Section 5. Related work is discussed in Section 6. Finally, we conclude in Section 7.

2 Preliminary Concepts

In this section, we formally define the notion of distributed programs. We also reiterate the concept of UNITY properties introduced by Chandy and Misra [7].

2.1 Distributed Programs

Intuitively, we define a distributed program in terms of a set of processes. Each process is in turn specified by a state-transition system and is constrained by some read/write restrictions over its set of variables.

Let $V = \{v_0, v_1 \cdots v_n\}$ be a finite set of variables with finite domains $D_0, D_1 \cdots D_n$, respectively. A *state*, say s, is determined by mapping each variable v_i in V, $0 \leq i \leq n$, to a value in D_i. We denote the value of a variable v in state s by $v(s)$. The set of all possible states obtained by variables in V is called the *state space* and is denoted by $\mathcal{S}$. A *transition* is a pair of states of the form (s_0, s_1) where $s_0, s_1 \in \mathcal{S}$.

Definition 1 (state predicate). Let $\mathcal{S}$ be the state space obtained from variables in V. A *state predicate* is a subset of $\mathcal{S}$. ∎

Definition 2 (transition predicate). Let $\mathcal{S}$ be the state space obtained from variables in V. A *transition predicate* is a subset of $\mathcal{S} \times \mathcal{S}$. ∎

Definition 3 (process). A process p is specified by the tuple $\langle V_p, T_p, R_p, W_p \rangle$ where V_p is a set of variables, T_p is a transition predicate in the state space of p (denoted $\mathcal{S}_p$), R_p is a set of variables that p can read, and W_p is a set of variables that p can write such that $W_p \subseteq R_p \subseteq V_p$ (i.e., we assume that p cannot blindly write a variable). ∎

Write restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process. Clearly, T_p must be disjoint from the following transition predicate due to inability of p to change the value of variables that p cannot write:

$$NW_p = \{(s_0, s_1) \mid v(s_0) \neq v(s_1) \text{ where } v \notin W_p\}.$$

Read restrictions. Let $p = \langle V_p, T_p, R_p, W_p \rangle$ be a process, v be a variable in V_p, and $(s_0, s_1) \in T_p$ where $s_0 \neq s_1$. If v is not in R_p, then p must include a corresponding transition from all states s'_0 where s'_0 and s_0 differ only in the value of v. Let (s_0', s_1') be one such transition. Now, it must be the case that s_1 and s_1' are identical except for the value of v, and, the value of v must be the same in s_0' and s_1'. For instance, let $V_p = \{a, b\}$ and $R_p = \{a\}$. Since p cannot read b, the transition $([a = 0, b = 0], [a = 1, b = 0])$ and the transition $([a = 0, b = 1], [a = 1, b = 1])$ have the same effect as far as p is concerned. Thus, each transition (s_0, s_1) in T_p is associated with the following *group predicate*:

$$\begin{aligned} Group_p(s_0, s_1) = \{(s_0', s_1') \mid \\ &(\forall v \notin R_p \; : \; (v(s_0) = v(s_1) \;\; \wedge \;\; v(s_0') = v(s_1'))) \wedge \\ &(\forall v \in R_p \; : \; (v(s_0) = v(s_0') \;\; \wedge \;\; v(s_1) = v(s_1')))\}. \end{aligned}$$

Definition 4 (distributed program). A *distributed program* Π is specified by the tuple $\langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ where $\mathcal{P}_\Pi$ is a set of processes and $\mathcal{I}_\Pi$ is a set of initial states. Without loss of generality, we assume that the state space of all processes in $\mathcal{P}_\Pi$ is identical (i.e., $\forall p, q \in \mathcal{P}_\Pi :: (V_p = V_q) \wedge (D_p = D_q)$). Thus, the set of variables (denoted V_Π) and state space of program Π (denoted $\mathcal{S}_\Pi$) are identical to the set of variables and state space of processes of Π, respectively. In this sense, the set $\mathcal{I}_\Pi$ of initial states of Π is a subset of $\mathcal{S}_\Pi$. ∎

Notation. Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a distributed program (or simply a program). The set $\mathcal{T}_\Pi$ denotes the collection of transition predicates of all processes of Π, i.e., $\mathcal{T}_\Pi = \bigcup_{p \in \mathcal{P}_\Pi} T_p$.

Definition 5 (computation). Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a program. An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ is a *computation* of Π iff the following three conditions are satisfied: (1) $s_0 \in \mathcal{I}_\Pi$, (2) $\forall i \geq 0 : (s_i, s_{i+1}) \in \mathcal{T}_\Pi$, and (3) if $\overline{s}$ reaches a *terminating* state s_l where there does not exist s such that $s \neq s_l$ and $(s_l, s) \in \mathcal{T}_\Pi$, then we extend $\overline{s}$ to an infinite computation by stuttering at s_l using transition (s_l, s_l). ∎

Notice that we distinguish between a terminating computation and a *deadlocked* computation. Precisely, if a computation $\overline{s}$ reaches a terminating state s_d such that there exists no process p in $\mathcal{P}_\Pi$ where $(s_d, s) \in T_p$ for some state s, then s_d is a *deadlock state* and $\overline{s}$ is a *deadlocked computation.* For a distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, we say that a sequence of states $\overline{s} = \langle s_0, s_1 \cdots s_n \rangle$ is a *computation prefix* of Π iff $\forall j \mid 0 \leq j < n : (s_j, s_{j+1}) \in \mathcal{T}_\Pi$.

2.2 UNITY Properties

UNITY properties are categorized by two classes of *safety* and *progress* properties defined next [7].

Definition 6 (UNITY safety properties). Let P and Q be arbitrary state predicates.

- (Unless) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P unless Q' iff $\forall i \geq 0 : (s_i \in (P \cap \neg Q)) \Rightarrow (s_{i+1} \in (P \cup Q))$. Intuitively, if P holds in a

state of $\overline{s}$, then either (1) Q never holds in $\overline{s}$ and P is continuously true, or (2) Q becomes true and P holds at least until Q becomes true.
- (Stable) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'stable P' iff $\overline{s}$ satisfies P unless *false*. Intuitively, P is stable iff once it becomes true, it remains true forever.
- (Invariant) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'invariant P' iff $s_0 \in P$ and $\overline{s}$ satisfies stable P. An invariant property always holds. ∎

Definition 7 (UNITY progress properties). Let P and Q be arbitrary state predicates.

- (Leads-to) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P leads-to Q' iff $(\forall i \geq 0 : (s_i \in P) \Rightarrow (\exists j \geq i : s_j \in Q))$. In other words, if P holds in a state s_i, $i \geq 0$, of $\overline{s}$, then there exists a state s_j in $\overline{s}$, $i \leq j$, such that Q holds in s_j.
- (Ensures) An infinite sequence of states $\overline{s} = \langle s_0, s_1 \cdots \rangle$ satisfies 'P ensures Q' iff for all i, $i \geq 0$, if $P \cap \neg Q$ is true in state s_i, then (1) $s_{i+1} \in (P \cup Q)$, and (2) $\exists j \geq i : s_j \in Q$. In other words, if P becomes true in s_i, there exists a state s_j where Q eventually becomes true and P remains true everywhere in between s_i and s_j. ∎

In our formal framework, unlike standard UNITY in which *interleaved fairness* is assumed, we assume that all program computations are unfair. This assumption is necessary when dealing with addition of UNITY progress properties to programs. We also note that the definition of ensures property is slightly different from that in [7]. Precisely, in Chandy and Misra's definition, P ensures Q implies that (1) P leads-to Q, (2) P unless Q, and (3) there is at least one action that always establishes Q whenever it is executed in a state where P is true and Q is false. Since, we do not model actions explicitly in our work, we have removed the third requirement. Finally, as described in Subsection 2.1, in this paper, our focus is only on programs with *finite* state space.

We now define what it means for a program to refine a UNITY property. Note that throughout this paper, we assume that a program and its properties have identical state space.

Definition 8 (refines). Let $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ be a program and $\mathcal{L}$ be a UNITY property. We say that Π refines $\mathcal{L}$ iff all computations of Π are infinite and satisfy $\mathcal{L}$. ∎

Definition 9 (specification). A UNITY *specification* Σ is the conjunction $\bigwedge_{i=1}^{n} \mathcal{L}_i$ where each $\mathcal{L}_i$ is a UNITY safety or progress property. ∎

One can easily extend the notion of refinement to UNITY specifications as follows. Given a program Π and a specification $\Sigma = \bigwedge_{i=1}^{n} \mathcal{L}_i$, we say that Π refines Σ iff for all i, $1 \leq i \leq n$, Π refines $\mathcal{L}_i$.

Concise representation of safety properties. Observe that the UNITY safety properties can be characterized in terms of a set of *bad transitions* that should never occur in a program computation. For example, $\mathsf{stable}\ P$ requires that a transition, say (s_0, s_1), where $s_0 \in P$ and $s_1 \notin P$, should never occur in any computation of a program that refines $\mathsf{stable}\ P$. Hence, for simplicity, in this paper, when dealing with safety Unity properties of a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, we assume that they are represented by a transition predicate $\mathcal{B} \subseteq \mathcal{S}_\Pi \times \mathcal{S}_\Pi$ whose transitions should never occur in any computation.

3 Problem Statement

Given are a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a (new) Unity specification Σ_n. Our goal is to devise an automated method which revises Π so that the revised program (denoted $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$) (1) refines Σ_n, and (2) continues refining its existing Unity specification Σ_e, where Σ_e is unknown. Thus, during the revision, we only want to reuse the correctness of Π with respect to Σ_e in the sense that the correctness of Π' with respect to Σ_e is derived from 'Π refines Σ_e'.

Intuitively, in order to ensure that the revised program Π' continues refining the existing specification Σ_e, we constrain the revision problem so that the set of computations of Π' is a subset of the set of computations of Π. In this sense, since Unity properties are not existentially quantified (unlike in Ctl), we are guaranteed that all computations of Π' satisfy the Unity properties that participate in Σ_e.

Now, we formally identify constraints on $\mathcal{S}_{\Pi'}$, $\mathcal{I}_{\Pi'}$, and $\mathcal{T}_{\Pi'}$. Observe that if $\mathcal{S}_{\Pi'}$ contains states that are not in $\mathcal{S}_\Pi$, there is no guarantee that the correctness of Π with respect to Σ_e can be reused to ensure that Π' refines Σ_e. Also, since $\mathcal{S}_\Pi$ denotes the set of all states (not just reachable states) of Π, removing states from $\mathcal{S}_\Pi$ is not advantageous. Likewise, $\mathcal{I}_{\Pi'}$ should not have any states that were not there in $\mathcal{I}_\Pi$. Moreover, since $\mathcal{I}_\Pi$ denotes the set of all initial states of Π, we should preserve them during the revision. Finally, we require that $\mathcal{T}_{\Pi'}$ should be a subset of $\mathcal{T}_\Pi$. Note that not all transitions of $\mathcal{T}_\Pi$ may be preserved in $\mathcal{T}_{\Pi'}$. Hence, we must ensure that Π' does not deadlock. Based on Definitions 8 and 9, if (i) $\mathcal{T}_{\Pi'} \subseteq \mathcal{T}_\Pi$, (ii) Π' does not deadlock, and (iii) Π refines Σ_e, then Π' also refines Σ_e. Thus, the *revision problem* is formally defined as follows:

Problem Statement 1 Given a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a Unity specification Σ_n, identify $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that:

$(C1)$ $\mathcal{S}_{\Pi'} = \mathcal{S}_\Pi$,
$(C2)$ $\mathcal{I}_{\Pi'} = \mathcal{I}_\Pi$,
$(C3)$ $\mathcal{T}_{\Pi'} \subseteq \mathcal{T}_\Pi$, and
$(C4)$ Π' refines Σ_n. ∎

Note that the requirement of deadlock freedom is not explicitly specified in the above problem statement, as it follows from 'Π' refines Σ_n'. Throughout the paper, we use '*revision* of Π with respect to a specification Σ_n (or property $\mathcal{L}$)' and '*addition* of Σ_n (respectively, $\mathcal{L}$) to Π' interchangeably.

4 Adding UNITY Safety Properties to Distributed Programs

As mentioned in Section 2, Unity safety properties can be characterized by a transition predicate, say $\mathcal{B}$, whose transitions should occur in no computation of a program. In a centralized setting where processes have no restrictions on reading and writing variables, a program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ can be easily revised with respect to $\mathcal{B}$ by simply (1) removing the transitions in $\mathcal{B}$ from $\mathcal{T}_\Pi$, and (2) making newly created deadlock states unreachable [8].

To the contrary, the above approach is not adequate for a distributed setting, as it is *sound* (i.e., it constructs a correct program), but not *complete* (i.e., it may fail to find a solution while there exists one). This is due to the issue of read restrictions in distributed programs, which associates each transition of a process with a group predicate. This notion of grouping makes the revision complex, as a revision algorithm has to examine many combinations to determine which group of transitions must be removed and, hence, what deadlock states need to be handled. Indeed, we show that the issue of read restrictions changes the class of complexity of the revision problem entirely.

Instance. A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a Unity safety specification Σ_n.

Decision problem. Does there exist a program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that Π' meets the constraints of Problem Statement 1 for the above instance?

We now show that the above decision problem is NP-complete by a reduction from the well-known *satisfiability* problem. The SAT problem is as follows:

> Let $x_1, x_2 \cdots x_N$ be propositional *variables*. Given a Boolean formula $y = y_{N+1} \wedge y_{N+2} \cdots y_{M+N}$, where each *clause* y_j, $N+1 \leq j \leq M+N$, is a disjunction of three or more literals, does there exist an assignment of truth values to $x_1, x_2 \cdots x_N$ such that y is satisfiable?

We note that the unconventional subscripting of clauses in the above definition of the SAT problem is deliberately chosen to make our proofs simpler.

Theorem 1. *The problem of adding a* Unity *safety property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only need to show that the problem is NP-hard. Towards this end, we present a polynomial-time mapping from an instance of the SAT problem to a corresponding instance of our revision problem. We construct the instance $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ as follows.

Variables. The set of variables of program Π and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{-1, 0, 1\}$, $\{-1, 0, 1\}$, $\{0, 1\}$, $\{0, 1\}$, $\{-N \cdots -2, -1, 1, 2 \cdots M+N\} \cup \{j^i \mid (1 \leq$

$i \leq N) \wedge (N+1 \leq j \leq M+N)\}$. We note that j^i in the last set is not an exponent, but a denotational symbol.

Reachable states. The set of reachable states in our mapping is as follows:

- For each propositional variable x_i, $1 \leq i \leq N$, in the instance of the SAT problem, we introduce the following states (see Figure 1): $a_i, b_i, b'_i, c_i, c'_i, d_i$, and d'_i. We require that states a_1 and a_{N+1} are identical.
- For each clause y_j, $N+1 \leq j \leq M+N$, we introduce state r_j.
- For each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i in clause y_j, $1 \leq i \leq N$, we introduce the following states: $r_{ji}, s_{ji}, s'_{ji}, t_{ji}$, and t'_{ji}.

Value assignments. Assignment of values to each variable at reachable states is shown in Figure 1 (denoted by $< v_0, v_1, v_2, v_3, v_4 >$). We emphasize that assignment of values in our mapping is the most crucial factor in forming group predicates. For reader's convenience, Table 1 illustrates the assignment of values to variables more clearly.

Table 1. Assignment of values to variables in proof of Theorem 1

(a)

State / Variable name	v_0	v_1	v_2	v_3	v_4
a_i	-1	1	0	1	i
b_i	0	0	0	0	$-i$
b'_i	0	0	0	0	i
c_i	1	0	1	1	$-i$
c'_i	0	1	1	1	i
d_i	0	1	1	1	$-i$
d'_i	1	0	1	1	i

(b)

State / Variable name	v_0	v_1	v_2	v_3	v_4
r_j	0	0	1	0	j
r_{ji}	0	0	0	0	j^i
s_{ji}	0	1	1	1	j^i
s'_{ji}	1	0	1	1	j^i
t_{ji}	1	-1	0	1	j^i
t'_{ji}	-1	-1	0	1	j^i

Processes. Program Π consists of four processes. Formally, $\mathcal{P}_\Pi = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\mathcal{P}_\Pi$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_2, v_3\}$ and $W_{p_1} = \{v_0, v_2, v_3\}$.
  - $R_{p_2} = \{v_1, v_2, v_3\}$ and $W_{p_2} = \{v_1, v_2, v_3\}$.
  - $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_1, v_2, v_4\}$.
  - $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_0, v_1, v_3, v_4\}$.
- **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 1):
  - $T_{p_1} = \{(b'_i, d'_i), (b_i, c_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b'_i, c'_i), (b_i, d_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_3} = \{(c'_i, a_{i+1}), (c_i, a_{i+1}), (d'_i, a_{i+1}), (d_i, a_{i+1}) \mid 1 \leq i \leq N\}$.
  - $T_{p_4} = \{(a_i, b_i), (a_i, b'_i) \mid 1 \leq i \leq N\}$.

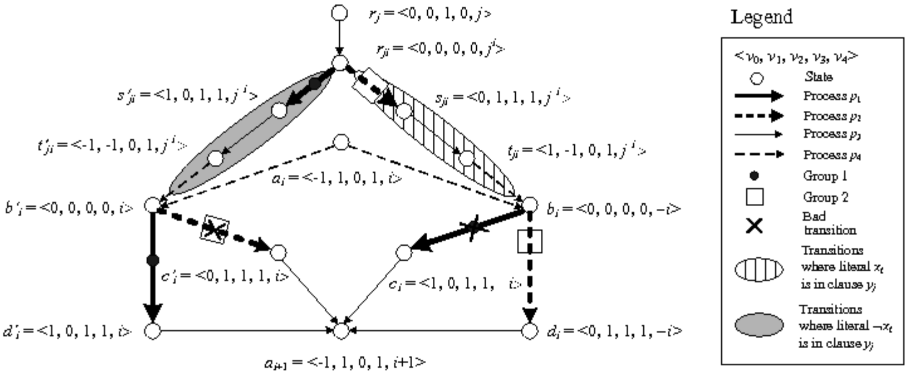


Fig. 1. Mapping SAT to addition of UNITY safety properties

Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_3} and the following:

- If x_i is a literal in clause y_j, then we include transition (r_{ji}, s_{ji}) in T_{p_2}, (s_{ji}, t_{ji}) in T_{p_3}, and (t_{ji}, b_i) in T_{p_4}.
- If $\neg x_i$ is a literal in clause y_j, then we include transition (r_{ji}, s'_{ji}) in T_{p_1}, (s'_{ji}, t'_{ji}) in T_{p_3}, and (t'_{ji}, b'_i) in T_{p_4}.

Note that only for the sake of illustration, Figure 1 shows all possible transitions. However, in order to construct Π, based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of the transitions.

Initial states. The set $\mathcal{I}_\Pi$ of initial states represents clauses of the instance of the SAT problem, i.e., $\mathcal{I}_\Pi = \{r_j \mid N+1 \leq j \leq M+N\}$.

Safety property. Let P be a state predicate that contains all reachable states in Figure 1 except c_i and c'_i (i.e., $c_i, c'_i \in \neg P$). Thus, the properties stable P and invariant P can be characterized by the transition predicate $\mathcal{B} = \{(b_i, c_i), (b'_i, c'_i) \mid 1 \leq i \leq N\}$. Similarly, let P and Q be two state predicates that contain all reachable states in Figure 1 except c_i and c'_i. Thus, the safety property P unless Q can be characterized by $\mathcal{B}$ as well. In our mapping, we let $\mathcal{B}$ represent the safety specification for which Π has to be revised.

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s'_{ji}), (b'_i, d'_i)$, and (b_i, c_i) are grouped in p_1.
2. Due to inability of process p_2 to read variable v_4, for all i, $1 \leq i \leq N$, transitions $(r_{ji}, s_{ji}), (b_i, d_i)$, and (b'_i, c'_i) are grouped in p_2.
3. Transitions grouped with the rest of the transitions in Figure 1 are unreachable and, hence, are irrelevant.

Now, we show that the answer to the SAT problem is affirmative if and only if there exists a solution to the revision problem. Thus, we distinguish two cases:

- ($\Rightarrow$) First, we show that if the given instance of the SAT formula is satisfiable, then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program Π', that is obtained by adding the safety property represented by $\mathcal{B}$ to program Π as follows.
  - The state space of Π' consists of all the states of Π, i.e., $\mathcal{S}_\Pi = \mathcal{S}_{\Pi'}$.
  - The initial states of Π' consists of all the initial states of Π, i.e., $\mathcal{I}_\Pi = \mathcal{I}_{\Pi'}$.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *true*, then we include the following transitions: (a_i, b_i) in T_{p_4}, (b_i, d_i) in T_{p_2}, and (d_i, a_{i+1}) in T_{p_3}.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *false*, then we include the following transitions:(a_i, b'_i) in T_{p_4}, (b'_i, d'_i) in T_{p_1}, and (d'_i, a_{i+1}) in T_{p_3}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include the following transitions: (r_j, r_{ji}) and (s_{ji}, t_{ji}) in T_{p_3}, (r_{ji}, s_{ji}) in T_{p_2}, and (t_{ji}, b_i) in T_{p_4}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include the following transitions: (r_j, r_{ji}) and (s'_{ji}, t'_{ji}) in T_{p_3}, (r_{ji}, s'_{ji}) in T_{p_1}, and (t'_{ji}, b'_i) in T_{p_4}.

  As an illustration, we show the partial structure of Π', for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$, in Figure 2. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that Π' meets the requirements of the Problem Statement 1:

  1. The first three constraints of the decision problem are trivially satisfied by construction.
  2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if Π refines UNITY specification Σ_e, then Π' refines Σ_e as well. Moreover, if a computation of Π' reaches a state b_i for some i, from an initial state r_j (i.e., x_i is *true* in clause y_j), then that computation cannot violate safety since bad transition (b_i, c_i) is removed. This is due to the fact that (b_i, c_i) is grouped with transition (r_{ji}, s'_{ji}) and this transition is not included in $\mathcal{T}_{\Pi'}$, as literal x_i is *true* in y_j. Likewise, if a computation of Π' reaches a state b'_i for some i, from initial state r_j (i.e., x_i is *false* in clause y_j), then that computation cannot violate safety since transition (b'_i, c'_i) is removed. This is due to the fact that (b'_i, c'_i) is grouped with transition (r_{ji}, s_{ji}) and this transition is not included in $\mathcal{T}_{\Pi'}$, as x_i is *false*. Thus, Π' refines Σ_n.

- ($\Leftarrow$) Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let Π' be the program that is obtained by adding the safety property Σ_n to program Π. Now, in order to obtain a

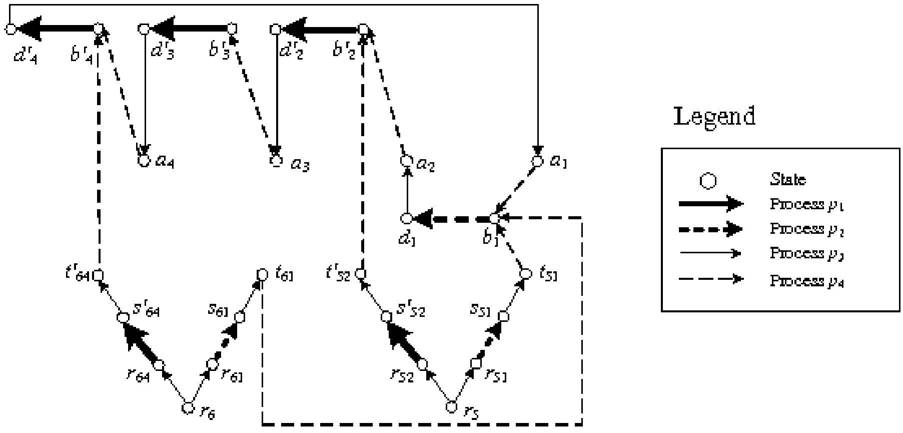


Fig. 2. The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$

solution for SAT, we proceed as follows. If there exists a computation of Π' where state b_i is reachable, then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \leq j \leq M+N$, and let r_j be the corresponding initial state in $\mathcal{I}_{\Pi'}$. Since r_j is an initial state and Π' cannot deadlock, the transition (r_j, r_{ji}) must be present in $\mathcal{T}_{\Pi'}$, for some i, $1 \leq i \leq N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}. Observe that $\mathcal{T}_{\Pi'}$ cannot have both transitions, as grouping of transitions will include both (b_i, c_i) and (b'_i, c'_i) which in turn causes violation of safety by Π'. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji}, then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5 Adding UNITY Progress Properties to Distributed Programs

This section is organized as follows. In Subsection 5.1, we show that adding a UNITY progress property to a distributed program is NP-complete. Then, in Subsection 5.2, we present a symbolic heuristic for adding a leads-to property to a distributed program.

5.1 Complexity

In a centralized setting, where programs have no restriction on reading and writing variables, a program, say $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$, can be easily revised with respect to a progress property by simply (1) breaking non-progress cycles that prevent a program to eventually reach a desirable state predicate, and (2) removing deadlock states [8]. To the contrary, in a distributed setting, due to the issue of grouping, it matters which transition (and as a result its corresponding group) is removed to break a non-progress cycle.

Instance. A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a UNITY progress property Σ_n.

Decision problem. Does there exist a program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ such that Π' meets the constraints of Problem Statement 1 for the above instance?

Theorem 2. *The problem of adding a* UNITY *progress property to a distributed program is NP-complete.*

Proof. Since showing membership to NP is straightforward, we only show that the problem is NP-hard by a reduction from the SAT problem. First, we present a polynomial-time mapping.

Variables. The set of variables of program Π and, hence, its processes is $V = \{v_0, v_1, v_2, v_3, v_4\}$. The domain of these variables are respectively as follows: $\{0,1\}$, $\{0,1\}$, $\{-N \cdots -2, -1, 1, 2 \cdots M+N\} \cup \{j^i \mid (1 \leq i \leq N) \wedge (N+1 \leq j \leq M+N)\}$, $\{-1,0,1\}$, and $\{-1,0,1\}$.

Reachable states. The set of reachable states in our mapping is as follows:

- For each propositional variable x_i, $1 \leq i \leq N$, we introduce the following states (see Figure 3): a_i, a_i', b_i, b_i', c_i, c_i', d_i, d_i', Q_i, and Q_i'.
- For each clause y_j, $N+1 \leq j \leq M+N$, we introduce state r_j.
- For each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we introduce states r_{ji}, s_{ji}, and s_{ji}'.

Value assignments. Assignment of values to each variable at reachable states is shown in Figure 3 (denoted by $< v_0, v_1, v_2, v_3, v_4 >$). For reader's convenience, Table 2 illustrates the assignment of values to variables more clearly.

Processes. Program Π consists of four processes. Formally, $\mathcal{P}_\Pi = \{p_1, p_2, p_3, p_4\}$. Transition predicate and read/write restrictions of processes in $\mathcal{P}_\Pi$ are as follows:

- **Read/write restrictions.** The read/write restrictions of processes p_1, p_2, p_3, and p_4 are as follows:
  - $R_{p_1} = \{v_0, v_1, v_3\}$ and $W_{p_1} = \{v_0, v_1, v_3\}$.
  - $R_{p_2} = \{v_0, v_1, v_4\}$ and $W_{p_2} = \{v_0, v_1, v_4\}$.
  - $R_{p_3} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_3} = \{v_0, v_2, v_3, v_4\}$.
  - $R_{p_4} = \{v_0, v_1, v_2, v_3, v_4\}$ and $W_{p_4} = \{v_1, v_2, v_3, v_4\}$.

Table 2. Assignment of values to variables in proof of Theorem 2

(a)

State / Variable name	v_0	v_1	v_2	v_3	v_4
a_i	1	0	$-i$	-1	-1
a'_i	1	0	i	-1	1
b_i	0	0	$-i$	0	0
b'_i	0	0	i	0	0
c_i	1	1	$-i$	0	1
c'_i	1	1	i	1	0
d_i	0	1	i	1	-1
d'_i	0	1	$-i$	1	1
Q_i	1	1	$-i$	1	0
Q'_i	1	1	i	0	1

(b)

State / Variable name	v_0	v_1	v_2	v_3	v_4
r_j	0	1	j	1	1
r_{ji}	0	0	j^i	0	0
s_{ji}	1	1	j^i	0	1
s'_{ji}	1	1	j^i	1	0

– **Transition predicates.** For each propositional variable x_i, $1 \leq i \leq N$, we include the following transitions in processes p_1, p_2, p_3, and p_4 (see Figure 3):
  - $T_{p_1} = \{(b'_i, c'_i), (b_i, Q_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_2} = \{(b_i, c_i), (b'_i, Q'_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_3} = \{(a_i, b_i), (a'_i, b'_i), (c_i, d_i), (c'_i, d'_i), (Q_i, Q_i), (Q'_i, Q'_i) \mid 1 \leq i \leq N\}$.
  - $T_{p_4} = \{(d'_i, b_i), (d_i, b'_i) \mid 1 \leq i \leq N\}$.

  Moreover, corresponding to each clause y_j, $N+1 \leq j \leq M+N$, and variable x_i, $1 \leq i \leq N$, in clause y_j, we include transition (r_j, r_{ji}) in T_{p_4} and the following:
  - If x_i is a literal in clause y_j, then we include transition (r_{ji}, s_{ji}) in T_{p_2}, and (s_{ji}, a_i) in T_{p_4}.
  - If $\neg x_i$ is a literal in clause y_j, then we include transition (r_{ji}, s'_{ji}) in T_{p_1} and (s'_{ji}, a'_i) in T_{p_4}.

  Note that for the sake of illustration, Figure 3 shows all possible transitions. However, in order to construct Π', based on the existence of x_i or $\neg x_i$ in y_j, we only include a subset of transitions.

Initial states. The set $\mathcal{I}_\Pi$ of initial states represents clauses of the Boolean formula in the instance of the SAT problem, i.e., $\mathcal{I}_\Pi = \{r_j \mid N+1 \leq j \leq M+N\}$.

Progress property. In our mapping, the desirable progress property is of the form $\Sigma_n \equiv (\mathit{true}$ leads-to $Q)$, where $Q = \{Q_i, Q'_i \mid 1 \leq i \leq N\}$ (see Figure 3). Observe that Σ_n is a leads-to as well as an ensures property. This property in Linear Temporal Logic (LTL) is denoted by $\Box\Diamond Q$ (called *always eventually* Q).

Before we present our reduction from the SAT problem using the above mapping, we make the following observations regarding the grouping of transitions in different processes:

1. Due to inability of process p_1 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s'_{ji}), (b'_i, c'_i), and (b_i, Q_i) are grouped in process p_1.

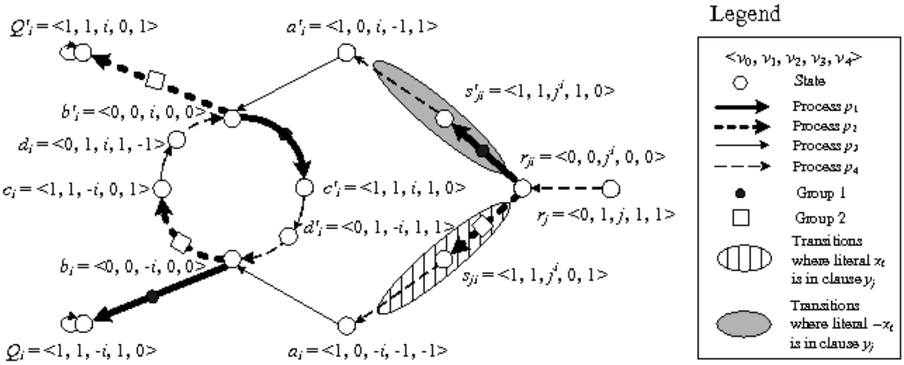


Fig. 3. Mapping SAT to addition of a progress property

2. Due to inability of process p_2 to read variable v_2, for all i, $1 \leq i \leq N$, transitions (r_{ji}, s_{ji}), (b_i, c_i), and (b'_i, Q'_i) are grouped in process p_2.
3. Transitions grouped with the rest of the transitions in Figure 3 are unreachable and, hence, are irrelevant.

We distinguish the following two cases for reducing the SAT problem to our revision problem :

- $(\Rightarrow)$ First, we show that if the given instance of the SAT formula is satisfiable, then there exists a solution that meets the requirements of the revision decision problem. Since the SAT formula is satisfiable, there exists an assignment of truth values to all variables x_i, $1 \leq i \leq N$, such that each y_j, $N+1 \leq j \leq M+N$, is true. Now, we identify a program Π', that is obtained by adding the progress property $\Box\Diamond Q$ to program Π as follows.
  - The state space of Π' consists of all the states of Π, i.e., $\mathcal{S}_\Pi = \mathcal{S}_{\Pi'}$.
  - The initial states of Π' consists of all the initial states of Π, i.e., $\mathcal{I}_\Pi = \mathcal{I}_{\Pi'}$.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *true*, then we include the following transitions: (a_i, b_i), (c_i, d_i), and (Q'_i, Q'_i) in T_{p_3}, (b_i, c_i) and (b'_i, Q'_i) in T_{p_2}, and, (d_i, b'_i) in T_{p_4}.
  - For each variable x_i, $1 \leq i \leq N$, if x_i is *false*, then we include the following transitions: (a'_i, b'_i), (c'_i, d'_i), and (Q_i, Q_i) in T_{p_3}, (b'_i, c'_i) and (b_i, Q_i) in T_{p_1}, and, (d'_i, b_i) in T_{p_4}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal x_i, if x_i is *true*, we include transitions (r_j, r_{ji}) and (s_{ji}, a_i) in T_{p_4}, and, transition (r_{ji}, s_{ji}) in T_{p_2}.
  - For each clause y_j, $N+1 \leq j \leq M+N$, that contains literal $\neg x_i$, if x_i is *false*, we include transitions (r_j, r_{ji}) and (s'_{ji}, a'_i) in T_{p_4}, and, transition (r_{ji}, s'_{ji}) in T_{p_1}.

  As an illustration, we show the partial structure of Π', for the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = $ *true*, $x_2 = $ *false*, $x_3 = $ *false*, and

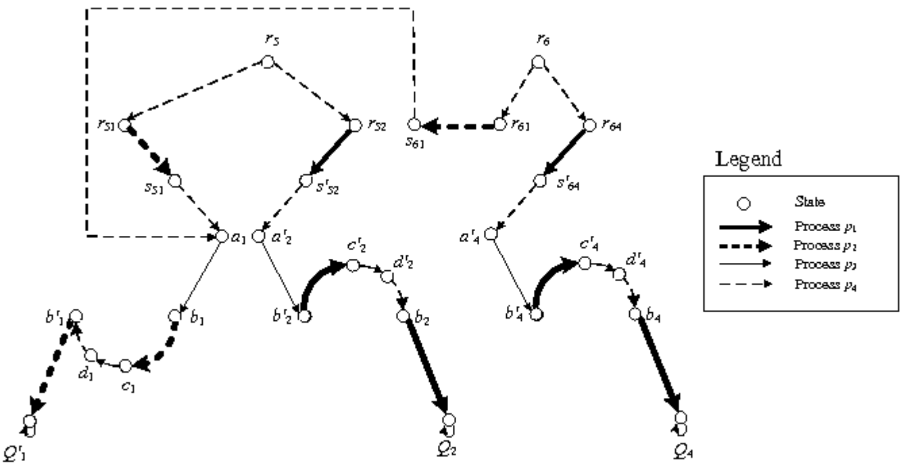


Fig. 4. The structure of the revised program for Boolean formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$, where $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{false}$, and $x_4 = \mathit{false}$

$x_4 = \mathit{false}$ in Figure 4. Notice that states whose all outgoing and incoming transitions are eliminated are not illustrated. Now, we show that Π' meets the requirements of the Problems Statement 1:

1. The first three constraints of the decision problem are trivially satisfied by construction.
2. We now show that constraint $C4$ holds. First, it is easy to observe that by construction, there exist no reachable deadlock states in the revised program. Hence, if Π refines Unity specification Σ_e, then Π' refines Σ_e as well. Moreover, by construction, all computations of Π' eventually reach either Q_i or Q'_i and will stutter there. This is due to the fact that if literal x_i is *true* in clause y_j, then transition (r_{ji}, s'_{ji}) is not included in $\mathcal{T}_{\Pi'}$ and, hence, its group-mates (b'_i, c'_i) and (b_i, Q_i) are not in $\mathcal{T}_{\Pi'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q'_i without meeting a cycle. Likewise, if literal $\neg x_i$ is *false* in clause y_j, then transition (r_{ji}, s_{ji}) is not included in $\mathcal{T}_{\Pi'}$ and, hence, its group-mates (b_i, c_i) and (b'_i, Q'_i) are not in $\mathcal{T}_{\Pi'}$ as well. Consequently, a computation that starts from r_j eventually reaches Q_i without meeting a cycle. Hence, Π' refines $\Sigma_n \equiv \Box\Diamond Q$.

– $(\Leftarrow)$ Next, we show that if there exists a solution to the revision problem for the instance identified by our mapping from the SAT problem, then the given SAT formula is satisfiable. Let Π' be the program that is obtained by adding the progress property in $\Sigma_n \equiv \Box\Diamond Q$ to program Π. Now, in order to obtain a solution for SAT, we proceed as follows. If there exists a computation of Π' where state a_i is reachable, then we assign x_i the truth value *true*. Otherwise, we assign the truth value *false*.

We now show that the above truth assignment satisfies all clauses. Let y_j be a clause for some j, $N+1 \leq j \leq M+N$, and let r_j be the corresponding initial state in $\mathcal{I}_{\Pi'}$. Since r_j is an initial state and Π' cannot deadlock, the transition (r_j, r_{ji}) must be present in $\mathcal{T}_{\Pi'}$, for some i, $1 \leq i \leq N$. By the same argument, there must exist some transition that originates from r_{ji}. This transition terminates in either s_{ji} or s'_{ji}. Observe that $\mathcal{T}_{\Pi'}$ cannot have both transitions, as grouping of transitions will include transitions (b_i, c_i) and (b'_i, c'_i). If this is the case, Π' does not refine the property $\Box\Diamond Q$ due to the existence of cycle $b_i \rightarrow c_i \rightarrow d_i \rightarrow b'_i \rightarrow c'_i \rightarrow d'_i \rightarrow b_i$. Thus, there can be one and only one outgoing transition from r_{ji} in $\mathcal{T}_{\Pi'}$. Now, if the transition from r_{ji} terminates in s_{ji}, then clause y_j contains literal x_i and x_i is assigned the truth value *true*. Hence, y_j evaluates to true. Likewise, if the transition from r_{ji} terminates in s'_{ji}, then clause y_j contains literal $\neg x_i$ and x_i is assigned the truth value *false*. Hence, y_j evaluates to true. Therefore, the assignment of values considered above is a satisfying truth assignment for the given SAT formula. ∎

5.2 A Symbolic Heuristic for Adding Leads-To Properties

We now present a polynomial-time (in the size of the state space) symbolic (BDD[1]-based) heuristic for adding leads-to properties to distributed programs. Leads-to properties have interesting applications in automated addition of recovery for synthesizing fault-tolerant distributed programs.

The NP-hardness reduction presented in the proof of Theorem 2 precisely shows where the complexity of the problem lies in. Indeed, Figure 3 shows that transition (b_i, c_i) which can potentially be removed to break the non-progress cycle $b_i \rightarrow c_i \rightarrow d_i \rightarrow b'_i \rightarrow c'_i \rightarrow d'_i \rightarrow b_i$ is grouped with the critical transition (r_{ji}, s_{ji}) which ensures that state r_{ji} and consequently initial state r_j are not deadlocked. The same argument holds for transitions (b'_i, c'_i) and (r_{ji}, s'_{ji}). Thus, a heuristic that adds a leads-to property to a distributed program needs to address this issue.

Our heuristic works as follows (cf. Figure 5). The Algorithm Add_LeadsTo takes a distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and a property P leads-to Q as input, where P and Q are two arbitrary state predicates in the state space of Π. The algorithm (if successful) returns transition predicate of the derived program $\Pi' = \langle \mathcal{P}_{\Pi'}, \mathcal{I}_{\Pi'} \rangle$ that refines P leads-to Q as output. In order to transform Π to Π', first, the algorithm ranks states that can be reached from P based on the length of their shortest path to Q (Line 2). Then, it attempts to break non-progress cycles (Lines 3-13). To this end, it first computes the set of cycles that are reachable from P (Line 4). This computation can be accomplished using any BDD-based cycle detection algorithm. We apply the Emerson-Lie method [10]. Then, the algorithm removes transitions from $\mathcal{T}_\Pi$ that participate in a cycle

[1] Ordered Binary Decision Diagrams [6] represent Boolean formulae as directed acyclic graphs making testing of functional properties such as satisfiability and equivalence straightforward and extremely efficient.

```
Algorithm 1 Add_LeadsTo
Input: A distributed program $\Pi = \langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$ and property $P$ leads-to $Q$.
Output: If successful, transition predicate $\mathcal{T}_{\Pi'}$ of the new program.

 1: repeat
 2:     Let $Rank[i]$ be the state predicate whose length of shortest path
        to $Q$ is $i$, where $Rank[0] = Q$ and $Rank[\infty]$ = the state predicate
        that is reachable from $P$, but cannot reach $Q$;
 3:     for all $i$ and $j$ do
 4:         $C$ := ComputeCycles($\mathcal{T}_\Pi, P$);
 5:         if $(i \leq j) \wedge (i \neq 0) \wedge (i \neq \infty)$ then
 6:             $tmp := Group(\langle C \wedge Rank[i] \rangle \wedge \langle C \wedge Rank[j] \rangle')$;
 7:             if removal of $tmp$ from $\mathcal{T}_\Pi$ eliminates a state from $Q$ then
 8:                 Make $\langle C \wedge tmp \rangle$ unreachable;
 9:             else
10:                 $\mathcal{T}_\Pi := \mathcal{T}_\Pi - tmp$;
11:             end if
12:         end if
13:     end for
14: until $Rank[\infty] = \{\}$
15: $\mathcal{T}_{\Pi'}$ := EliminateDeadlockStates($P$, $Q$, $\langle \mathcal{P}_\Pi, \mathcal{I}_\Pi \rangle$);
16: return $\mathcal{T}_{\Pi'}$;
```

Fig. 5. A symbolic heuristic for adding a leads-to property to a distributed program

and whose rank of source state is less than or equal to the rank of destination state (Lines 6-10). However, since removal of a transition must take place with its entire group predicate, we do not remove a transition that causes creation of deadlock states in Q. Instead, we make the corresponding cycle unreachable (Line 8). This can be done by simply removing transitions that terminate in a state on the cycle. Thus, if removal of a group of transitions does not create new deadlock states in Q, the algorithm removes them (Line 10). Finally, since removal of transitions may create deadlock states outside Q but reachable from P, we need to eliminate those deadlock states (Line 15). Such elimination can be accomplished using the BDD-based method proposed in [4].

Given $O(n^2)$ complexity of the cycle detection algorithm [10], it is straightforward to observe that the complexity of our heuristic is $O(n^4)$, where n is the size of state space of Π. In order to evaluate the performance of our heuristic, we have implemented the Algorithm Add_LeadsTo in our tool SYCRAFT [5]. This heuristic can be used for adding *recovery* in order to synthesize fault-tolerant distributed programs as follows. Let S be a set of legitimate states (e.g., an invariant predicate) and T be the *fault-span* predicate (i.e., the set of states reachable in the presence of faults). First, we add all possible transitions that start from $T - S$ and end in T. Then, we apply the Algorithm Add_LeadsTo for property $(T - S)$ leads-to S.

Figure 6 illustrates experimental results of our heuristic for adding such recovery. All experiments are run on a PC with a 2.8GHz Intel Xeon processor and

	Space		*Time(s)*		
	reachable states	**memory (KB)**	**cycle detection**	**pruning transitions**	**total**
BA^5	10^4	12	0.5	2.5	3
BA^{10}	10^8	18	5	18	23
BA^{15}	10^{12}	26	47	76	125
BA^{20}	10^{16}	29	522	372	894
BA^{25}	10^{20}	30	3722	1131	4853
TR^5	10^2	6	0.2	0.3	0.5
TR^{10}	10^5	7	13	2	15
TR^{15}	10^7	10	470	10	480
TR^{20}	10^9	33	2743	173	2916
TR^{25}	10^{11}	53	22107	2275	24382

Fig. 6. Experimental results of the symbolic heuristic

1.2GB RAM. The BDD representation of the Boolean formulae has been done using the Glu/CUDD package[2]. Our experiments target addition of recovery to two well-known problems in fault-tolerant distributed computing, namely, the *Byzantine agreement* problem [14] (denote BA^i) and the *token ring* problem [2] (denoted TR^i), where i is the number of processes. Figure 6 shows the size of reachable states in the presence of faults, memory usage, total time spent to add the desirable leads-to property, time spent for cycle detection (i.e., Line 4 in Figure 5), and time spent for breaking cycles by pruning transitions. Given the huge size of reachable states and complexity of structure of programs in our experiments, we find the experimental results quite encouraging. We note that the reason that TR and BA behave differently as their number of processes grow is due to their different structures, existing cycles, and number of reachable states. In particular, the state space of TR is highly reachable and its original program has a cycle that includes all of its legitimate states. This is not the case in BA. We also note that in case of TR, the symbolic heuristic presented in this subsection tend to be slower than the constructive layered approach introduced in [4]. However, the approach in this paper is more general and has a better potential of success than the approach in [4].

6 Related Work

The most relevant work to this paper proposes automated transformation techniques for adding Unity properties to centralized programs [8]. The authors show that addition of multiple Unity safety properties along with a single progress property to a centralized program can be accomplished is polynomial-time. They

[2] Colorado University Decision Diagram Package, available at `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

also show that the problem of simultaneous addition of two leads-to properties to a centralized program is NP-complete. Also in this context, Jobstmann et al. [11] independently show that the problem of repairing a centralized program with respect to two progress properties in NP-complete.

Existing synthesis methods in the literature mostly focus on deriving the synchronization skeleton of a program from its specification (expressed in terms of temporal logic expressions or finite-state automata) [9,15,16,1,3]. Although such synthesis methods may have differences with respect to the input specification language and the program model that they synthesize, the general approach is based on the satisfiability proof of the specification. This makes it difficult to provide *reuse* in the synthesis of programs, i.e., any changes in the specification require the synthesis to be restarted from scratch.

Algorithms for automatic addition of fault-tolerance to distributed programs are studied from different perspectives [12,13,4]. These (enumerative and symbolic) algorithms add fault-tolerance concerns to existing programs in the presence of faults, and guarantee not to add new behaviors to the input program in the absence of faults. Most problems in addition of fault-tolerance to distributed programs are known to be NP-complete.

7 Conclusion and Future Work

In this paper, we concentrated on automated techniques for *revising* finite state distributed programs with respect to UNITY properties. We showed that unlike centralized programs, the revision problem for distributed programs with respect to only one safety or one progress property is NP-complete. Thus, the results in this paper is a theoretical evidence to the belief that designing distributed programs is strictly harder than centralized programs even in the context of finite state systems. Our NP-completeness results also generalize the results in [12,13] in the sense that the revision problems remain NP-complete even if the input program is not subject to faults. We also introduced and implemented a BDD-based heuristic for adding a leads-to property to distributed programs in our tool SYCRAFT [5]. Our experiments show encouraging results paving the path for applying automated techniques for deriving programs that are *correct-by-construction* in practice.

For future work, we plan to generalize the issue of distribution by incorporating communication channels in addition to read/write restriction. We also plan to identify sub-problems where one can devise sound and complete algorithms that add UNITY properties to distributed programs in polynomial-time. We also plan to devise heuristics for adding other types of UNITY properties to distributed programs. Another interesting direction is to study the revision problem where programs are allowed to have a combination of fair and unfair computations. We conjecture that this generalization makes the revision problem more complex.

References

1. Arora, A., Attie, P.C., Emerson, E.A.: Synthesis of fault-tolerant concurrent programs. In: Principles of Distributed Computing (PODC), pp. 173–182 (1998)
2. Arora, A., Kulkarni, S.S.: Component based design of multitolerant systems. IEEE Transactions on Software Engineering 24(1), 63–78 (1998)
3. Attie, P., Emerson, E.A.: Synthesis of concurrent programs for an atomic read/write model of computation. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(2), 187–242 (2001)
4. Bonakdarpour, B., Kulkarni, S.S.: Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In: IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 3–10 (2007)
5. Bonakdarpour, B., Kulkarni, S.S.: SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 167–171. Springer, Heidelberg (2008)
6. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
7. Chandy, K.M., Misra, J.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Boston (1988)
8. Ebnenasir, A., Kulkarni, S.S., Bonakdarpour, B.: Revising UNITY programs: Possibilities and limitations. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 275–290. Springer, Heidelberg (2006)
9. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Science of Computer Programming 2(3), 241–266 (1982)
10. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional model mu-calculus. In: Logic in Computer Science (LICS), pp. 267–278 (1986)
11. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
12. Kulkarni, S.S., Arora, A.: Automating the addition of fault-tolerance. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 82–93. Springer, Heidelberg (2000)
13. Kulkarni, S.S., Ebnenasir, A.: The complexity of adding failsafe fault-tolerance. In: International Conference on Distributed Computing Systems (ICDCS), pp. 337–344 (2002)
14. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 4(3), 382–401 (1982)
15. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 6(1), 68–93 (1984)
16. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Principles of Programming Languages (POPL), pp. 179–190 (1989)

On the Solvability of Anonymous Partial Grids Exploration by Mobile Robots*

R. Baldoni[1], F. Bonnet[2], A. Milani[3], and M. Raynal[2]

[1] Università di Roma "La Sapienza", Roma, Italy
[2] IRISA, Université de Rennes 1, Rennes, France
[3] LADyR, GSyC, Universidad Rey Juan Carlos, Spain

Abstract. Given an arbitrary partial anonymous grid (a finite grid with possibly missing vertices or edges), this paper focuses on the exploration of such a grid by a set of mobile anonymous agents (called robots). Assuming that the robots can move synchronously, but cannot communicate with each other, the aim is to design an algorithm executed by each robot that allows, as many robots as possible (let k be this maximal number), to visit infinitely often all the vertices of the grid, in such a way that no vertex hosts more than one robot at a time, and each edge is traversed by at most one robot at a time.

The paper addresses this problem by considering a central parameter, denoted ρ, that captures the view of each robot. More precisely, it is assumed that each robot sees the part of the grid (and its current occupation by other robots, if any) centered at the vertex it currently occupies and delimited by the radius ρ. Based on such a radius notion, a previous work has investigated the cases $\rho = 0$ and $\rho = +\infty$, and shown that, while there is no solution for $\rho = 0$, $k \leq p - q$ is a necessary and sufficient requirement when $\rho = +\infty$, where p is the number of vertices of the grid, and q a parameter whose value depends on the actual topology of the partial grid. This paper completes our previous results by addressing the more difficult case, namely $\rho = 1$. It shows that $k \leq p - 1$ when $q = 0$, and $k \leq p - q$ otherwise, is a necessary and sufficient requirement for solving the problem. More generally, the paper shows that this case is the borderline from which the considered problem can be solved.

Keywords: Anonymity, Grid exploration, Partial grid, Mobile agent, Mutual exclusion, Robot, Synchronous system.

1 Introduction

Graph exploration by robots. The graph exploration problem consists in making one or several mobile entities visit each vertex of a connected graph. The mobile entities are sometimes called agents or robots (in the following we use the word "robot"). The exploration is perpetual if the robots have to revisit forever each vertex of the graph. Perpetual

* The work has been carried out in the context of the European Network of Excellence "ReSIST" (IST-4-026764-NOE) and was supported in part by Comunidad de Madrid grant S-0505/TIC/0285; Spanish MEC grants TIN2005-09198-C02-01 and TIN2008-06735-C02-01.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 428–445, 2008.

exploration is required when robots have to move to gather continuously evolving information or to look for dynamic resources (resources whose location changes with time). If nodes and edges have unique labels, the exploration is relatively easy to achieve.

The graph exploration problem becomes more challenging when the graph is anonymous (i.e., the vertices, the edges, or both have no label). In such a context, several bounds have been stated. They concern the total duration needed to complete a visit of the nodes (e.g. [4,10]), or the size of the memory of the robot necessary to explore a graph (e.g., it is proved in [6] that a robot needs $O(D \log d)$ bits of local memory in order to explore any graph of diameter D and maximum degree d). Impossibility results for one or more robots with bounded memory (computationally speaking, a robot is then a finite state automaton) to explore all graphs have been stated in [12]. The major part of the results on graph exploration consider that the exploration is made by a single robot. Only very recently, the exploration of a graph by several robots has received attention also from a practical side [8]. This is motivated by research for more efficient graph explorations, fault-tolerance, or the need to overcome impossibilities due to the limited capabilities of a single robot.

The constrained exploration problem. Considering the case where the graph is an anonymous partial grid (the grid is connected but has missing vertices/edges), and where the robots can move synchronously but cannot communicate with each other, the paper considers the following instance of the graph exploration problem, denoted the *Constrained Perpetual Graph Exploration* problem ($CPGE$). This problem consists in designing an algorithm executed by each robot that (1) allows as many robots as possible (let k be this maximal number), (2) to visit infinitely often all the vertices of the grid, in such a way that the following mutual exclusion constraints are always satisfied: no vertex hosts more than one robot at a time, and each edge is traversed by at most one robot at a time. These constraints are intended to abstract the problem of collision that robots may incur when moving within a short distance from each other or the necessity for the robots to access resources in mutual exclusion (This mutual exclusion constraint has been considered in [9] in a robot movement problem in a grid). The same algorithm has to work despite the topology of the grid and has to avoid collisions despite the number of robots located on the grid and their initial position. On the other hand, complete exploration may not be ensured if robots are too many.

Results exposed in the paper rest on three parameters, denoted p, q and ρ. The first parameter p is related to the size of the grid, namely, it is the number of vertices of the partial connected grid. The second parameter q is related to the structure of the partial grid. This parameter is defined from a *mobility tree* (a new notion we have introduced in [2]) that can be associated with each partial grid. So, each pair (p, q) represents a subset of all possible partial grids with p vertices. Finally, the third parameter ρ is not related to the grid, but captures the power of the robots when we consider the part of the grid they can see. More precisely, a robot sees the part of the grid centered at its current position and covered by a radius ρ. From an operational point of view, the radius notion allows the robots that are at most ρ apart one from the other to synchronize their moves without violating the vertex and edge mutual exclusion constraints.

In a previous work, we have investigated the extremal cases $\rho = 0$ and $\rho = +\infty$. The associated results are the following ones:

- Case $\rho = +\infty$ (addressed in [3]). In that case, $k \leq p-q$ is a necessary and sufficient requirement for solving the $CPGE$ problem. Let us observe that $\rho = +\infty$ means that the robots knows the structure of the grid and the current position of the robots on that grid. (The initial anonymity assumption of the vertices and the robots can then be overcome.)
- Case $\rho = 0$ (addressed in [3]). In that case, the $CPGE$ problem cannot be solved (i.e., we have $k = 0$) for any grid such that $p > 1$ (a grid with more than one vertex). This means that, whatever the grid, if the robots cannot benefit from some view of the grid, there is no algorithm run by robots that can solve the $CPGE$ problem.

This paper addresses the case $\rho = 1$. Assuming a grid with more than one vertex, it shows that $k \leq p-1$ when $q = 0$, and $k \leq p-q$ otherwise, is a necessary and sufficient requirement for solving the $CPGE$ problem. Basically, the "only if" direction follows from $k \leq p-q$ bound when $\rho = +\infty$. The difficult part is the "if" direction. To that end, the paper presents an algorithm that, when the previous requirement is satisfied, allows each robot to learn the grid, and to synchronize in order each of them navigate it forever, despite anonymity. As we will see, this is not a trivial task.

Finally, the paper discusses issues related to solvability of the $CPGE$ problem when $1 < \rho < +\infty$. It is important to notice that the previous investigations show that $\rho = 1$ is a critical radius value as it defines the fundamental demarcation line for the solvability of the $CPGE$ problem.

Roadmap. The paper is made up of 6 sections. Section 2 presents related works. Section 3 first presents the computation model, and defines formally the $CPGE$ problem. Then, Section 4 briefly states the previous results (cases $\rho = 0$ and $\rho = +\infty$), while Section 5 addresses the case $\rho = 1$. Finally, Section 6 concludes the paper by piecing together all the results, and discussing the case $1 < \rho < +\infty$.

2 Related Work

On the initial assumptions. As already indicated, graph exploration is the process by which each vertex of a graph is visited by some entity. A great research effort on graph exploration by robots has been done on the minimal assumptions (in terms of robots/network requirements) required to explore a graph (e.g., [7]). Some works focus on robots endowed with a finite persistent storage and direct communication with other robots (e.g., [1]). Some works assume that each robot has the capability to see where other robots are currently placed (e.g., [5]). Some other works study how the knowledge of the map (graph) by the robots reduces the complexity of the exploration (e.g., [11]).

On the type of graph exploration. Graph exploration is mainly divided into *perpetual* graph exploration (e.g., [4]) where the robots have to travel the graph infinitely often, and graph exploration *with stop* (e.g., [7]) where each robot has to eventually stop after having explored the graph. Some papers focus on the exploration of the graph by a single robot (e.g., [1,6]). Cooperative graph exploration by a team of mobile robots has also received attention (e.g., [5,9]). As an example of multi-robot exploration, it is shown in [5] that the minimum number of robots required to solve the exploration with stop of a ring of size n is $O(\log n)$ when the exploration is done by oblivious anonymous robots that move asynchronously.

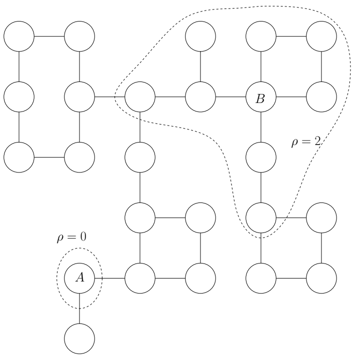
Lower bounds have been established for the perpetual graph exploration problem (e.g., [4]). These bounds concern the period necessary for a robot to complete the visit of the graph, assuming constraints either on the robots, or on the graph. Differently, the upper bound introduced in the Section 3.3 concerns the maximum number of robots that can visit the graph infinitely often without ever colliding.

Constrained graph exploration. The $CPGE$ problem defined in Section 3.2 is the perpetual graph exploration problem augmented with the mutual exclusion property on the vertices and the edges of the graph. These mutual exclusion constraints have been already stated and used [9] where the graphs considered are grids. The problem addressed in that paper is different from $CPGE$. More precisely, in [9], each robot has to visit some target vertices of the grid, and any two distinct robots have different targets. That paper establishes a lower bound on the time (number of rounds in a synchronous system) necessary to solve that problem and presents an optimal algorithm.

The problem of robot collision in also addressed in [14], where is proposed a collision prevention algorithm for robots moving on the plane.

3 Computation Model, Problem Specification and Mobility Tree

3.1 Computation Model



The grid. The underlying grid is made up of a finite set of vertices, each vertex being connected to at least one and at most four other vertices according to the classical grid pattern. If two vertices are connected, we say there is an edge connecting them. The grid is anonymous in the sense no vertex has an identity. Moreover, there is a global sense of direction present in the grid: each vertex is able to distinguish its north, east, south and west neighbors. The grid is represented as graph $G = (S, E)$ with $|S| = p$. An example of a partial grid (with $p = 25$ vertices) is depicted on the left.

The robots. A mobile agent (robot) is an automaton whose computational power is a Turing machine. The moves of a robot are defined by the algorithm it executes. All the robots execute the same algorithm. It is assumed that local computation takes no time.

The finite set of robots R is such that $|R| \leq |S|$. The robots do not have a name: they are anonymous. The robots have a common clock, and the time is divided into synchronous rounds [13]. At each round a robot can move from the vertex where it currently stays to a neighbor vertex, or stays at the same vertex. A move of a robot from a vertex to a neighbor vertex is done in one round.

Notation. *Given a robot a and a round r, $V(a,r)$ denotes the (single) vertex where the robot a is located at the beginning of round r.*

Radius. The radius an algorithm is instantiated with is a non-negative integer ρ that provides each robot with the following information.

Let us first consider the case $\rho \neq 0$. At the beginning of any round r, $\forall a \in R$, the robot a, that is currently located at the vertex $V(a,r)$, sees the sub-grid centered at $V(a,r)$, including the vertices whose distance to $V(a,r)$ is at most ρ. It also sees whether these vertices are currently occupied by robots or not (i.e., for any such vertex v, whether the predicate $\exists\, x \in R :\ V(x,r) = v$ is true or false). An example of radius $\rho = 2$ is depicted in the previous figure: the robot located in the vertex denoted B knows the part of the grid surrounded by the corresponding dotted line (for the vertices at distance $\rho = 2$, it knows only their edges within distance $\rho = 2$).

With a light abuse of the previous notation of radius we consider the following definition for $\rho = 0$: a robot knows the edges of the vertex it is located in. An example is depicted in the previous figure: the robot in the vertex denoted A knows that this vertex has a east edge and a south edge. The fundamental difference with $\rho = 1$ lies in the fact that, when $\rho = 0$, the robot located in A cannot know whether the end vertices associated with these edges are occupied or not by robots.

More generally, the radius notion captures the possibility for robots to synchronize their moves when they are apart from each other at a distance $\leq \rho$.

3.2 The Constrained Perpetual Grid Exploration Problem

The *Constrained Perpetual Grid Exploration* Problem ($CPGE$) can be formally defined by the following three properties.

- Perpetual Exploration.

  $$\forall v \in S :\ \forall a \in R :\ \ \{r \mid V(a,r) = v\} \text{ is not finite.}$$

  (For any vertex v and any robot a, there are infinitely many rounds where a visits v.)
- Vertex Mutual Exclusion.

  $$\forall r \geq 0 :\ \forall (a,b) \in R \times R :\ \ (a \neq b) \Rightarrow \big(V(a,r) \neq V(b,r)\big).$$

  (At the beginning of any round, no two robots are at the same vertex.)
- Edge Mutual Exclusion.

  $$\forall r \geq 0 : \forall (a,b) \in R \times R : (a \neq b) \Rightarrow \big[\big(V(a,r+1) = V(b,r)\big) \Rightarrow \big(V(b,r+1) \neq V(a,r)\big)\big]\,.$$

  (During a round, no two robots move on the same edge, i.e., they cannot exchange their positions).

This paper is on solving the $CPGE$ problem for as many robots as possible. More precisely, we are interested in finding the greatest number of robots and designing an algorithm A (executed by each robot) that solves the $CPGE$ problem whose precise definition appears in Section 3.4.

3.3 Classes of Graphs Defined by the Parameter q

In [2] we prove that arbitrary undirected connected graphs can be classified according to some common topological aspect, which we formalize by the definition of a parameter q. This parameter contributes to state an upper bound on the number of robots beyond which the $CPGE$ problem cannot be solved. As we consider here that the graph on which the robots move is an incomplete grid, the results exposed in [2] are still valid when we replace the word "graph" by the word "grid".

In [2] we introduce the notion of *mobility tree* which is instrumental to extract from a grid the parameter $q \geq 0$ associated with its structure. Each partial grid is univocally associated to a corresponding *mobility tree*, while the same *mobility tree* may be associated to more than one partial grid. For self-containment of the paper, the definitions of both the mobility tree and of the parameter q are recalled in the following.

Preliminary Definitions. A vertex v is a *leaf* of a graph $G = (S, E)$ if there is a single vertex v' such that $(v, v') \in E$. The degree d of a vertex v is the integer $\left|\{v' \mid (v, v') \in E\}\right|$. A *bridge* is an edge whose deletion disconnects the graph. A graph without bridge is a *bridgeless* graph. A path from a vertex v to a vertex v' is *simple* if no vertex appears on it more than once.

A graph $G' = (S', E')$ is a *subgraph* of a graph $G = (S, E)$ if $S' \subseteq S$ and $E' \subseteq E$. In that case, we also say that $G = (S, E)$ is a *supergraph* of $G' = (S', E')$. A *non-singleton* subgraph contains at least two vertices. The subgraph $G' = (S', E')$ is *induced* by the set of vertices S', if E' contains all the edges of E whose end-points are in S'. As, in the following, all the subgraphs we consider are induced subgraphs we omit the term "induced" to not overload the presentation.

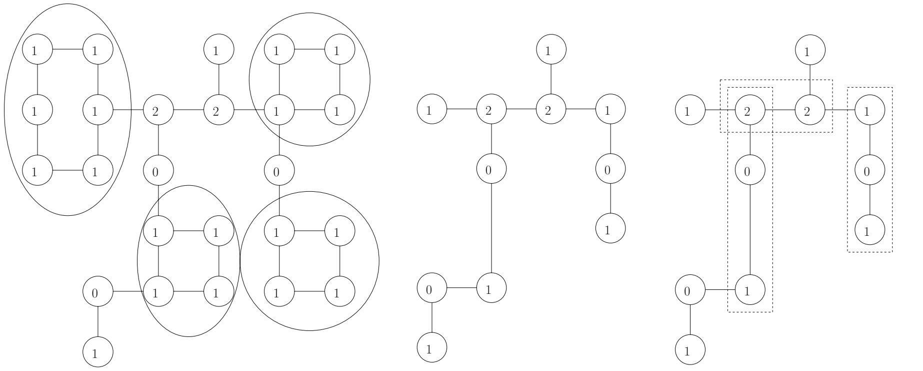
A subgraph G' is *maximal* with respect to a property P if G' satisfies P, while none of its supergraphs satisfies P. So, "bridgeless" and "non-singleton" are properties that a (sub)graph satisfies or does not satisfy.

From a Graph to a Tree: The Reduction Procedure

Definition 1. *(Mobility Tree) Let the* labeled mobility tree *associated with a graph* $G = (S, E)$ *be the labeled tree* $G' = (S', E')$ *derived from* G *through the following reduction procedure:*

1. Initial labeling. *Each vertex* $v \in G$ *is first labeled as follows:*
   - *Label* 0*: if* v *does not belong to a bridgeless subgraph of* G *and its degree is two;*
   - *Label* 1*: if* v *is a leaf of* G *or belongs to a non-singleton bridgeless subgraph of* G*;*
   - *Label* 2*: otherwise.*
2. Compression. *Each maximal non-singleton bridgeless subgraph of* G *is reduced to a vertex with label* 1.

Figure 1 shows an example of the previous reduction procedure. The initial grid G is the grid depicted in page 3. The result of the initial labeling of its vertices is described in Figure 1(a). The non-singleton maximal bridgeless subgraphs of G are



(a) Initial labeling and identification of subgraphs to be compressed

(b) Labeled mobility tree G'

(c) Three mutual exclusion paths

Fig. 1. A graph, its labeled mobility tree, and exclusion paths

surrounded by a circle in that figure. Finally, the resulting labeled mobility tree obtained from the compression of the non-singleton maximal bridgeless subgraphs is shown in Figure 1(b). Figure 1(c) depicts three (out of seven) mutual exclusion paths.

The mobility tree of a graph G is intended to point out the noteworthy features of G as far as the solvability of $CPGE$ is concerned. First, it points out those subgraphs of G (corresponding to vertices with label 1 in the mobility tree) where $CPGE$ could be solved in each of such subgraphs in isolation with a number of robots equal to the number of vertices of the subgraph. These subgraphs are indeed either leafs of G or non-singleton bridgeless subgraphs of G. Second, the mobility tree shows those paths of G that have to be traversed by a single robot at a time to move from one of the previous subgraphs of G to another one in order to extend the solvability of $CPGE$ in G. Let us therefore introduce the notion of Mutual Exclusion Path.

Definition 2. *(Mutual Exclusion Path) Let $\mathcal{P}$ be a path $(v, v^1, v^2 \ldots v^m, v')$ of the mobility tree G' from vertex v to $v^{'}$. $\mathcal{P}$ is a mutual exclusion path of G' iff:*

- *The labels of v and $v^{'}$ are different from 0;*
- *If the path from v to v' contains more than one edge (i.e. if there are vertices v^h, $1 \leq h \leq m$), the intermediate vertices (i.e. the v^h) are labeled* 0.

As an example, Figure 1(c) shows three mutual exclusion paths, $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, of the labeled mobility tree shown in Figure 1(b).

Definition 3. *(Length of a Mutual Exclusion path) In a labeled mobility tree $G' = (S', E')$, let the length of a Mutual Exclusion path between any two vertices v, $v' \in G'$ be the number of edges from v to v' augmented with j, where j is the number of vertices with label 2 in that path.*

The length of $\mathcal{P}_1$ depicted in Figure 1(c) is $2 + j = 3$ (as $j = 1$). The length of $\mathcal{P}_2$ is $1 + j = 3$ (as $j = 2$) while the length of $\mathcal{P}_3$ is $2 + 0 = 2$ (as $j = 0$). Intuitively, *the length of a mutual exclusion path* represents the minimum number of vertices that have to be initially empty (i.e., without robot assignment) in order for the robots to be able to solve the $CPGE$ problem *with respect to that path*. Therefore computing the maximal length of the mutual exclusion paths of a mobility tree associated with a graph G becomes a key factor to compute the upper bound on the number of robots to keep $CPGE$ solvability in G.

Definition 4. *For any $p > 0$ and $q \geq 0$, let $\mathcal{G}(p, q)$ be the set of graphs such that $\forall\ G \in \mathcal{G}(p, q)$: (1) G has p vertices, and (2) q is the maximal length of the mutual exclusion paths of the mobility tree associated with G.*

Two graphs belong to the same class $\mathcal{G}(p, q)$ if they both have the same number of vertices p and the same maximal length q of the mutual exclusion path of their respective mobility trees. The following theorem defines a bound on the number of robots beyond which $CPGE$ cannot be solved.

Theorem 1. [2] *Let G be a graph of the class $\mathcal{G}(p, q)$ and k. There exists no algorithm that solves the $CPGE$ problem for G when there are more than $k = p - q$ robots located on G.*

3.4 f-Solving the $CPGE$ Problem

Let A_ρ denote an algorithm instantiated with the radius value ρ, and a grid G of p vertices. Let y ($0 \leq y \leq p$) denote the number of robots initially placed on the grid under consideration.

Definition 5. *Let f be a function from the set of classes $\mathcal{G}(p, q)$ to the set of non-negative integers, i.e., $f(p, q) \in \{0, 1, \ldots\}$. Given such a function f, an algorithm A_ρ f-*solves *the $CPGE$ problem if A_ρ (1) never violates the vertex and edge mutual exclusion properties (i.e., whatever the value of $y \in \{0 \ldots, p\}$), and (2) solves the $CPGE$ problem for any graph in the class $\mathcal{G}(p, q)$ and any number y of robots such that $0 \leq y \leq f(p, q)$.*

Let us notice that Theorem 1 states that, whatever the value of ρ there is no A_ρ algorithm that f-solves the $CPGE$ problem when there exists a class $\mathcal{G}(p, q)$ such that $f(p, q) > p - q$. As we can see, the algorithms we are interested in always preserve the safety properties (here vertex and edge mutual exclusions) whatever the number y of robots, i.e., even when $y > f(p, q)$.

4 Previous Results ($\rho = 0$ and $\rho = +\infty$)

Theorem 2. [3] *There is an algorithm $A_{+\infty}$ that f-solves the $CPGE$ problem iff for all classes $\mathcal{G}(p, q)$ $f(p, q) \leq p - q$.*

Theorem 3. [3] *There is an algorithm A_0 that f-solves the $CPGE$ problem iff (1) $f(p, q) = 0$ when $(p, q) \neq (1, 0)$, and (2) $f(1, 0) \leq 1$ otherwise.*

5 f-Solvability of the $CPGE$ Problem When $\rho = 1$

That section considers the case where the (vision) radius of each robot is $\rho = 1$. The result of that section is the following main theorem.

Theorem 4. *There is an algorithm A_1 that f-solves the $CPGE$ problem iff $f(1,0) \leq 1$, and $\forall p > 1$ $f(p,0) \leq p-1$, and $\forall q \neq 0$ $f(p,q) \leq p-q$.*

This theorem shows two noteworthy things. First, $\rho = 1$ is the borderline from which the $CPGE$ problem has a non-trivial solution. Second, it shows that, except for $q = 0$, the maximal number of robots for which it can be f-solved is the same as for $\rho = +\infty$, namely, $p - q$.

5.1 Two Simple Lemmas

Lemma 1. *When $p = 1$ (single vertex grid), the $CPGE$ problem can be solved iff there is at most one robot.* (The proof is trivial.).

Lemma 2. *There is no algorithm A_1 that f-solves the $CPGE$ problem if $\exists p > 1$, $f(p,0) > p-1$ or $\exists q \neq 0, f(p,q) > p-q$.*

Proof. For the case $q \neq 0$, the proof follows directly from Theorem 1. Let us now consider the case $q = 0 \wedge p > 1$. Then, $f(p,0) > p-1$ implies that each vertex is occupied by a robot. As $q = 0$, the robots have to move along cycles to f-solve the $CPGE$ problem. Moreover, as $\rho = 1$ there is no possibility for a robot, without moving, to detect a cycle. It follows that there is no possibility of agreement among the robots to move synchronously along a cycle, which proves the lemma. □

So, Lemma 1 proves Theorem 4 for the trivial grid (only one vertex), while Lemma 2 proves the its “only if” direction for the other cases. It remains to prove its difficult part, namely, ”if” part for $p \neq 1$.

5.2 An Algorithm That f-Solves the $CPGE$ Problem

This section presents an algorithm that f-solves the $CPGE$ problem when $p > 1 \wedge f(p,0) \leq p-1$, and when $q \neq 0 \wedge f(p,q) \leq p-q$. The algorithm is based on the following two observations.

- First, in order to let the robots move while avoiding collision, a single vertex without robots (hole) is sufficient if, at any round, all the robots that move, move in the same direction and a robot moves only if its destination vertex is free (this requires $\rho \geq 1$).
- Second, if (1) there is a round where each robot knows the map of the partial grid, (2) there are at most $p - q$ robots and (3) they know where they are located, then the robots can globally synchronize their moves to perpetually explore the grid.

Thus, the whole algorithm consists of the following three steps:

Step 1: Map Building. First, each robot runs an algorithm to (1) know the map and (2) attain a round in which each robot knows that all the robots know the map (see Section 5.3).

Step 2: Evaluate if there are at most $p - q$ robots. Once the robots are in the round in which each robot knows that all the robots know the map, each robot runs an algorithm to learn whether there are more than $p - q$ robots (see Section 5.6).

Step 3: Perpetual Exploration. If the number of robots is $\leq p - q$, each robot supposes there are exactly $p - q$ robots in the system, completing the R real robots by "virtual" robots ($R \leq p - q$ is unknown). Thanks to the repositioning, each robot agrees on the current (real or virtual) robots location. Then, each robot can run the exploration algorithm $\mathsf{explore}_{\infty}()$ described in [3] with $p - q$ robots to ensure the perpetual exploration property.

5.3 Every Robot Learns the Map

In this section we present an algorithm that solves the map building (Step 1, Section 5.2).

Lemma 3. *Consider a partial grid with p vertices and a number of robots $\leq p - 1$, and radius $\rho = 1$. There is an algorithm and an integer k_{max} such that after a finite number*[1] *of rounds, every robot knows (1) the map (i.e., the structure of the grid), (2) the value of k_{max}, and (3) the fact that each other robot knows both the map and k_{max}.*

An algorithm proving the lemma is described in Figure 2. It is based on the following two intuitions: i) each robot can easily build the map corresponding to the sub-grid it visits; ii) if all robots execute the same a-priori known sequence of movements, each robot can deduce additional information on the topology of the grid by looking at the movements of other robots.

Context-sensitive moves. To support the process of learning by observation, we introduce the notion of *context-sensitive move*, which makes a robot to move towards a given direction only if the vertex where it is located has a given neighborhood, that we call its *context*. Formally, the *context* of a vertex v is a subset $C_v \subseteq \{north, west, south, east\}$ such that for each direction in C_v, v has a neighbor in that direction. For example, the cs-move requiring a robot to move east from a vertex with a east and south neighbors, is different from the cs-move requiring it to move east from a vertex with a east, south and west neighbors. There are 32 possible cs-moves: 4 from the vertices with one edge, 12 from the vertices with two edges, 12 from the vertices with three edges, and 4 from the vertex with four edges.

The algorithm $\mathsf{get\_map}_1()$. The algorithm in Figure 2 works as follows: for any $1 \leq k < k_{max}$, all robots incrementally perform all possible sequences of k cs-moves. After executing a given sequence L (the same for all robots), the robots return to their initial position by executing the sequence $\overline{L}$ (L in the reverse order). k_{max} is the minimum value such that for any initial configuration with at most $p - 1$ robots, every robot completely knows the grid after trying all the possible sequences of k_{max} cs-moves.

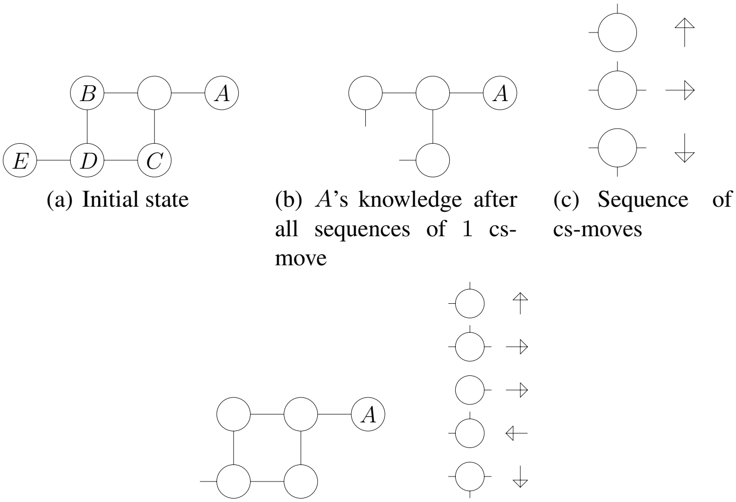
[1] This number is function of k_{max}, it corresponds to the number of rounds needed for robots to explore all the sequences of at most k_{max} cs-moves, namely $2 \sum_{k=1}^{k=k_{max}} 32^k$.

```
operation get_map₁ ():
$k \leftarrow 1$; $k_{max} \leftarrow +\infty$;
while ($k < k_{max}$)
    Execute deterministically all possible sequences composed of $k$ cs-moves while
            computing the corresponding visited graph;
    if (the graph has been entirely deduced)
        then Compute the parameters $p$ and $q$ of the graph; Compute $k_{max}$ end if;
    $k \leftarrow k + 1$
end while
```

Fig. 2. The algorithm get_map$_1$()

5.4 Example of an Execution of the Algorithm get_map$_1$()

We explain here the underlying principles of the algorithm by presenting its behavior in a particular case, the one depicted in Figure 3(a) where 5 robots are located on a grid of 6 vertices. At the initial state, there is a robot A located in a leaf of the grid, and its only neighbor vertex is free. The goal is to prove that after exhausting all possible sequences composed of cs-moves, A is able to deduce all the vertices of the grid.



(a) Initial state

(b) A's knowledge after all sequences of 1 cs-move

(c) Sequence of cs-moves

(d) A's knowledge after all sequences of 3 cs-moves

(e) Sequence of cs-moves

Fig. 3. A simple example

Sequences composed of a single cs-move. Among these sequences there is one that will move the robot A to the west. A then discovers the context of the vertex adjacent to its initial position. It is a vertex where the only missing neighbor is the one at the north. After this, A comes back to its initial position. Among the sequences of length 1, there is also one that will move robot C to its north (and then back) and another that will move robot B to its east neighbor (and then back). After each of those moves, A sees a robot located in its west neighbor vertex. From this observation A can deduce the type of the initial vertex both of robot C and B. This is because all the robots execute the same algorithm, and thus A knows the cs-moves that respectively forced C and B to move to its west neighbor. Figure 3(b) summarizes the knowledge of A after all sequences of a single cs-move.

Sequences composed of two cs-moves. During these sequences, A does not increase its knowledge of the map.

Sequences composed of three cs-moves. A is able to deduce the context of the vertex where D is initially located. Indeed, among all the sequences of three cs-moves, there is the one described in Figure 3(c). The execution of this sequence of cs-moves entails first a move of C to the north, then (during the next round) a move of D to the east, and finally (during the third and last round of the sequence) no move. During the last of those three rounds, only C is located in a vertex that has the context required to move, but it cannot move to the south because the corresponding vertex is occupied. This sequence of 3 cs-moves allows A to learn that a robot D has moved to the initial position of C. Otherwise, C would have moved to the south during the third move. Hence, A deduces the type of D's initial vertex. Figure 3(d) summarizes the knowledge of A after executing the sequences composed of 3 cs-moves.

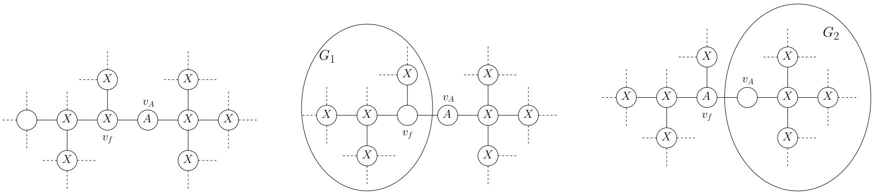
Sequences composed of four cs-moves. A does not increase its knowledge during these sequences of cs-moves.

Sequences composed of five cs-moves. A is able to deduce E's initial vertex when the algorithm will execute the sequence of cs-moves described in Figure 3(e). Indeed, in this sequence of cs-moves, C moves south, but is not able to go back north on the fifth move. This means that the robot E has blocked D which in turn has blocked C. After the moves composed of five cs-moves, A knows entirely the graph.

After discovering the graph. As soon as A knows the graph, it can simulate the behavior of all the robots in the system and consequently know their knowledge. More precisely, (1) A simulates the execution of the algorithm with any initial state of at most $p - 1$ robots. Then (2), A computes the maximum value $k_{\max}$ needed for any robot in any configuration to know entirely the graph. (3) After testing all the sequences of moves composed of k_{max} cs-moves, A knows that all robots knows (i) the map of the partial grid, and also that (ii) each robot has computed the same k_{max}. The robots can then enter the second step of the algorithm.

5.5 The Correctness of the Algorithm get_map$_1$()

We first prove that any robot A learns the map in a finite number of steps. In particular, Lemma 4 proves that, in a grid with only one free vertex, any robot A discovers in a finite number of rounds the whole grid.



(a) Initial configuration (b) Configuration after S_1 (c) Configuration after S_2

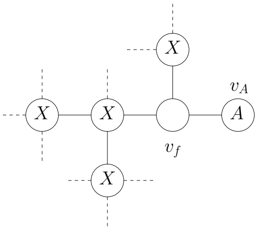
Fig. 4. Example for the proof with $d_f = 3$

Once a robot A has the knowledge of the whole grid G, it can compute the k_{max} needed for all robots to have the same knowledge. Indeed A is able to simulate the behaviors of any robot in any location since they all execute the same algorithm and then A can simulate all the possible executions on G with all possible initial configurations. A takes for k_{max} the maximum k needed for any robot to discover the grid in any initial configuration.

Lemma 4. *Consider a grid G of any class $\mathcal{G}(p, q)$ occupied by exactly $p - 1$ robots. Let A be a robot initially located on a vertex v_A. Let d_f denote the distance between v_A and the unique free vertex of G. After exhausting all the sequences of cs-moves of length $k + d_f$, the robot A knows the subgrid induced by all the vertices that are within distance $\lfloor \frac{k+1}{2} \rfloor$ from v_A.*

Proof. Let call S_1 a sequence of $d_f - 1$ cs-moves that "moves" the free vertex to an adjacent vertex v_f of A and S_2 the same sequence with the additional cs-move that moves A to v_f (v_A becomes then the free vertex). The lemma can be restated in the following way: (1) after exhausting all the sequences of cs-moves of length $k + d_f - 1$, the robot A knows the subgrid G_1 induced by all the vertices belonging to some path $v_A, v_f, \ldots$ that are within distance $\lfloor \frac{k+3}{2} \rfloor$ from v_A and (2) after exhausting all the sequences of cs-moves of length $k + d_f$, the robot A knows the subgrid G_2 induced by all the vertices belonging to some path $v_f, v_A, \ldots$ that are within distance $\lfloor \frac{k+3}{2} \rfloor$ from v_f (and consequently within distance $\lfloor \frac{k+1}{2} \rfloor$ from v_A).

The proofs for the two previous cases are the same considering that before the k cs-moves the sequence S_1 or S_2 is done. Thus, from this point, the proof considers only the first case and for the sake of simplicity we suppose that the free vertex is initially adjacent to A; it corresponds to the case when A is on a leaf whose adjacent vertex is free (see Figure on the right).



The proof is by induction on k. For $k = 0$ (*i.e.* before any cs-move), the robot A knows the grid at distance $\lfloor \frac{0+3}{2} \rfloor = 1$; A knows its own vertex and its adjacent vertex. We assume the property is true for some k, and prove it for $k + 1$. If k is odd then it is trivially true. So, consider k even. By the induction hypothesis, the robot A knows the subgrid induced by all the vertices that are within distance $d = \lfloor \frac{k+3}{2} \rfloor$ from v_A. In the following we prove that with sequences composed of $k + 1$ moves, A will learn the subgrid induced by all vertices that are within distance $d + 1$ (if any).

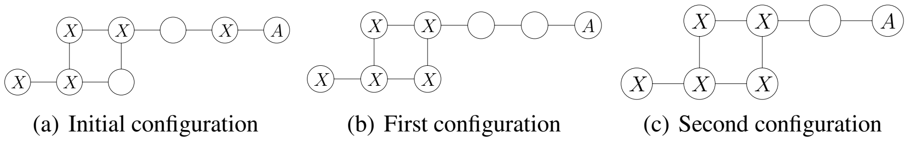
Let us consider a vertex v at distance $d+1$ of A; $p = v, v_1, \ldots, v_d, v_A$ denotes then a minimal path from v to A, whose length is $d+1$ (and where $v_d = v_f$). Let us now consider the sequence of cs-moves that tries to successively move:

1. the robot located on v_{d-1} to v_d, the robot located on v_{d-2} to v_{d-1}, ..., the robot located on v_1 to v_2 (all these $d-1$ cs-moves succeed because at the beginning of round r there is no robot located at vertex v_d. So at each round the robot expected to move, moves because the destination vertex is free),
2. the robot located on v to v_1 (with success, since v_1 become a free vertex thanks to the previous cs-moves),
3. the robot located on v_2 to v_1, the robot located on v_3 to v_2, ..., the robot located on v_d to v_{d-1} (all these $d-1$ cs-moves fail because their destination vertices are always occupied).

After this sequence of cs-moves, the robot A sees a robot on the vertex v_d. Since A knows the last k cs-moves executed, it can deduce the existence of vertex v. In particular, the presence of a robot on the vertex v_d implies that the last cs-move was not successful, which implies that the $d-1$ last cs-moves were not successful, which implies that the cs-move described in item 2 succeeded (the robot located on v to v_1) and then A deduces the existence (and the type) of the vertex v. The length of the sequence of cs-moves that allow A to deduce the existence of v is $(d-1)+1+(d-1) = k+1$. If the vertex v is not there, A will not observe a robot on the vertex v_d after this particular sequence. ([3] shows how to generalize the proof for the case where there are more than one empty vertex in the grid.) □

The clues of the generalization for more free vertices is now given. If there are multiple free vertices as shown in Figure 5(a), A's deduction is more complicated. Indeed in some situation A cannot easily learn the type of some vertices: Contrary to the initial proof, A is not able to deduce the type of the second free vertex since no robot will block other robots. However if the system reaches the configuration drawn in Figure 5(b), A will be able to deduce all vertices of the graph since it is "exactly" as if A were in the configuration of Figure 5(c).

Thus the number of rounds required for A to know/deduce the grid increases, but it remains always possible in any configuration.



(a) Initial configuration (b) First configuration (c) Second configuration

Fig. 5. Example with multiple free vertices

5.6 Evaluating the Predicate $\mathcal{P}_1 \equiv$ "Are There at Most $p - q$ Robots?"

Once each robot knows that all the robots know the map of the partial grid, they can compute the value of the parameters p and q. Then, the robots have to evaluate the predicate $\mathcal{P}_1 \equiv$"are there at most $p - q$ robots?".

To this end, each robot executes the following sequence of operations, where d is the diameter of the grid and R is the actual number of robots:

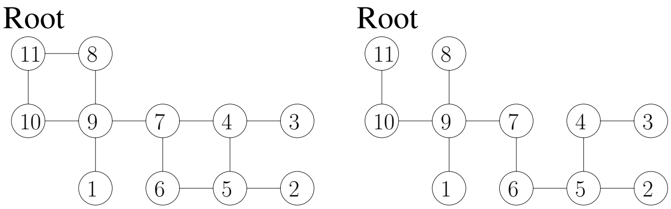
```
operation evaluating_$\mathcal{P}_1$ ():
(01) Deterministically assign an id to each vertex and define a tree;
(02) Move the robots to vertices whose ids goes from 1 to $R$ (in $p \times 4d$ rounds);
(03) Evaluate the predicate $\mathcal{P}_1$.
(04) Move the robots to vertices whose ids goes from 1 to $R$ (in $p \times 4d$ rounds);
```

evaluating_$\mathcal{P}_1$*: Line 01.* Due to the global sense of direction (as defined in Section 3.1), the robots can agree on a same predetermined vertex of the partial grid (e.g., the most "north-west" vertex of the grid). Starting from that vertex, each robot can assign an identifier to each vertex using a *Depth First Search* (DFS) algorithm where the identifiers are assigned according to the *Post-Ordering* rule. (Note that this labeling, from 1 to p, is done locally without any move.)

Due to the fact that the labeling is produced by a post-ordering DFS algorithm, it satisfies the following properties:

Property 1. If all vertices labeled from 1 to ℓ (with $\ell < p$) are suppressed from the grid, the remaining grid (made up of the vertices labeled $\ell + 1$ to p) remains connected. (An example is depicted in Figure 6.)

Property 2. There is a tree rooted at p such that each vertex has for father his adjacent vertex with the smallest id among the id greater than itself. (For example, the vertex labeled 4 with adjacent vertices 3, 5 and 7 has the vertex 5 for father.)



(a) Post Ordering (b) Corresponding Tree

Fig. 6. Example of labeling from a post-ordering DFS

evaluating_$\mathcal{P}_1$*: Line 02.* During this step, the robots move to reach the vertices with the lowest R ids (here "move" refers to simple moves, not cs-moves). Initially, each robot moves in order to try to reach the vertex 1. Each robot computes the shortest path from its current position to vertex 1 and then moves along this path.

In the following, we explain how each robot follows its path without colliding with the other robots. Each direction is associated with a value in the set $\{0, 1, 2, 3\}$. Let s be the direction where a robot has to move according to the next step of its path. Then this robot moves to its s neighbor at round r if (i) $(r \mod 4) = i$ where i is the value associated to the direction s and (ii) the s neighbor vertex is free.

As a result, after at most $4d$ rounds, there is a robot that occupies vertex 1. Starting from the $4d+1$ round, the remaining robots try now to reach the vertex 2. This process can continues until the p-th vertex thanks to the Property 1. Therefore, after $p \times 4d$ rounds, the R robots occupy the vertices of the grid labeled $1, 2, \ldots, R$. It is important to notice that, up to now, no robot knows the value of R.

evaluating_$\mathcal{P}_1$: *Line 03.* If the graph belongs to a (p, q) class of graphs with $q = 0$, the evaluation of the predicate is easy: since all robots know the graph, they know this value of $q = 0$ and then they evaluate the predicate to true. The following text concerns then only cases when $q > 0$.

The main idea is that at the beginning of this step, there is a set of robots (possibly empty) that is able to trivially evaluate the predicate $\mathcal{P}_1$. Then these robots (if any) coordinate to communicate to the remaining robots the result of the evaluation. Remember that the only way for robots to communicate is to move and to observe the occupation of vertices.

At the beginning of this step, the robots occupy the vertices of the grid labeled $1, 2, \ldots, R$, if there are some robots on the vertices labeled from $p - q + 1$ to p, they can trivially compute the predicate to false: indeed since any robot of this set occupies a vertex whose label is greater than $p - q$, it can conclude that there are more than $p - q$ robots in the grid. Thus these robots immediately evaluate the predicate to false.

In $(p - q)$ phases (starting at phase 1) all the other robots in the grid will be able to evaluate the predicate. Each phase takes $4d$ synchronization rounds. In particular, in the i-th phase the only robot which evaluates $\mathcal{P}_1$ is the one (if any) located at the vertex $p - q - i + 1$ (notice that for the first phase it corresponds to the vertex $p - q$ which is the highest one that could not compute immediately the predicate). This latter evaluates the predicate $\mathcal{P}_1$ as false if at the end of the i-th phase, it observes that its father vertex (as defined in Property 2) is occupied by a robot; true otherwise (i.e. its father vertex is free at that point).

At the beginning of phase i, the robots (if any) that occupy the vertices labeled from p until $p - q - i + 2$ knows if the predicate $\mathcal{P}_1$ is true or false, because they evaluated it during previous phases (or initially). So, they coordinate to ensure that at the end of the i-th phase, the robot located on the $p - q - i + 1$ vertex evaluates $\mathcal{P}_1$ correctly. In particular, if the predicate is false, by the end of the i-th phase these robots move to let one of them occupy the father of the vertex $p-q-i+1$. On the contrary, if the predicate is true, they move in order to make the father of the vertex $p - q - i + 1$ empty.

At round $4d$ of the phase $i = (p-q)$, all the robots agree on the value of the predicate $\mathcal{P}_1$. If the predicate is true, then they can start the last step and run the exploration algorithm $\mathsf{A}_\infty()$ (defined in Theorem 2) to ensure the perpetual exploration property.

evaluating_$\mathcal{P}_1$: *Line 04.* In order to execute (when $R \leq p - q$) the algorithm $A_{+\infty}()$ each robot has to know the location of every other robots. (The algorithm $A_{+\infty}()$, defined in Theorem 2 f-solves the $CPGE$ problem when the robots have an infinite radius ρ and $f(p, q) \leq p - q$ [3].) This is why line 02 is repeated in order to place robots in the first R position and then, after $p \times 4d$ rounds, every robot knows this positioning. However this repositioning is not enough since robots do not know the exact number R of robots, they just know $R \leq p - q$. The solution consists for each robot to assume there are exactly $p - q$ robots, and then the execution of the algorithm $\mathsf{explore}_\infty()$ is done with $(p - q) - R$ virtual robots that occupy the vertices labeled from $R + 1$ to $p - q$. (The introduction of virtual robots ensures that each robot starts the execution with the same configuration.)

6 Conclusion

To conclude, the following table summarizes the results of the paper:

Value of ρ	f-solvability of the $CPGE$ problem	Ref
$\rho = 0$	$f(1,0) \le 1$ and $f(p,q) = 0$ otherwise	[3]
$\rho = 1$	$f(p,0) \le p - 1$ when $p > 1$, and $f(p,q) \le p - q$ otherwise	[this paper]
$1 < \rho < +\infty$	$f(p,0) \le p - 1$ when $(p > 1 \wedge\ \mathcal{Q}_\rho)$, and $f(p,q) \le p - q$ otherwise	[conjecture]
$\rho = +\infty$	$f(p,q) \le p - q$	[3]

It is easy to see that $f(p,q) \le p - q$ is an upper bound on the number of robots in all cases (recall that $q = 0$ when $p = 1$). The case $\rho = 0$ requires that the grid be trivial (only one vertex), while there is no requirement on the structure of the grid for f-solving the $CPGE$ problem when $1 \le \rho \le +\infty$. It follows that $\rho = 1$ is a strong demarcation line delineating the cases from which f-solving the $CPGE$ problem becomes relevant.

The case $1 \le \rho \le +\infty$ shows that, the maximal number of robots for which one can f-solve the $CPGE$ problem, depends on the structure of the grid. This is captured by the parameter q derived from its structure (thanks to the notion of mobility tree).

When $1 \le \rho < +\infty$ and $q \ne 0$, $(p - q)$-solving the $CPGE$ problem is always possible, and is optimal (in the number of robots). When $q = 0$, there are cases where the maximal number of robots for which the $CPGE$ problem can be f-solved is smaller than $p - q = p - 0$, namely it is $p - 1$. The paper has identified the corresponding grid structures when $\rho = 1$: they are all the non-trivial grids (more than one vertex). As far as the cases $1 < \rho < +\infty$ are concerned, we conjecture that $f(p,0) \le p - 1$ when $(p > 1 \wedge\ \mathcal{Q}_\rho)$ where $\mathcal{Q}_\rho$ is a property that depends on the cycles that can be seen within the radius ρ. Moreover, we also conjecture that, given a grid, this property is such that $\mathcal{Q}_1 = \mathit{true}$, $\mathcal{Q}_{+\infty} = \mathit{false}$, and $\forall \rho : \mathcal{Q}_{\rho+1} \Rightarrow \mathcal{Q}_\rho$.

References

1. Awerbuch, B., Betke, M., Rivest, R.L., Singh, M.: Piecemeal Graph Exploration by a Mobile Robot. Information and Computation 152(2), 155–172 (1999)
2. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: Anonymous Graph Exploration without Collision by Mobile Robots. Inforamtion Processing Leters (to appear, 2008)
3. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: On the Solvability of Anonymous Partial Grids Exploration by Mobile Robots. Tech Report #1892, p. 21 , IRISA, Université de Rennes 1, France (2008), ftp.irisa.fr/techreports/2008/PI-1892.pdf
4. Dobrev, S., Jansson, J., Sadakane, K., Sung, W.K.: Finding Short Right-Hand-on-the-Wall Walks in Undirected Graphs. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 127–139. Springer, Heidelberg (2005)

5. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Computing Without Communicating: Ring Exploration by Asynchronous Oblivious Robots. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 105–118. Springer, Heidelberg (2007)
6. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph Exploration by a Finite Automaton. Theoretical Computer Science 345(2-3), 331–344 (2005)
7. Fraigniaud, P., Ilcinkas, D., Rajsbaum, S., Tixeuil, S.: Space Lower Bounds for Graph Exploration via Reduced Automata. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 140–154. Springer, Heidelberg (2005)
8. Franchi, A., Freda, L., Oriolo, G., Vendittelli, M.: A Randomized Strategy for Cooperative Robot Exploration. In: Int'l Conference on Robotics and Automation (ICRA 2007), pp. 768–774. IEEE press, Los Alamitos (2007)
9. Grossi, R., Pietracaprina, A., Pucci, G.: Optimal Deterministic Protocols for Mobile Robots on a Grid. Information and Computation 173(2), 132–142 (2002)
10. Ilcinkas, D.: Setting Port Numbers for Fast Graph Exploration. In: Flocchini, P., Gasieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 59–69. Springer, Heidelberg (2006)
11. Panaite, P., Pelc, A.: Impact of Topographic Information on Graph Exploration Efficiency. Networks 36(2), 96–103 (2000)
12. Rollik, H.A.: Automaten in Planaren Graphen. Acta Informatica 13, 287–298 (1980)
13. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM Journal on Computing 28(4), 1347–1363 (1999)
14. Yared, R., Defago, X., Wiesmann, M.: Collision Prevention Using Group Communication for Asynchronous Cooperative Mobile Robots. In: Proc. 21st Int'l IEEE Conference on Advanced Information Networking and Applications (AINA 2007), pp. 244–249. IEEE Computer Press, Los Alamitos (2007)

Taking Advantage of Symmetries: Gathering of Asynchronous Oblivious Robots on a Ring*

Ralf Klasing[1], Adrian Kosowski[1,2], and Alfredo Navarra[3]

[1] LaBRI - Université Bordeaux 1 - CNRS, 351 cours de la Liberation, 33405
Talence cedex, France
{Ralf.Klasing,Adrian.Kosowski}@labri.fr
[2] Department of Algorithms and System Modeling, Gdańsk University of Technology,
Narutowicza 11/12, 80952 Gdańsk, Poland
kosowski@sphere.pl
[3] Dipartimento di Matematica e Informatica, Università degli Studi di Perugia,
Via Vanvitelli 1, 06123 Perugia, Italy
navarra@dipmat.unipg.it

Abstract. One of the recently considered models of robot-based computing makes use of identical, memoryless mobile units placed in nodes of an anonymous graph. The robots operate in Look-Compute-Move cycles; in one cycle, a robot takes a snapshot of the current configuration (Look), takes a decision whether to stay idle or to move to one of the nodes adjacent to its current position (Compute), and in the latter case makes an instantaneous move to this neighbor (Move). Cycles are performed asynchronously for each robot.

In such a restricted scenario, we study the influence of symmetries of the robot configuration on the feasibility of certain computational tasks. More precisely, we deal with the problem of gathering all robots at one node of the graph, and propose a solution based on a symmetry-preserving strategy. When the considered graph is an undirected ring and the number of robots is sufficiently large (more than 18), such an approach is proved to solve the problem for all starting situations, as long as gathering is feasible. In this way we also close the open problem of characterizing symmetric situations on the ring which admit a gathering [R. Klasing, E. Markou, A. Pelc: Gathering asynchronous oblivious mobile robots in a ring, Theor. Comp. Sci. 390(1), 27-39, 2008].

The proposed symmetry-preserving approach, which is complementary to symmetry-breaking techniques found in related work, appears to be new and may have further applications in robot-based computing.

Keywords: Asynchronous system, Mobile robots, Oblivious robots, Gathering problem, Ring.

* The research was partially funded by the State Committee for Scientific Research (Poland) Grant 4 T11C 047 25, by the ANR-project "ALADDIN" (France), by the project "CEPAGE" of INRIA (France), and by European projects COST Action 293 "Graphs and Algorithms in Communication Networks" (GRAAL) and COST Action 295 "Dynamic Communication Networks" (DYNAMO).

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 446–462, 2008.

1 Introduction

The difficulty of many computational problems involving mobile entities (robots) is aggravated when robots cannot communicate directly, but have to take decisions about their moves only by observing the environment. One of the most restrictive scenarios considered in literature is the asynchronous Look-Compute-Move model for memoryless units which has been studied both for robots on the plane (the *continuous model* [13,20]) and for robots located on the nodes of a graph (the *discrete model* [10,11,16]). Herein we focus on computations in the discrete model which is described in more detail below.

1.1 The Discrete Model

Consider an anonymous graph in which neither nodes nor links have any labels. Initially, some of the nodes of the graph are occupied by robots and there is at most one robot in each node. Robots operate in Look-Compute-Move cycles. In each cycle, a robot takes a snapshot of the current configuration (Look), then, based on the perceived configuration, takes a decision to stay idle or to move to one of its adjacent nodes (Compute), and in the latter case makes an instantaneous move to this neighbor (Move). Cycles are performed asynchronously for each robot. This means that the time between Look, Compute, and Move operations is finite but unbounded, and is decided by the adversary for each robot. The only constraint is that moves are instantaneous, and hence any robot performing a Look operation sees all other robots at nodes of the ring and not on edges. However, a robot r may perform a Look operation at some time t, perceiving robots at some nodes, then Compute a target neighbor at some time $t' > t$, and Move to this neighbor at some later time $t'' > t'$, at which some robots are in different nodes from those previously perceived by r because in the meantime they performed their Move operations. Hence, robots may move based on significantly outdated perceptions. It should be stressed that robots are memoryless (oblivious), i.e., they do not have any memory of past observations. Thus, the target node (which is either the current position of the robot or one of its neighbors) is decided by the robot during a Compute operation solely on the basis of the location of other robots perceived in the previous Look operation. Robots are anonymous and execute the same deterministic algorithm. They cannot leave any marks at visited nodes, nor send any messages to other robots.

It is assumed that the robots have the ability to perceive, during the Look operation, if there is one or more robots located at the given node of the graph. This capability of robots is important and well-studied in the literature on robot gathering under the name of *multiplicity detection* [13,20]. In fact, without this capability, many computational problems (such as the gathering problem considered herein) are impossible to solve for all non-trivial starting configurations. It should be stressed that, during a Look operation, a robot can only tell if at some node there are no robots, there is one robot, or there is more than one robot: a robot does not see the difference between a node occupied by a or b robots, for distinct $a, b > 1$.

Problems studied so far in the discrete model include gathering on the ring [16], exploration of the ring [10], and tree exploration [11].

1.2 Our Results

In this paper, we consider one of the most fundamental problems of self-organization of mobile entities, known in the literature as the *gathering* problem. Robots, initially situated at different locations, have to gather at the same location (not determined in advance) and remain in it. Our considerations focus on gathering robots in the discrete model for the undirected ring; such a scenario poses a number of problems due to the high number of potential symmetries of the robot configuration. This problem was initially studied in [16], where certain configurations were shown to be gatherable by means of symmetry-breaking techniques, but the question of the general-case solution was posed as an open problem. Herein we provide procedures for gathering all configurations on the ring with more than 18 robots for which gathering is feasible, and give a full characterization of all such configurations (Theorem 6). In fact, we provide a new technique for dealing with symmetric configurations: our approach is based on preserving symmetry rather than breaking it.

1.3 Related Work

The problem of gathering mobile robots in one location has been extensively studied in the literature. Many variations of this task have been considered in different computational models. Robots move either in a graph, cf. e.g. [2,8,9,12,17], or in the plane [1,3,4,5,6,7,13,19,20,21], they are labeled [8,9,17], or anonymous [1,3,4,5,6,7,13,19,20,21], gathering algorithms are probabilistic (cf. [2] and the literature cited there), or deterministic [1,3,4,5,6,7,8,12,13,17,19,20,21]. Deterministic algorithms for gathering robots in a ring (which is a task closest to our current setting) have been studied e.g. in [8,9,12,14,17]. In [8,9,17] symmetry was broken by assuming that robots have distinct labels, and in [12] it was broken by using tokens. The very weak assumption of anonymous identical robots was studied in [1,3,4,5,6,7,13,19,20,21] where robots could move freely in the plane. The scenario was further refined in various ways. In [4,14] it was assumed that robots have memory, while in [1,3,5,6,7,13,19,20,21] robots were oblivious, i.e., it was assumed that they do not have any memory of past observations. Oblivious robots operate in Look-Compute-Move cycles, similar to those described in our scenario. The differences are in the amount of synchrony assumed in the execution of the cycles. In [3,21] cycles were executed synchronously in rounds by all active robots, and the adversary could only decide which robots are active in a given cycle. In [4,5,6,7,13,19,20] they were executed asynchronously: the adversary could interleave operations arbitrarily, stop robots during the move, and schedule Look operations of some robots while others were moving. It was proved in [13] that gathering is possible in the asynchronous model if robots have the same orientation of the plane, even with limited visibility. Without orientation, the gathering problem was positively solved in [5], assuming that robots have the capability of multiplicity detection. A complementary negative result concerning the asynchronous model was proved in [20]: without multiplicity detection, gathering robots that do not have orientation is impossible.

2 Terminology and Preliminaries

We consider an n-node anonymous ring without orientation. Initially, some nodes of the ring are occupied by robots and there is at most one robot in each node.

During a Look operation, a robot perceives the relative locations on the ring of multiplicities and single robots. We remind that a multiplicity occurs when more than one robot occupies the same location. For the purpose of the definition only, let us call one of the directions on the cycle *clockwise*, and the other *anti-clockwise*. Then, for a fixed robot r, let $S_C(r)$ denote the ordered sequence of distances from r to all single robots when traversing the cycle in the clockwise direction, and let $S_A(r)$ be the ordered sequence of such distances when moving anti-clockwise. Sets $M_C(r)$ and $M_A(r)$ are likewise defined for distances from r to all multiplicities. Then the *view* $V(r)$ provided to the robot r is defined as the set of ordered pairs $V(r) = \{(S_C(r), M_C(r)), (S_A(r), M_A(r))\}$. If there are no multiplicities, we will drop the second sequence in each case and write the view simply as the set of two sequences $V(r) = \{S_C(r), S_A(r)\}$.

The current configuration C of the system can be described in terms of the view of a robot r which is performing the Look operation at the current moment, but disregarding the location of robot r; formally, $C = \{\{(S_C(r) \oplus i, M_C(r) \oplus i), (S_A(r) \ominus i, M_A(r) \ominus i)\} : i \in [1, n]\}$, where operations $\oplus$ and $\ominus$ denote modulo n addition and subtraction, respectively. Note that the configuration is independent of the choice of robot r and of the choice of the clockwise direction.

A configuration C is called *periodic* if it is invariable under rotation, i.e. $C = C \oplus k$ for some integer $k \in [1, n-1]$. A configuration C is called *symmetric* if the ring has a geometrical *axis of symmetry*, which reflects single robots into single robots, multiplicities into multiplicities, and empty nodes into empty nodes. Note that a symmetric configuration is not periodic if and only if it has exactly one axis of symmetry [16]. A symmetric configuration C with an axis of symmetry s has an *edge-edge symmetry* if s goes through (the middles of) two antipodal edges; it has a *node-on-axis symmetry* if at least one node is on the axis of symmetry.

A *pole* is an intersection point of a line with the ring (this may either be a node or in between two nodes). For configurations with a single axis of symmetry, nodes on the axis of symmetry are natural gathering points. The pole of the axis of symmetry used by the considered algorithm for gathering is known as the *North pole*, the other pole is called the *South pole*.

The set of nodes of the ring forming a path between two robots, excluding endpoints, is called an *arc*. Two robots are called *neighbors* if at least one of the two arcs of the ring between them does not contain any robots. When uniquely defined, the arc of the ring between two neighboring robots u, v with no robots on it is called the *gap* $u - v$. The length of gap $u - v$ is denoted as $|u - v|$, obviously $|u - v| = |v - u|$. Two robots forming a multiplicity are assumed to form a gap of length 0. A gap of minimum length in a given configuration is simply called *minimal*.

The notation for gaps is extended to allow for *chains*, $u_1 - u_2 - \ldots - u_k$, i.e. sequences of robots separated by gaps. If some robots $u_i - \ldots - u_j$ form a multiplicity M, then the considered chain may be written compactly as $u_1 - \ldots - u_{i-1} - M - u_{j+1} - \ldots - u_k$.

We now introduce the concept of *extrapolated length* $|u \rightarrow v|$ of a gap $u - v$, useful for breaking ties in the gathering process. Let $u - v - v_1 - v_2 - \ldots - v_s$ be the longest possible chain such that for all i, $v_i \neq u$ and v_i does not belong to a multiplicity. Then $|u \rightarrow v| = (|u - v|, |u - v_1|, |u - v_2|, \ldots, |u - v_s|)$. Values of extrapolated gap lengths are compared lexicographically.

A key operation used in the gathering process is known as the *contraction* of a gap. Let $u - v$ be an arbitrary gap belonging to some chain $t - u - v - w$, such that $|u \rightarrow t| > |v \rightarrow w|$. Then the *contraction of* $u - v$ is the operation of moving robot u a single node towards robot v.

Note that if a configuration C' was formed from a configuration C by contraction of some gap $u - v$ (by moving u) in a chain $t - u - v - w$, then it is clear that in C' we have $|t - u| > |v - w|$. The corresponding *de-contraction of* $u - v$ in C' is uniquely defined as the operation of moving robot u a single node away from robot v unless some other symmetry has been determined.

3 Gathering Algorithm

Our algorithm describe the Compute part of the cycle of robots' activities. In order to simplify notation, they are often expressed using configurations (identical for all robots sharing the same snapshot of the system), and not locally-centered views. For example, if we require only robots specifying certain geometrical criteria to move, then each robot will be able to recognize whether to perform the specified action or not.

A gathering algorithm in [16] called RigidGathering provides a solution for all *rigid* configurations, i.e. configurations which are not symmetric and not periodic. It uses a sequential approach: in every configuration, exactly one specific robot is chosen by the algorithm (regardless of the robot running the algorithm), and this robot is then allowed to perform a move.

In our approach, we define an algorithm able to manage all symmetric and gatherable configurations, by allowing at any given time exactly two symmetric robots, u and $\bar{u}$, to make corresponding moves, hence preserving symmetry. Observe that there are two possible move scenarios leading to different types of new configurations:

- *Both robots, u and $\bar{u}$, make their moves simultaneously.* In this case, in the new configuration the axis of symmetry remains unchanged. For the correctness of the algorithm, it is essential to ensure that no new axes of symmetry are formed (since otherwise the new configuration cannot be gathered).
- *One of the robots, say u, performs its move before the other robot $\bar{u}$.* All other robots must be able to recognize that the current configuration is one move

away from a symmetry, and robot $\bar{u}$ is now the only one allowed to perform a move. Observe that it is then irrelevant whether robot $\bar{u}$ performed its Look operation before or after robot u was moved; the outcome of its move is exactly the same.

The algorithm proposed herein detects configurations which have exactly one axis of symmetry of the node-on-axis type (which we call *A-type configurations*) and those which are exactly one step away from such a configuration (*B-type configurations*). For all other gatherable non-symmetrical configurations (*C-type configurations*), a step of the RigidGathering algorithm from [16] is performed. It is assumed that the number of robots is larger than 18 (for an explanation of this value, cf. Remark 1). Thus, for example if the system starts in an A-type configuration, it remains in A-type configurations possibly alternating with B-type configurations. If the system starts without an axis of symmetry, it may either perform a gathering passing through C-type configurations only, or may at some point switch from a C-type configuration to a B-type configuration, and then remain confined to B-type an A-type configurations. In consequence, the eventual convergence of our algorithm to a gathering relies only on the convergence of the RigidGathering algorithm from [16], and on the convergence of the rules we introduce for A-type and B-type configurations.

Our algorithm runs in four main phases; these are informally outlined in the following subsection, and formalized in Subsection 3.2.

3.1 Illustration of Approach

Let us suppose that the system starts in an A-type configuration. (Note that in view of impossibility results from [16] (see Theorem 5), symmetric configurations which are not A-type configurations are never gatherable.) We temporarily also assume here that there are initially no robots on the axis of symmetry.

The four phases of our algorithm can be outlined as follows. In the first phase of the algorithm, we lead the system to an A-type configuration with exactly two (symmetrical) multiplicities. In the second phase, all of the other robots (with the exception of two symmetrically located robots called *guards*) are gathered into the multiplicities. In the third phase, the multiplicities are moved to their final gathering point on the axis of symmetry, away from the guards (remember that there is a node-on-axis symmetry in our case). Finally, in the fourth phase the guards join the single remaining multiplicity in the gathering point.

The current phase of the algorithm can be determined by only looking at the state of the system; this will be discussed in more detail later. The single axis of symmetry is maintained throughout the process. In the first phase, the locations of all minimal gaps are used for this purpose. In the second phase the axis is determined by positions of the multiplicities, while in the third phase by the positions of the guards. Finally, in the fourth phase the gathering point with the only multiplicity is known.

Referring to Figures 1 and 2, we now describe in more details the basic intuitions of our algorithm. In both figures, configurations a describe two possible

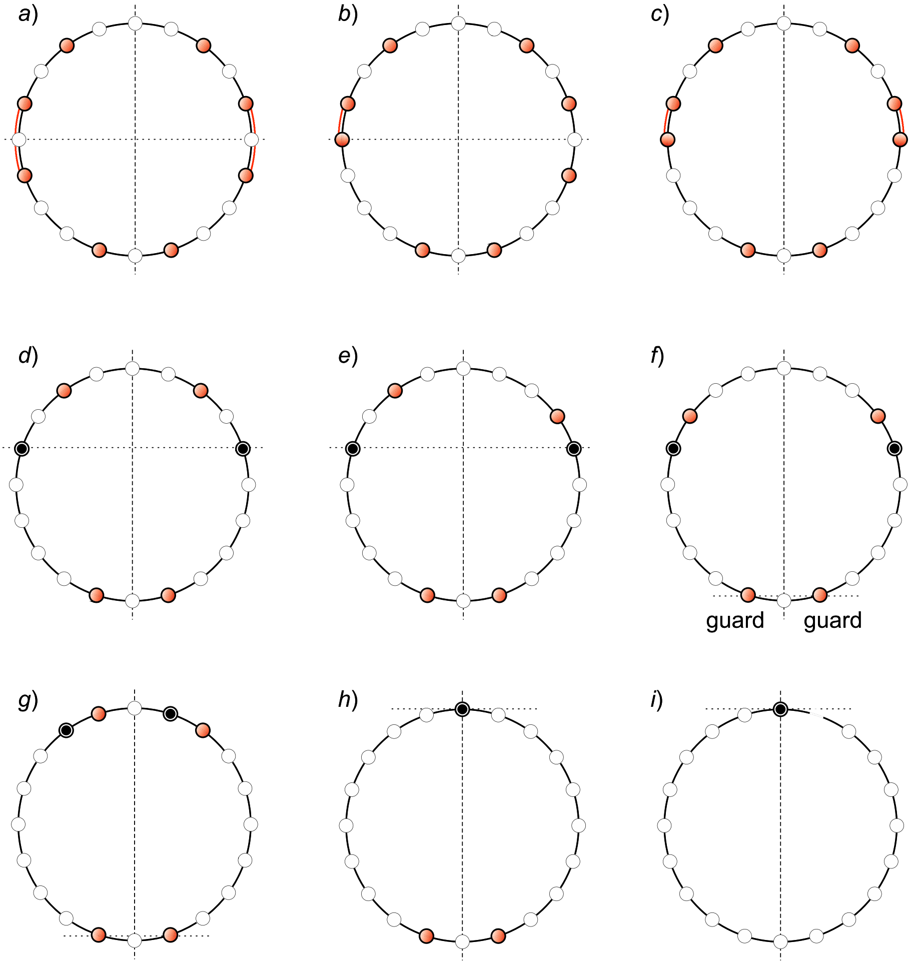


Fig. 1. An example of a scenario for a symmetric configuration. White nodes represent empty nodes, shaded nodes are nodes occupied by a single robot, black nodes are nodes occupied by at least two robots, i.e., multiplicities. The North pole is at the top of the axis of symmetry. The dashed horizontal line can be understood as a helper line for recognizing the axis of symmetry.

initial states of the system (A-type configurations). In the first phase, the objective is to create two symmetric multiplicities such that both arcs of the circle between them contain at least two robots, neither of which is at a distance of one from a multiplicity. The normal move (Fig. 1*a*) consists in the contraction of two symmetrical minimal gaps. The pair of minimal gaps is selected in such a way that the contraction does not create two multiplicities which violate the imposed constraint on robots on the arcs between them; if there exists a minimal gap crossing the axis of symmetry, this is not chosen either. It may happen that no minimal gap appropriate for contraction exists (Fig. 2*a*). In that case,

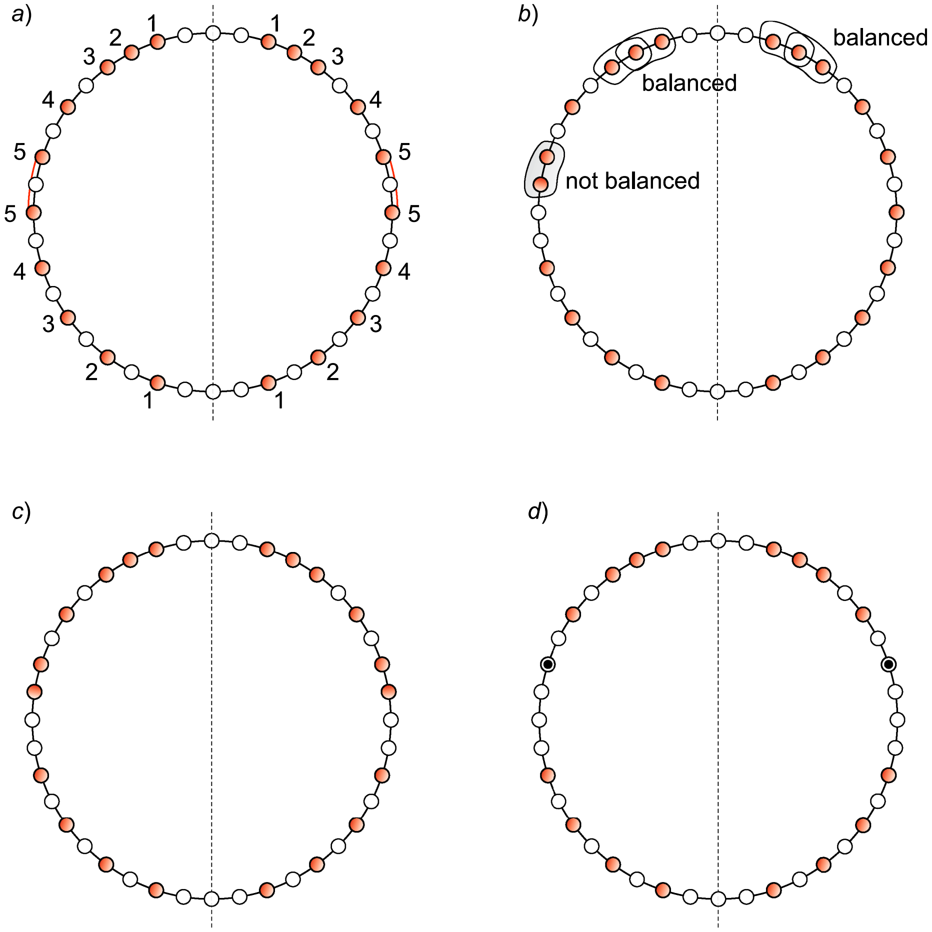


Fig. 2. An example of a scenario for a symmetric configuration (contraction of equatorial gaps) Details of the construction are given in Subsection 3.3

we select for contraction the pair of (not necessarily minimal) gaps which are central in terms of the number of robots separating them from the poles of the axis of symmetry (gaps between robots 5–5 in Fig. 2*a*); if there are two pairs of candidate gaps, a tie-breaking mechanism is applied.

The performed contractions result in a new symmetrical configuration (configurations *c* in both figures), possibly preceded by a state of violated symmetry in a B-type configuration (configurations *b*). The process of selecting the gap for contraction allows the robots to recreate configuration *a* knowing configuration *b* only, and to proceed from there to configuration *c*. Configuration *c* is in fact an A-type configuration just as configuration *a*, and the procedure repeats until the two sought multiplicities are created (configurations *d*). At this point, the first phase of the algorithm is complete. Note that the contraction rules applied in

Fig. 2 require a sufficiently large number of robots (more than 18, see Remark 1) to guarantee correctness.

The next phases of the algorithm are shown in Fig. 1 only. In phase 2 it is necessary to decide on one of the two poles of the axis of symmetry as the gathering point (the North pole). The poles are chosen so that the northern arc between multiplicities has more robots than the southern arc; in case of a tie, the side on which the nearest robots are closer to the multiplicities is the northern one. The robots are moved in symmetrical pairs towards their respective multiplicities, starting from the robots on the northern arc (Fig. 1e, f). Note that the definition of the North and South is consistently preserved throughout the process. Phase 2 ends when nearly all the robots have been merged into the multiplicities, and the remaining robots occupy not more than 6 nodes in an arrangement matching a specific pattern (Fig. 1f). Two robots, separated by gaps from the multiplicities, always remain on the southern arc and act as the guards of the axis of symmetry throughout phase 3. The multiplicities are moved step by step towards the North pole; note that not all the robots in a multiplicity have to move simultaneously (Fig. 1g). When the pattern shown in Fig. 1h is achieved, phase 4 starts, and the two remaining robots are moved step by step until they reach the North pole (Fig. 1i), and the algorithm is complete.

3.2 Formalization of Approach

The distinction among the four phases of the proposed algorithm is in fact possible knowing only the current configuration C. To do this, we now introduce some further notation.

A configuration can also be represented in the form of a string of characters as follows: starting from an arbitrary node and moving around the cycle in a chosen direction, for each node we append a character representing whether the node is empty, contains a single robot, or a multiplicity. We say that configuration C matches a *chain pattern* $[P]$, $C \in [P]$, if there exists a string representation of C belonging to the language described by the tokens in $[P]$. For some integer values a and b, token $\sigma_{a:b}$ is understood as between a and b occurrences of single robots (possibly separated by any number of empty nodes) followed by at least one empty node. Token $\mu_{a:b}$ is understood as between a and b occurrences of consecutive non-empty nodes, at least one of which is a multiplicity, followed by at least one empty node. Ranges of the form $a : a$ are simply written as a. For example, in Fig. 1 the pattern $[\mu_{1:2}, \sigma_{1:2}, \mu_2]$ is matched by configurations f and g.

Herein we restrict ourselves to a presentation of the algorithm for the case of an initial configuration with exactly one axis of symmetry, having a node-on-axis type symmetry, without any robots on the axis. The case that allows robots to reside on the axis of symmetry can be addressed by a minor modification of the algorithm as outlined at the end of the section.

The proposed algorithm performs the gathering in four basic phases, as defined in Table 1.

When performing its Compute step, each robot can clearly determine which phase of the algorithm it is currently running (cases not covered in the table

Table 1. Division into phases, assuming no robots on the axis of symmetry in the initial state: $m(C)$ — number of multiplicities in configuration C, $p(C)$ — total number of different nodes occupied by robots in C

Phase	Multiplicities	Occupied nodes	Additional constraints
1	$m(C) < 2$	$p(C) > 6$	none
2	$m(C) = 2$	$p(C) \geq 6$	if $p(C) = 6$, then $C \notin [\mu_{1:2}, \sigma_2, \mu_{1:2}]$
3	$m(C) = 2$	$4 \leq p(C) \leq 6$	if $p(C) = 6$, then $C \in [\mu_{1:2}, \sigma_2, \mu_{1:2}]$
4	$m(C) \geq 1$	$p(C) \leq 3$	none

cannot appear in the initial state and do not occur later due to the construction of the algorithm). The algorithm is defined so as to guarantee that when two robots are allowed to move simultaneously, their views correspond to the same phase of the algorithm. Bearing this in mind, we can now consider the four phases separately in the following subsections.

3.3 Phase 1: Obtaining Two Non-adjacent Multiplicities

The algorithm is defined by the following elements:

- A subroutine defining a move for an A-type configuration which leads to a new A-type configuration, assuming that both the robots which are chosen to move perform their action simultaneously.
- A subroutine for detecting the preceding A-type configuration when the current state of a system is a B-type configuration.

The procedure for A-type configurations is presented as Algorithm 1. A gap $u - v$ is called *equatorial* with respect to a line s if the number of robots on the arc from u to one pole of s and from v to the other pole of s differs by at most 1 (a multiplicity is counted as 2 robots).

For completeness of the procedure, it is necessary to provide some mechanism of choosing one of several possible candidate gaps. Such ties are easily broken, since for a given configuration it is possible to define a partial order on the set of robots in which only symmetrical robots are not comparable [16].

The definition of the procedure always allows a move of exactly two symmetrical robots. We now show that the above set of rules is sufficient to gather an A-type configuration, provided that both symmetrical robots always perform their Look operations as well as Move operations simultaneously (we will call this a *symmetry-preserving scheduler*).

Case of a Symmetry-Preserving Scheduler. Before proceeding with the proofs, we recall the obvious geometrical fact that if for a configuration on the ring it is in some way possible to distinguish (select) exactly two arcs, then the configuration can only have zero, one, or two perpendicular axes of symmetry.

Algorithm 1. Procedure for A-type configurations (Phase 1)

```
(i) Choose a pair of minimal gaps $u - v$ and $\bar{u} - \bar{v}$ such that the following conditions
    are fulfilled:
      • $u - v$ does not intersect the axis of symmetry ($u \neq \bar{v}$),
      • the contraction of $u - v$ and $\bar{u} - \bar{v}$ does not create two multiplicities with no
        other robots in between them,
      • the contraction of $u - v$ and $\bar{u} - \bar{v}$ does not create two multiplicities with
        exactly two robots in between, adjacent to these multiplicities,
    then perform the contractions of $u - v$ and $\bar{u} - \bar{v}$.
(ii) If no such pair exists, perform the contraction of chosen gaps $u - v$ and $\bar{u} -$
    $\bar{v}$ which are equatorial with respect to the axis of symmetry. If there are two
    pairs of equatorial gaps of different lengths, the shorter pair is always chosen for
    contraction.
```

Theorem 1. *Under a symmetry-preserving scheduler, the new configuration after performing rule* (i) *is also an A-type configuration.*

Proof. Indeed, consider the contraction of minimal gap $u{-}v$ in a chain $t{-}u{-}v{-}w$ and its complement $\bar{u} - \bar{v}$ in chain $\bar{t} - \bar{u} - \bar{v} - \bar{w}$. The obtained configuration has exactly two minimal gaps, $u - v$ and $\bar{u} - \bar{v}$, and $|t - u| \neq |v - w|$ by the properties of a contraction. Thus, after the move the axis of symmetry remains unchanged and no new axes are created, since gap $u - v$ must be reflected into $\bar{u} - \bar{v}$. □

For a given configuration C, we will call a gap $u - v$ *balanced* if for the chain $s{-}t{-}u{-}v{-}w{-}x$ we have $|t{-}u| = |v{-}w|$ or $|u{-}v| \in \{|s{-}t|, |t{-}u|, |v{-}w|, |w{-}x|\}$.

Remark 1. If for a given A-type configuration rule (i) cannot be applied, we can make the following statements:

– The set of minimal gaps consists of between 1 and 10 gaps formed by at most 12 robots — at most 6 robots surrounding each pole of the axis of symmetry (3 robots on one side and 3 on the other); otherwise, a minimal gap formed by any other robots can always be contracted.
– All the minimal gaps are balanced; this can be shown by a simple enumeration of all possibilities.
– Taking into account the assumption that there are more than 18 robots on the ring, there exist on the cycle exactly two symmetrical maximal arcs containing more than $\frac{18-12}{2} = 3$ (i.e. at least 4) robots each, such that none of these robots are part of some minimal gap.

Taking into account the above remark, we proceed to prove the following.

Theorem 2. *Under a symmetry-preserving scheduler, the new configuration after performing rule* (ii) *is also an A-type configuration.*

Proof. We need to consider two cases:

(1) if the contraction of $u - v$ and $\bar{u} - \bar{v}$ creates no new minimal distances, then the axis of symmetry and the set of minimal gaps remain unchanged. There could exist at most one more candidate for an axis of symmetry for the new configuration, perpendicular to the original axis. Since the number of robots on both sides of an axis of symmetry is the same, either the new axis crosses the newly contracted gaps $u - v$ and $\bar{u} - \bar{v}$ or it crosses u and $\bar{u}$. In the first case we have a contradiction since for the chains $t - u - v - w$ we have $|t - u| \neq |v - w|$. In the second case, a contradiction arises as well, since the shortest equatorial gaps have been contracted and hence the shortest gaps cannot be reflected by the new axis into the longest ones.

(2) if the contraction of $u - v$ and $\bar{u} - \bar{v}$ creates two new minimal distances, then these are the only two non-balanced minimal distances on the ring. By a similar argument as before, these two non-balanced minimal distances must be reflected by the axis into each other, so the axis of symmetry is unique. □

Finally, we make a note on the convergence of the performed process.

Theorem 3. *Under a symmetry-preserving scheduler, Phase 1 is completed after a finite number of steps.*

Proof. If rule (i) is performed then the length of the minimal gap decreases in each step. Otherwise, the length of the equatorial gap decreases, while the length of the minimal gap remains unchanged (since all minimal gaps are then concentrated around the poles). The process obviously converges to a minimal gap length of 0, hence we obtain 2 multiplicities and, by Table 1, Phase 1 is complete. □

Extension to the General Scheduler. We now proceed to define the second required subroutine, namely, a procedure to show for a B-type configuration a unique preceding A-type configuration.

Depending on the rule used in the preceding A-type configuration and the outcome of the move, we have the following cases:

B1. The current configuration was obtained by contracting a minimal gap in an A-type configuration using rule (i).
B2: The current configuration was obtained by contracting an equatorial gap in an A-type configuration using rule (ii), but without creating any new minimal gaps in the process.
B3: The current configuration was obtained by contracting an equatorial gap in an A-type configuration using rule (ii), but creating a new minimal gap in the process.

Before proceeding any further, for a configuration we define a *compass axis* as any line s fulfilling the following constraints:

- s is an axis of symmetry of the set of balanced minimal gaps,
- the number of robots on both sides of s is equal,

- all the balanced minimal gaps are contained within a set of 12 robots — 6 robots surrounding each pole of s (3 robots on one side and 3 on the other).

We are now ready to prove the following theorem.

Theorem 4. *The sets of A-, B1-, B2-, and B3-type configurations are all pairwise disjoint.*

Proof. A B1-type configuration has exactly one non-balanced minimal gap. In consequence, such a configuration obviously cannot have an axis of symmetry.

A B2-type configuration has the same set of minimal gaps as the original A-type configuration, hence we can make use of Remark 1 also for this configuration. In consequence, a B2-type configuration has between 1 and 10 minimal gaps, all of which are balanced, and exactly one compass axis identical to the axis of symmetry of the original A-type configuration. Since the compass axis of a configuration is the only possible candidate for its axis of symmetry, and a B2-type configuration is exactly one move apart from an A-type configuration having this axis as an axis of symmetry, a B2-type configuration has no axes of symmetry.

A B3-type configuration has the same set of balanced minimal gaps as the original A-type configuration, and additionally one more non-balanced gap obtained as a result of the contraction (thus between 2 and 11 minimal gaps in total). As in the previous case, this means that a B3-type configuration has exactly one compass axis and no axis of symmetry.

Table 2. Telling apart different types of configurations: $q(C)$ — total number of minimal gaps in C, $q_b(C)$ — total number of balanced minimal gaps in C, $s(C)$ — number of axes of symmetry

Type	Minimal gaps	Balanced minimal gaps	Axes of symmetry
A	irrelevant	irrelevant	$s(C) = 1$
B1	$q(C) = 1$	$q_b(C) = 0$	$s(C) = 0$
B2	$1 \leq q(C) \leq 10$	$q_b(C) = q(C)$	$s(C) = 0$
B3	$2 \leq q(C) \leq 11$	$q_b(C) = q(C) - 1$	$s(C) = 0$

Taking into account the above observations (see Table 2), we obtain that for a given configuration C we can determine if it is an A-type configuration, or a candidate for a B1, B2, or B3-type configuration. In the latter cases, there exists exactly one possibility of recreating the potentially preceding A-type configuration. For a B1-type configuration, it is necessary to de-contract the unique minimal gap. For a B2 or B3-type configuration, the shortest of the gaps equatorial with respect to the compass axis should be de-contracted. If this preceding configuration is indeed an A-type configuration, then the next move is uniquely defined by imitating the stated procedure for A-type configurations. Otherwise, configuration C is some other (type C) configuration which does not require special treatment and can be solved following [16]. □

3.4 Phase 2: Partial Gathering with 2 Multiplicities

The first phase ends when two symmetrical multiplicities are created. Throughout the second phase of the algorithm, the two existing multiplicities M_1 and M_2 make no moves. Multiplicities M_1 and M_2 divide the ring into two parts, which we will call *northern* (around the North pole) and *southern* (around the South pole). Each of these parts initially contains at least two robots not directly adjacent to a multiplicity. Throughout the process North and South are defined in such a way as to fulfill the following conditions:

- the number of nodes in the northern part is odd,
- if both parts have an odd number of nodes, the southern part always contains not less than one robot, and not less robots than the northern part,
- if both parts have an odd number of nodes and contain the same number of robots, consider the chain $r_N - M_1 - r_S$ with robot r_N in the northern part and robot r_S in the southern part; then $|M_1 \to r_S| > |M_1 \to r_N|$.

The gathering procedure, presented as Algorithm 2, is defined so as to move all but at most 4 of the single robots into the two existing multiplicities (without creating any new multiplicities).

Algorithm 2. Procedure for Phase 2

```
(i)  If the northern part contains at least one robot, move a robot in the northern part,
     such that there are no robots between itself and one of the multiplicities, towards
     this multiplicity (in case of choice of robots, select the one with a longer way left
     to go; if the distance is the same, both robots are allowed to move).
(ii) Otherwise, perform an analogous operation in the southern part but for the two
     symmetric nodes closest to the pole (these nodes will play as guards in the next
     phase).
```

It is important to note that the adopted definition of North and South guarantees that the same labeling of the poles is maintained throughout the process.

In accordance with Table 1, the phase ends when all but at most four single robots have been merged with the multiplicities. The last pair of robots in the southern part has not yet made a move and is separated by at least one empty field from a multiplicity; these robots will serve as guards in the last phases of the algorithm.

3.5 Phase 3: Gathering 2 Multiplicities Using Guards

The third phase of the algorithm is performed when $C \in [\mu_{1:2}, \sigma_2, \mu_{1:2}]$. The two robots u and v corresponding to the token σ_2 define a unique axis of symmetry, orthogonal to the gap $u - v$. The remaining robots (and multiplicities) can move towards the North pole of this axis; for a given configuration, only those robots which have the longest way to go are allowed to move. In this way the configuration pattern is maintained throughout the process, until the moving robots converge on the three nodes near their destination. The configuration pattern then changes to $C \in [\mu_{1:3}, \sigma_2]$, and can be likewise maintained until all robots except for the guards gather at the required pole in a single multiplicity ($C \in [\mu_1, \sigma_2]$).

3.6 Phase 4: Withdrawing Guards to the Gathering Point

In this phase, the unique multiplicity on the North pole determines the gathering point for the remaining guards. The guards can be moved towards the multiplicity following the rule that if the guards are at a different distance from the multiplicity, the guard further away should move (in case of a tie, both guards are allowed to move). The configuration is maintained in the pattern $C \in [\mu_1, \sigma_2]$. Only in at most two final moves we have $C \in [\mu_{1:3}]$ or $C \in [\mu_{1:2}]$ (still with exactly one multiplicity). Eventually, $C \in [\mu_1]$ and the gathering is complete.

3.7 Remarks on the Algorithm

An extended version of the algorithm which is capable of additionally gathering the case of symmetrical configurations with at least one robot on the unique axis of symmetry can be designed analogously in four phases:

- Phase 1 of the algorithm remains unaffected. In the definition of the equatorial gap, the robot in the pole should be ignored.
- Phases 2, 3 and 4 are slightly modified to allow for a single guard robot on the South pole (instead of a pair of guard robots in the southern part).

Note that in the case of robots on the axis of symmetry it may also be possible to design algorithms which break the symmetry by immediately moving the robot located on the axis, as in the case of an odd number of robots described in [16]. In our approach symmetry is never broken until the robots from the poles are moved into a multiplicity (in particular, if there is a single guard robot on the South pole, in Phase 4 it has to be moved to the multiplicity on the North pole).

For configurations with more than 18 robots, our algorithm is complementary to the impossibility result shown in [16].

Theorem 5 ([16]). *Gathering is not feasible for initial configurations which are periodic or have an edge-edge symmetry.*

In this way, we have obtained the sought characterization of initial configurations on the ring.

Theorem 6. *For more than* 18 *anonymous and oblivious robots located on different nodes of a ring, gathering is feasible if and only if the initial configuration is not periodic and does not have an edge-edge symmetry.*

4 Conclusions

We have studied the gathering problem in the discrete model, solving it on a ring for any number of robots larger than 18. The applied technique relies on preserving symmetries (as a matter of fact, our algorithm occasionally creates symmetric configurations from asymmetrical initial configurations).

Theorem 6 implies that, for any number of robots larger than 18, gathering is feasible if and only if, in the initial configuration the robots can elect a node (not necessarily occupied by a robot). Although it is conjectured in [16] that such a claim should also hold in cases with an even number of robots between 4 and 18, this is not always true. For instance, the only possible configuration of 4 robots on a 5-node cycle is not gatherable, although the single empty node can be initially elected as a candidate for the gathering point. Providing an additional characterization for the cases of between 4 and 18 robots is an interesting open problem. Some partial results in this direction have recently been shown in [15].

A natural next step is to consider the gathering problem for other graph classes with high symmetry (such as toruses), and if possible propose an algorithmic approach which solves the problem in the general case. The gathering problem could also be considered for variants of the model, such as robots having limited visibility, although such restrictions often lead to a large number of initial configurations for which gathering is impossible. It is not clear whether allowing robots to have small (constant) memory would help address such problems with achieving a gathering. Finally, it is interesting to ask whether the technique of preserving symmetries proposed herein can also be applied in other contexts.

References

1. Agmon, N., Peleg, D.: Fault-Tolerant Gathering Algorithms for Autonomous Mobile Robots. SIAM Journal on Computing 36(1), 56–82 (2006)
2. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous. Kluwer Academic Publishers, Dordrecht (2002)
3. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed Memoryless Point Convergence Algorithm for Mobile Robots with Limited Visibility. IEEE Transactions on Robotics and Automation 15(5), 818–828 (1999)
4. Cieliebak, M.: Gathering Non-oblivious Mobile Robots. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 577–588. Springer, Heidelberg (2004)
5. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the Robots Gathering Problem. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1181–1196. Springer, Heidelberg (2003)
6. Cohen, R., Peleg, D.: Robot Convergence via Center-of-Gravity Algorithms. In: Kralovic, R., Sýkora, O. (eds.) SIROCCO 2004. LNCS, vol. 3104, pp. 79–88. Springer, Heidelberg (2004)
7. Cohen, R., Peleg, D.: Convergence of Autonomous Mobile Robots with Inaccurate Sensors and Movements. SIAM Journal on Computing 38(1), 276–302 (2008)
8. De Marco, G., Gargano, L., Kranakis, E., Krizanc, D., Pelc, A., Vaccaro, U.: Asynchronous deterministic rendezvous in graphs. Theoretical Computer Science 355, 315–326 (2006)

9. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)
10. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Computing without communicating: ring exploration by asynchronous oblivious robots. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 105–118. Springer, Heidelberg (2007)
11. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Remembering without Memory: Tree Exploration by Asynchronous Oblivious Robots. In: Shvartsman, A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058, pp. 33–47. Springer, Heidelberg (2008)
12. Flocchini, P., Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Multiple Mobile Agent Rendezvous in a Ring. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 599–608. Springer, Heidelberg (2004)
13. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of Asynchronous Robots with Limited Visibility. Theoretical Computer Science 337(1-3), 147–168 (2005)
14. Gąsieniec, L., Kranakis, E., Krizanc, D., Zhang, X.: Optimal Memory Rendezvous of Anonymous Mobile Agents in a Unidirectional Ring. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 282–292. Springer, Heidelberg (2006)
15. Haba, K., Izumi, T., Katayama, Y., Inuzuka, N., Wada, K.: On the Gathering Problem in a Ring for 2n Autonomous Mobile Robots, Technical Report COMP2008-30, IEICE, Japan (2008)
16. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theoretical Computer Science 390(1), 27–39 (2008)
17. Kowalski, D., Pelc, A.: Polynomial deterministic rendezvous in arbitrary graphs. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 644–656. Springer, Heidelberg (2004)
18. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
19. Prencipe, G.: CORDA: Distributed Coordination of a Set of Autonomous Mobile Robots. In: Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS), pp. 185–190 (2001)
20. Prencipe, G.: On the Feasibility of Gathering by Autonomous Mobile Robots. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 246–261. Springer, Heidelberg (2005)
21. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM Journal on Computing 28(4), 1347–1363 (1999)

Rendezvous of Mobile Agents When Tokens Fail Anytime

Shantanu Das, Matúš Mihalák, Rastislav Šrámek,
Elias Vicari*, and Peter Widmayer*

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{dass,mmihalak,rsramek,vicariel,widmayer}@inf.ethz.ch

Abstract. We consider the problem of Rendezvous or gathering of multiple autonomous entities (called mobile agents) moving in an unlabelled environment (modelled as a graph). The problem is usually solved using randomization or assuming distinct identities for the agents, such that they can execute different protocols. When the agents are all identical and deterministic, and the environment itself is symmetrical (e.g. a ring) it is difficult to break the symmetry between them unless, for example, the agents are provided with a token to mark the nodes. We consider fault-tolerant protocols for the problem where the tokens used by the agents may disappear unexpectedly. If all tokens fail, then it becomes impossible, in general, to solve the problem. However, we show that with any number of failures (less than a total collapse), we can always solve the problem if the original instance of the problem was solvable. Unlike previous solutions, we can tolerate failures occurring at arbitrary times during the execution of the algorithm. Our solution can be applied to any arbitrary network even when the topology is unknown.

Keywords: Rendezvous, Mobile Agents, Asynchronous, Anonymous Networks, Symmetry-breaking, Fault Tolerance, Faulty Token.

1 Introduction

We consider one of the fundamental problems in computing with autonomous mobile agents—the problem of gathering the agents, called *Rendezvous* [1]. The typical setting is when there is a communication network represented by a graph of n nodes and there are k mobile entities called agents, that are dispersed among the nodes of the network. The objective of the Rendezvous problem is to make all the agents gather together in a single node of the network. Solving Rendezvous is a basic step when performing some collaborative task with multiple distributed autonomous entities and it is known that many other problems such as network decontamination, intruder detection and distributed search, are easier to solve if the agents are able to rendezvous. When the nodes of the network are labelled

* The authors are partially supported by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss NSF under grant number 5005 – 67322.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 463–480, 2008.

with distinct labels from a totally ordered set, it is possible to gather at a predetermined location, for instance, the node having the smallest label. The problem is however more interesting when such unique labels are not present, i.e. when the network is anonymous. In this case, the agents have to reach an agreement among themselves about the node where to meet.

The problem of Rendezvous in an anonymous network has been studied for both synchronous [10,17,16] and asynchronous systems [9,14,15]; in this paper we focus on the more difficult case of asynchronous systems. In such systems, there is no common notion of time for the agents, and the speed of the agents traversing the network can be arbitrary. In fact, Rendezvous is not always deterministically solvable in asynchronous systems. However, Rendezvous can be achieved under certain conditions, by using a simple marking device called a *token* (also called a *pebble* or *marker*) [4,7,11,12,19]. Under this model, each agent has a token that it can put on a node to mark it. However, the tokens of all agents are identical (i.e., an agent cannot distinguish its token from a token of another agent). In [12], it was shown how to achieve Rendezvous in an anonymous ring network, when every agent puts its token on its starting location. Often it is assumed that the environment in which the agents are operating is hostile towards them i.e. it does not help in the task of achieving Rendezvous. For example, when an agent leaves a token, it is not guaranteed that the token would remain there until the agent returns (we consider asynchronous systems where there are no time bounds on the steps taken by an agent). Flocchini *et al.* [11] consider the model where some of the tokens are faulty, i.e. a token may suddenly disappear after an agent leaves it on a node. The solution given in the above paper depends on the very strong condition that the values of n and k are such $\gcd(n, k') = 1$, $\forall\ k' \leq k$. This condition was later removed in [7]. However both these solutions work only in the special case when the tokens fail either immediately on being placed or they never fail at all. In other words, they do not solve the real problem of coordination between the agents using tokens that are unreliable.

In this paper, we first show how Rendezvous can be achieved in a ring network assuming that the tokens may fail at arbitrary times during the execution of the algorithm. Our protocol solves the problem under the exact conditions that are necessary for solving Rendezvous in the fault-free scenario. Further if Rendezvous is impossible in the given instance, the agents are able to detect it and terminate within a finite time. Thus, we solve the problem of *Rendezvous with Detect.* We only require that at least one token always survives (which is a necessary condition for feasibility of Rendezvous). Our algorithm has the same asymptotic cost as the previous solutions, in terms of the number of agent moves.

For arbitrary (and unknown) networks, there were no previous solutions for Rendezvous tolerating token failures. We show that Rendezvous can be solved in general graphs tolerating up to $k - 1$ failures, for any instance where the problem is solvable in absence of failures. Thus, the conditions for solvability of Rendezvous do not depend on the occurrences of token failures (excluding the case when all tokens fail). Even for this case, our algorithm can tolerate failures occurring at arbitrary times during the execution of the algorithm.

1.1 Our Model

We consider a network consisting of n nodes which is represented by an undirected connected graph $G = (V, E)$ (we shall use the words network and graph interchangeably). There are k *mobile agents* initially located in distinct nodes of the network. The node in which an agent initially resides is called its *homebase*. The agents are allowed to move along the edges of the graph. Each such move may take a finite but arbitrarily long time (i.e. the system is strongly asynchronous). The nodes of the network are anonymous (i.e. without any identifiers). At each node v of the graph, the incident edges are locally labelled, so that an agent arriving at the node v can distinguish among them. Additionally, the agent can identify the edge through which it arrived at the node[1]. The edge labelling of the graph G is given by $\lambda = \{\lambda_v : v \in V\}$, where for each vertex v of degree d, $\lambda_v : \{e(v, u) : u \in V \text{ and } e \in E\} \rightarrow \{1, 2, ..., d\}$ is a bijection specifying the local labelling at v.

The agents are exactly identical to each other and they follow the same (deterministic) algorithm. Each agent has a token which is a simple marking device that can be released at a node in order to mark it. Any agent visiting a node would be able to "see" the token left at that node, but would not be able to identify who left that token (i.e. the tokens are also anonymous like the agents). Each agent has a finite amount of memory that can be used to carry information. Two agents can communicate (exchange information) with each other, only when they are at the same node. An agent can also detect how many agents are present at a node. However, if two agents are traversing the same edge (from the same or opposite direction), they may not communicate or even see each other. It is also possible that an agent passes another agent on the same edge (without being aware of it). Each agent knows n, the size of the network, and k, the number of present agents. The objective is to reach an agreement about a unique node in the graph where all the agents should meet. The cost of a solution in this model is measured as the total number of moves (edge traversals) performed by all the agents combined.

In our model some of the tokens may be *faulty*. A faulty token is one which disappears at some instant during the execution of the algorithm and never appears again. We make the important assumption that at most $k - 1$ tokens may fail. In other words, at least a token is always present. Otherwise we would be in the situation as when no tokens were available and Rendezvous is not deterministically possible.

1.2 Our Results

We show that it is possible to achieve Rendezvous in asynchronous anonymous environments, even if the tokens that are used to break symmetry may fail. Unlike previous attempts which solved only very restricted special cases of the

[1] This ensures that the agent do not keep moving back and forth on the same edge forever.

problem, we present solutions which work in the most general setting, i.e. in an arbitrary unknown network where tokens may fail independently and at any time. Our algorithm works for any number of failures $0 \leq f \leq k-1$. In particular, it works also for the case when $f = 0$, i.e. when no tokens fail. We note that it is easier to solve Rendezvous for either only the fault-free scenario (i.e. $f = 0$) or only the faulty cases (i.e. $f \geq 1$) [2]. However, it is difficult to construct an algorithm which works for both scenario (or, when the switch from the fault-free to the faulty case can occur anytime during the execution of the algorithm). Also note that at best such an algorithm can achieve Rendezvous for only those instances where Rendezvous is possible in the absence of failures. Not only do our algorithms achieve this, but the agents can also detect the instances where Rendezvous is impossible and thus, they can terminate explicitly under all scenarios. Such an algorithm is said to solve *Rendezvous with Detect*.

In Section 2.2 we present solutions for the ring network which is the only case that has been studied before. Our algorithm solves *Rendezvous with Detect* when n and k are known, using at most $O(k\ n)$ moves. Based on the ideas of the solution for ring networks, we also present a (more complicated) solution for the general case, when the network topology is arbitrary and unknown. For an unknown anonymous network, there are no known sub-exponential cost algorithms for even traversing the network (i.e. visiting every node). Our algorithm for solving rendezvous in this case requires $O(k\ \Delta^{2n})$ moves for graphs of maximum degree Δ.

2 Rendezvous in a Ring Network

For problems involving symmetry-breaking, the ring networks represent a topologically simple but highly symmetrical (and hence difficult to solve) instance of the problem. As it is easier to visualize the problem in a ring network, we will first consider the case of ring networks, before proposing a solution for graphs of general topology. It was shown in [7] that it is not possible to solve *Rendezvous with Detect* in an asynchronous anonymous ring in the presence of token failures, if the value of k is unknown to the agents. In this paper, we assume that the agents have prior knowledge of the values of both n and k.

2.1 Properties and Conditions

If there are no failures, then the solvability of the Rendezvous problem depends on the initial locations of the agents. We can represent the initial location of the agents in a ring of size n by the sequence S_A of inter-agent distances, starting from an arbitrary agent A, in any direction (clockwise or counterclockwise), i.e., by the sequence $S_A := (d_0, d_1, \cdots, d_{k-1})$, where d_i denotes the distance between the i-th and $(i+1)$-th agent as they appear, in the order starting from agent A, on the ring (in the clockwise or counterclockwise direction). Observe

[2] If at least one failure is guaranteed then the failures can be used to break symmetry between the agents.

that for every other agent A' and every direction (clockwise or counterclockwise), the sequence $S_{A'}$ is just a "rotation" of S_A or a "rotation" of the *reversed sequence* of S_A (where the *reversed sequence* of S_A is $(d_{k-1}, d_{k-2}, \cdots, d_1, d_0)$). For any arbitrary sequence $S = (d_0, d_1, \ldots, d_{r-1})$ of r integers, we define the following. For any i, $0 \leq i \leq r$, $Sum_i(S)$ is defined as $\sum_{j=0}^{i-1} d_j$ (i.e. the sum of the first i elements of S). The *reversal* of S is defined as the sequence $Rev(S)$ =(d_{r-1},d_{r-2},...,d_0). For any i, $0 \leq i < r$, the *i-th rotation* of S is defined as the sequence $Rot_i(S)$ =(d_i,d_{i+1},...,d_{r-1},d_0,..., d_{i-1}). A sequence S is periodic if $\exists i$, $0 < i \leq (r/2)$, such that $Rot_i(S) = S$. A sequence S is called *rotation-reversal free*, if for every i, $0 < i < r$, $Rot_i(S) \neq S$ and $Rev(Rot_i(S)) \neq S$. Observe that if S is rotation-reversal free, then also $Rot_i(S)$ and $Rev(Rot_i(S))$ are rotation-reversal free, for any i, $0 < i < r$.

Lemma 1 ([7]). *Rendezvous of k agents can be solved in a ring, in absence of failures, if and only if the sequence $S = (d_0, d_1, \ldots, d_{k-1})$ of initial inter-agent distances (starting from any agent) satisfies the following conditions:*

(i) S is rotation-reversal free, or
(ii) S is not periodic, and there exists i, $0 < i \leq k-1$, such that $S = Rev(Rot_i(S))$ and at least one of $Sum_i(S)$ and $n - Sum_i(S)$ is even.

If S =(d_0,d_1,d_2,...,d_{k-1}) is the sequence of inter-agent distances and it satisfies the conditions of Lemma 1, then we can define a unique node as the Rendezvous location, denoted by RV-point(S), in the following way[3]. If condition (i) of the above lemma holds, then there is a unique i, $0 \leq i \leq k-1$, such that either $Rot_i(S)$ or $Rev(Rot_i(S))$ is the lexicographically smallest sequence obtained by applying any number of reversions and rotations on S. In this case, we define RV-point(S) as the location of the i-th agent. Otherwise, if condition (ii) holds, then there exist unique i and j, $0 \leq i, j \leq k-1$, $i \neq j$, such that $Rot_i(S) = Rev(Rot_j(S))$ is the lexicographically smallest sequence (obtained by rotations and reversals on S). In this case, at least one of the segments (of the ring) between the i-th and the j-th agent is of even size (the *even* segment). Let u and v respectively be the homebase of the i-th and the j-th agent, and x be the node exactly in the middle of the even segment[4]. If the labelled path (using the labels of the edges) from x to u is lexicographically smaller than the path from x to v, then u is defined as RV-point(S), otherwise RV-point(S) is v. Since there is a local ordering on the edges incident at each node (in particular node x), the labels along the two paths would not be identical. We point out some important properties of the RV-point(S):

- RV-point is defined if and only if the instance of the Rendezvous problem is solvable (i.e. if and only if the conditions of Lemma 1 are met).
- For the sequence S of inter-agent distances of a solvable instance, RV-point(S) is a unique location in the ring and it is the homebase of some agent.

[3] To be precise, RV-point(S) depends on the edge-labelling of the ring as well.

[4] If both segments from u to v are even-sized, we pick the lexicographically smaller one.

A simple strategy for solving Rendezvous with Detect in the absence of failures is the following [12]. Each agent puts its token at the starting location and goes around the ring once (i.e. moves n steps in the same direction), to compute the sequences S. The agent now moves to the location defined by RV-point(S), or detects that the instance is not solvable. Notice that the agents may move in different directions and the sequences obtained by them would be distinct (but where one can be transformed into another by using rotation and/or a reversal operation), they would still gather in the same location. However, this strategy would fail in the presence of token failures (unless some strong conditions are imposed on the values of n and k). We note that the (exact) conditions for solving Rendezvous (in ring networks) in presence of arbitrary token failures were, until this paper, not known. We show that the conditions of Lemma 1 are sufficient to solve Rendezvous with Detect in a ring even in the faulty scenario when tokens may fail at arbitrary times, provided that the agents know the values of n and k (and at least one token does not fail).

2.2 Solution Protocol

We know that Rendezvous in a ring (with no faults on tokens) is possible only if the initial sequence of inter-agent distances satisfies the conditions of Lemma 1. We show that even if some of the tokens disappear, the agents are still able to reconstruct the initial sequence of inter-agent distances and thus to decide whether the instance is solvable, and to eventually meet at the Rendezvous location.

The main idea behind our solution of reconstructing the sequence of the inter-agent distances is the following. Agents whose tokens have failed (called Runners) "run" around and inform other agents about their location, while those agents whose token did not fail (called Owners), wait at the homebase to receive the "running" agent. The difficulty of this approach is when no token disappears (as then no agent is "running", and thus every agent waits in its homebase – a deadlock). On the other hand, if we know that no token disappeared, and the instance is solvable, we can elect a leader agent – the agent with its homebase as the Rendezvous location RV-point(S), who starts "running" and informs others about the Rendezvous location. Combining both cases in one algorithm is, in our opinion, the most interesting part of the solution.

More precisely, the agents do the following. In the beginning, each agent puts its token on its homebase, and walks once around the ring for n steps (thus, it comes back to its homebase), making notice of inter-agent distances as induced by the observed tokens during the walk[5]. We call this walk the *initial walk*. Comparing k with the number of found tokens during the walk (the agent considers its own token at the end of its walk), the agent knows whether some of the tokens disappeared (during the agent's walk). If this is the case, then the agent (1) either waits at its homebase, if its token is present (and stays there even if the token disappears later on – in this case the agent acts as a token and informs other

[5] The walk of each agent is determined by the first edge the agent uses when starting from its homebase, which can be e.g. the edge with the smallest label. This also determines whether the agent runs clockwise or counterclockwise.

passing-by agents about this), or (2) walks around the ring to the next token (or to an Owner who acts as a token) – the agent associates itself with the owner of this token (the agent may need to wait for the Owner which might still be on its initial walk around the ring), and starts "running" around the ring to inform every Owner about the location of the failed token. When a Runner gets back to the Owner it is associated to, it attempts to reconstruct the original sequence of inter-agent distances. If this information is complete, it goes around and informs all Owners about the meeting point (and the total number of Runners). The informed owners then subsequently inform their associated Runners about this, and walk "together" to the Rendezvous location.

In the other case, when the initial walk of an agent results in encountering all k tokens, and thus computing the correct sequence S of inter-agent distances of k agents, we have to modify the above algorithm. To ensure that even in the case when no token fails (after the initial walk of every agent), there is a special agent – a "leader" – which walks once around the ring and informs all other Owners about the situation. The special agent is the agent whose homebase is the Rendezvous location RV-point(S). If a token should fail after the election of a "leader" but before that all agents know about the Rendezvous location, an agent becomes Runner and the algorithm reverts to the aforementioned case.

Thus, the agents may have one of the following roles in our protocol. In case, the agent's own token fails during the initial walk, the agent becomes a Runner. In case the agent finds all tokens during the initial walk, and its homebase is the Rendezvous location, the agent becomes a special agent called the Leader. Runners and the Leader move around the ring collecting information and exchanging this information with Owners, leading into informing all Owners (and thus subsequently all Runners) about the (computed) Rendezvous location. An Owner is an agent that does not identify itself as a Runner or the Leader. The role of Owners is to help Runners in coordinating with each other.

We now present the complete algorithm for solving the Rendezvous problem in a ring when tokens may fail anytime (and at least one token does not fail).

Algorithm *RVring* :

```
1. Put token at starting location;
   Move $n$ steps, and compute inter-token sequence $S$ (we call it the initial
   walk);
2. If own token disappeared then become Runner;
   Else become Owner;
   If $k$ tokens were found
      If $S$ satisfies solvability condition,
         If own location is RV-point($S$) then become Leader;
      Else ($S$ does not satisfy solvability conditions)
         set Status to FAILURE;
3. A Leader agent executes the following:

   (The Leader's checking walk)
   Go around the ring, waiting at each token for the Owner to return;
```

```
If all $k-1$ Owners were found,
  (The Leader's gathering walk)
  Collect all Owners to RV-point($S$) and set Status to SUCCESS;
Else (some token disappeared)
  If own token disappeared then become Runner;
  Else become Owner;
```

4. An Owner agent simply waits at its homebase, communicates with passing agents and follows their instructions. Each Owner stores the information about its associated Runners. The Owner moves only when the Leader tells to move, or if the Rendezvous location is known, it has met every Runner in their teaching walk (the Owner counts the number of Runners that it met in the teaching walk) and all its associated Runners have been informed.
5. A Runner agent executes the Runner algorithm (explained below).

Remarks:

- After step 2 of the algorithm *RVRing*, only Leader can change its role. Especially, an Owner agent never becomes Runner even if its token disappears later on.
- If a node v contains an Owner agent then it is assumed that there is also a token at node v. This means that if a token disappears after its Owner has returned, this has no effect on the execution of the algorithm. The Owner simulates the presence of the token communicating it to any other agent that visits v.

Runner's Algorithm:

(1) *A* Runner *agent associates with exactly one* Owner *agent.*
An agent becomes a Runner only if it finds that the token at its homebase has disappeared. Such an agent moves to the next node that contains a token (or contains an Owner) and waits for the owner of that token. In case the token disappears before the owner of the token comes to the node, then the Runner moves to the next token (or Owner) and repeats the same. Once the Runner meets the awaited Owner of the token, it *associates* with this Owner. The Runner agent remembers the distance and direction from its homebase to the homebase of the associated Owner. This information is communicated to the associated Owner. Owner keeps track of the number of its associated Runners and of their homebase locations.

(2) *A* Runner *tells the other* Owners *about its homebase.*
After associating with an Owner, the Runner agent goes once around the ring (we call it the Runner*'s teaching walk*), stopping at each token, waiting for the Owner of that token and then communicating the information about the position of its homebase to that Owner.

(3) When the Runner reaches its associated Owner again, it learns the sequence S' (possibly incomplete) that the Owner has gathered so far. If S' is the correct sequence and Rendezvous is solvable for this instance, then the

Runner agent computes the RV-point and communicates it to all Owners. The Runner agent can check if the sequence is complete, i.e., if $|S'| = k$. If the sequence is complete, the agent walks around again (we call it a *Runner's informing walk*), stops at every Owner and informs the Owner about the complete sequence S' and about the number of Runners (determined by k minus the number of Owners). In this walk the Runner does not wait at unoccupied nodes with a token on it. If S' satisfies the conditions of Lemma 1 (i.e., the instance of the Rendezvous problem is solvable), the status of the Owner is set to SUCCESS, otherwise the status is set to FAILURE.

(4) A Runner agent waits at the homebase of its associated Owner until the status of this Owner is changed to SUCCESS or FAILURE, upon which the Runner learns the original sequence of inter-agent distances S. If the status is SUCCESS, it moves to the Rendezvous location RV-point(S).

2.3 Proof of Correctness

We need to show that every agent learns the correct sequence of the original inter-agent distances S, and that (in case the instance is solvable) every agent moves to the Rendezvous location RV-point(S).

We proceed with small (reassuring) observations.

1. *Every agent finishes its initial walk.* This is a rather trivial observation.
2. *If the instance is not solvable and there is no* Runner*, then every agent detects this after the initial walk.* This is again obvious, as in case there is no Runner, every agent computes the correct sequence S and thus sets its status to FAILURE.
3. *After the initial walk, every* Runner *(if there are some) associates with an* Owner*.* This follows from the assumption that there is at least one token which does not disappear. Thus, the Runner either finds an Owner immediately, or it finds a token. The token belongs to an agent in its initial walk, or to a Leader in its checking walk. In both cases the agent comes back.
4. *Every* Runner *finishes its teaching walk.* The Runner may wait for an agent to come to its token from its initial walk, but we know that every agent finishes this phase. If the token disappears before the Owner of the token returns, the Runner continues. Further, if the Runner waits at a token of the Leader, the Leader finds out that there is a Runner and thus the Leader goes back home and becomes Owner.
5. *No* Owner *leaves its homebase before meeting every* Runner*.* If the Owner moved because the Leader said so, then there is no Runner; Otherwise (see the algorithm) the Owner waits until it sees all Runners in their teaching walk. The Owner checks this by keeping track of how many Runners it met in the teaching walk, and by comparing this count with the total number of Runners (which it learns from a Runner in an informing walk). The Runners inform an Owner in which walk they are.

6. *Every* Owner *meets every* Runner *in the* Runner*'s teaching walk.* Indeed, consider any Owner and any Runner. We know that the Owner does not move if it did not meet every Runner (see previous observation). As every Runner goes for its teaching walk, it eventually meets the considered Owner.

We now prove the first lemma which helps us to reach our goal.

Lemma 2. *If there is no* Leader*, or the* Leader *did not meet* $k-1$ Owners *in its checking walk, then there is a* Runner *which is informed of the initial sequence* S *of inter-agent distances after its teaching walk.*

Proof. It should be obvious that if there is no Leader, then there exist Runners. We now show that at least one Runner obtains the original sequence S of inter-agent distances. Consider a Runner agent which was the last among all Runners to finish its teaching walk (let us call it the *last* Runner[6]). Thus, at this time, every other Runner has already communicated its homebase information. When the *last* Runner reaches the Owner which it is associated to, this Owner possesses all information about the homebases of all Runners (and knows the position of all Owners from the initial walk). Hence, the original sequence S can be reconstructed by this Runner.

Lemma 3. *Every agent learns the original sequence* S *of inter-agent distances.*

Proof. Let us first consider the situation when all agents became either Owner or Leader. In this case they all learn the correct S, as they all met all tokens during the initial walk.

Otherwise, there exists at least one Runner. Thus, by the previous lemma, there is a Runner R which learns in its teaching walk the original sequence S. Thus, this Runner R begins its informing walk, in which it informs all Owners it meets about S. We claim that after the Runner R finishes its informing walk, every Owner knows S. Indeed, when R arrives during its informing walk at a homebase of an Owner, the Owner is either present there, and thus learns S from R, or the Owner is not present there, which means that the Owner left, which is only possible if the Owner knows S. (We note that this may happen e.g. if another Runner with complete information about S already finished its informing walk, and thus left together with its associated Owner to the Rendezvous-location.)

Lemma 4. *If the instance of the Rendezvous problem is not solvable, then all agents will learn this information, and will stop moving at some time.*

Proof. By the previous lemma, every agent will learn the initial sequence S of inter-agent distances, from which it finds out that the instance of the problem is not solvable. Notice that every agent stops after walking at most three times around the ring in case that the instance is not solvable.

[6] There could possibly be multiple such agents. For the sake of arguments, we arbitrarily choose one of them.

Lemma 5. *If the instance of the Rendezvous problem is solvable, then the agents meet at the Rendezvous location RV-point(S), where S is the initial sequence of inter-agent distances.*

Proof. Due to Lemma 3 we know that every agent learns S and thus knows where is the Rendezvous location RV-point(S). We now show that every agent eventually walks to the Rendezvous location. We consider first the case when an agent becomes LEADER and in its checking walk it meets $k-1$ OWNERS. Clearly, in this case, after the LEADER finishes its gathering walk, all OWNERS walk to the Rendezvous location.

Let us now consider the case when there is no LEADER, or the LEADER does not meet $k-1$ OWNERS. In this case, there are only OWNERS and RUNNERS present (eventually, after the LEADER ends its checking walk and becomes RUNNER or OWNER). Let us consider a RUNNER agent. By Lemma 3 we know that the RUNNER learns the initial sequence S and whenever this happens the RUNNER simply walks to the Rendezvous location RV-point(S). Let us now consider an OWNER. Again by Lemma 3 we know that it learns the initial sequence S. The OWNER walks to the Rendezvous location RV-point(S) only when it knows that all RUNNERS met the OWNER in their teaching walks. As every RUNNER will eventually finish its teaching walk (see the observations above), the OWNER will have met all RUNNERS and thus walks to the Rendezvous location RV-point(S).

Theorem 1. *The algorithm* RVring *solves the Rendezvous problem with detect whenever there are at most $k-1$ token failures, where k is the number of agents. The total number of moves made by the agents is $O(k\ n)$ and the memory requirement for each agent is $O(k \log n)$.*

Proof. Due to Lemmas 4 and 5, we know that the algorithm always terminates explicitly. Further, whenever Rendezvous is solvable, the algorithm achieves Rendezvous of k agents and otherwise detects unsolvability. This proves correctness of the algorithm. To bound the number of total movements of the agent, observe that in the algorithm *RVring* each agent (RUNNER, LEADER or OWNER) performs a constant number of traversals of the ring (at most five in case of a RUNNER – the initial walk, the association of RUNNERS to OWNERS, the checking walk, the informing walk, and the walk to the Rendezvous location) where each traversal takes exactly n moves. The memory required to store the sequence S' of inter-token distances is $O(g \log n)$ where $g \leq k$ is the number of OWNER agents.

Our algorithm compares favorably with previous results which solve the Rendezvous problem in fault-free scenario (or the restricted faults scenario) with the same cost and the same memory requirements. In fact, it is easy to show that any algorithm for solving *Rendezvous with Detect* requires $\Omega(k\ n)$ moves in the worst case.

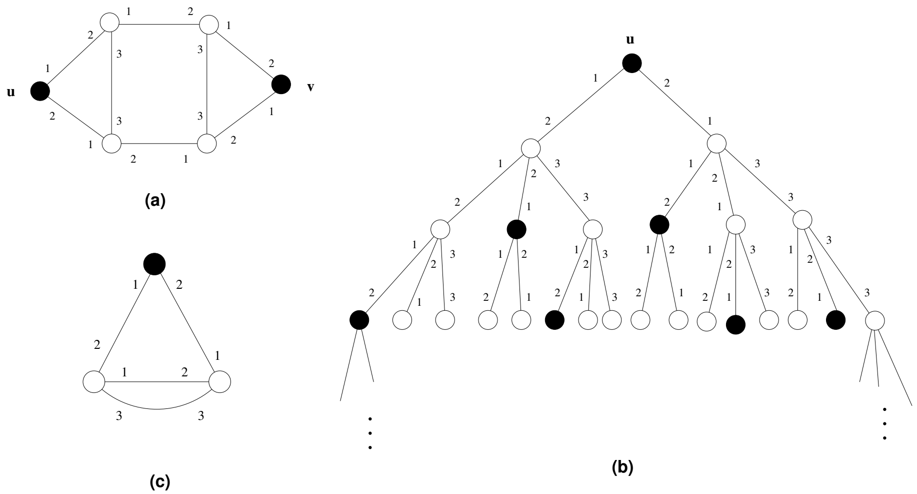


Fig. 1. (a) A network with $n = 6$ and two agents at nodes u and v. (b) The bi-colored view of node u. (c) The quotient graph of the network.

3 Rendezvous in Arbitrary Networks

3.1 Conditions for Solving Rendezvous

In the general case of arbitrary graphs, the initial location of the agents in the graph $G(V, E)$ is specified using a placement function $p : V \rightarrow \{0, 1\}$, where $p(v) = 1$ if the node v is a homebase of an agent and $p(v) = 0$ otherwise[7]. The function p is a bi-coloring on the set of vertices of G (We say that a vertex v is black if $p(v) = 1$ and white if $p(v) = 0$). An instance of the Rendezvous problem is defined by the tuple (G, λ, p) where λ is the edge labelling on the graph G and p defines the initial location of agents in G. The conditions for solving Rendezvous in arbitrary graphs are related to the conditions for leader election in the anonymous message-passing network model, which has been studied extensively [2,5,20]. The concept of a *view* of a node in a network was introduced by Yamashita and Kameda [20] with respect to the symmetry-breaking problem in message passing systems. The *view* of a node v in the labelled graph (G, λ) is an infinite edge-labelled rooted tree that contains all (infinite) walks starting at v. For the Mobile Agent model, the concept of view can be extended to that of a bi-colored view, where the vertices in the view are colored black or white depending on whether or not they represent the homebase of some agent (recognizable by the presence or not of a token).

In the network represented by (G, λ, p), the *bi-colored view* of a node v is an infinite edge-labelled rooted tree T whose root represents the node v, and for each neighboring node u_i of v in G, there is a vertex x_i in T (colored as u_i) and

[7] Recall that the agents start from distinct nodes, so each node contains at most one agent initially.

an edge from the root to x_i with the same label as the edge from v to u_i in G. The subtree of T rooted at x_i is (recursively) the view of the node u_i.
The following result is known for Rendezvous of agents in arbitrary graphs:

Lemma 6 ([2,3,20]). *When the agents do not have a-priori knowledge of the network topology, the Rendezvous problem is solvable in an arbitrary network (G, λ, p) if and only if each node in G has a distinct bi-colored view.*

Notice that the view of a node $u \in G$ contains multiple occurrences of each vertex in G (see Figure 1). In fact the view of a node u truncated to depth $2n-2$ (denoted by T_u^{2n-2}) contains the view of any other node up to depth of at least $n-1$. From the results of Norris [18], it is known that the views of two nodes are identical if and only if their views up to depth $n-1$ are identical. Thus, from the view T_u^{2n-2} of any node u, it is possible to obtain a so-called *closed view* where vertices with identical views are merged into a single vertex. The (multi)-graph H thus obtained (which may contain self-loops and parallel edges), is called the quotient-graph [20] of the network. In the notion of *graph coverings*, one can say the graph G covers the graph H and this covering preserves the labelling of the edges and the bi-coloring on the vertices. If the view of each node is distinct, then the quotient graph H is isomorphic to G and we say that (G, λ, p) is covering minimal [5]. Hence, an alternate (but equivalent) condition for solving the Rendezvous problem is that the network is covering minimal with respect to coverings that preserve the edge-labelling and the vertex-coloring. In this case, we can define a unique rendezvous location in G by ordering the bi-colored views (according to some fixed total order) and choosing the node whose bi-colored view is the smallest.

3.2 Rendezvous Algorithm for Unknown Graphs

In the case of an unknown arbitrary graph, we have the following strategy for solving Rendezvous. Each agent can perform a (blind) depth-first traversal up to depth $2n-2$ to construct the bi-colored view. (This is the initial walk of the agent.) It is possible that tokens may disappear during the traversal. So, if a token at a node v disappears between two consecutive visits to v by the agent, then the agent would obtain an inconsistent coloring of the view (without being able to detect this inconsistency by just looking at the view). We say that a bi-coloring of a view is *inconsistent* if multiple occurrences of the same vertex in the view are colored differently. However, it is easy to check for consistency by simply repeating the view computation one more time and then comparing the results.[8] If the view is inconsistent, then we know there must be at least one token failure, and in that case the agent does not need to construct the correct bi-colored view in its initial walk (as explained below). Otherwise, given the view of the network and any consistent bi-coloring of the view, each agent can compute the closed view (H, λ_H, P_H), where H is a multi-graph, λ_H is an edge labelling

[8] If there is any inconsistency due to disappearance of tokens, then the bi-colored views obtained in two consecutive computations would be distinct.

and P_H is a vertex bi-coloring of H. We define $k' = |\{v \in H : P_H(v) = 1\}|$ as the agent-count of H. Observe the following:

- The labelled graph (G, λ, p) covers the labelled multigraph (H, λ_H, P_H). Thus $|H|$ divides $|G|$.
- The number of tokens detected is $k' \times (n/|H|)$. Thus, $k' \times (n/|H|) \leq k$. If P_H is the correct bi-coloring then $k/k' = n/|H|$. Thus, an agent can determine whether the bi-coloring that it computed is the correct bi-coloring or not.
- If P_H is the correct bi-coloring and Rendezvous is solvable in (G, λ, p), then H is isomorphic to G and $k' = k$. This implies that if an agent obtains the correct bi-coloring, then it can detect whether Rendezvous is solvable or not. Further, if Rendezvous is solvable, the agent can traverse G using the map H and it can also determine the unique Rendezvous location as defined earlier.

We now show how to extend the algorithm for rings to obtain a Rendezvous algorithm for arbitrary graphs, using the above ideas.

Each agent computes the bi-colored view in the first round of the algorithm. As mentioned before an agent needs to perform the view computation twice to check for consistency. In case the results of the two computations differ, then the agent knows that some token must have failed and thus it returns to its homebase and becomes either Owner or Runner (depending on whether its own token is present). On the other hand, if no tokens fail, then the computed bi-coloring would be the correct one and we proceed as in the previous algorithm, by electing a Leader agent. In case of token failures, there would exist some Runner agents and each Runner would try to compute the correct bi-colored view. This done in two rounds as follows. First each Runner traverses the view and communicates to each Owner the (labelled) path it traversed to reach this Owner. In the second round the Runner traverses the view again to obtain information from each Owner (about the paths leading to other Runners). This information can be used to obtain the bi-coloring of the view. As before, we can show that at least one Runner would compute the correct bi-coloring. In the case when Rendezvous is solvable, this agent can build a map of the graph and determine the unique Rendezvous location; Thus, we can proceed in a manner similar to the previous algorithm. In the other case, when Rendezvous is not solvable, this agent can detect this and inform each Owner about the impossibility of Rendezvous. The new algorithm, called *RV-General* is described below.

Algorithm *RV-general* :

```
1. Put token at starting location ;
   Construct bi-colored view up to depth $2n - 2$;
   Check for consistency of the bi-coloring;

2. If own Token disappeared then become Runner;
   Else become Owner;
      If the bi-colored view is consistent,
              Construct the quotient-graph $H$
```

```
    (let $n'$ be the size of $H$ and $k'$ be the number of black nodes);
    If $n' = n$ and $k' = k$,
      If own location is RV-location($H$) then become Leader;
    Else if $n/n' = k/k'$ declare Failure; (set Status to FAILURE)
```

3. A Leader agent executes the following:

```
Traverse $H$, waiting at each token (for the Owner to return);
If all $k - 1$ Owners were found,
  Collect all Owners to RV-location($H$) and set Status to SUCCESS;
Else (some token disappeared)
    If own Token disappeared then become Runner;
    Else become Owner;
```

4. An Owner agent simply waits at its homebase, communicates with passing agents and follows their instructions. Each Owner stores (communicates) the information about paths to Runners. The Owner moves only when the Leader tells to move, or if the Rendezvous location is known, it has met every Runner in their teaching walk (the Owner counts the number of Runners that it met in their teaching walk and compares this with the Runner-Number) and all its associated Runners have been informed.
5. A Runner agent executes the Runner algorithm (explained below).

Runner's Algorithm:

(1) A Runner agent perform a depth-first traversal of the view, and it associates with the first Owner agent that it meets (as before, the Runner waits at each token for the Owner to return). During the depth-first traversal, the Runner agent remembers the labelled path it traversed from its homebase to the current node and if the current node has a token, the agent communicates the path information to the Owner of this token. This is the Runner's *teaching walk*.

(2) A Runner agent, after completing its *teaching walk*, returns to its associated Owner and collects information from the Owner about paths to other Runners. Based on this information, it computes a bi-coloring of the view and builds the quotient graph H. Note that this may not be the correct quotient-graph of G (due to missing information about some Runners).

(3) Let n' be the size of H and k' be the number of black nodes in H. If $n' = n$ and $k' = k$, then H is identical to G and this represents a solvable instance of the Rendezvous problem. In this case, the Runner computes the RV-location and then traverses the graph G, stopping at each Owner to communicate the RV-location and the Runner-Number (This is simply a count of the number of vertices in the bi-colored view of this Owner, which corresponds to some Runner homebase). The Runner also changes the status of each Owner to SUCCESS.
Otherwise, if $H < G$, but $n/n' = k/k'$, then this represents an unsolvable instance of the Rendezvous problem. In this case, the Runner traverses a

spanning tree of H and changes the status of each OWNER (that it meets) to FAILURE.

(4) A RUNNER agent waits at the homebase of its associated OWNER until the status of this OWNER is changed to SUCCESS or FAILURE. If the status is SUCCESS, it moves to the Rendezvous location RV-point(S).

The correctness of algorithm *RV-General* can be proved in a similar manner as for the previous algorithm. We first show that the agents are able to compute the original bi-coloring and thus construct the minimum-base of the network.

Lemma 7. *Every* OWNER *agent is able to obtain the minimum-base of the network. Further if there are some* RUNNERS *then at least one* RUNNER *also obtains this information.*

Proof. We first show that each OWNER obtains the correct bi-coloring of the view. If there are no failures, then the result holds trivially, so we consider only the case when some tokens failed. Consider an OWNER agent A sitting at its homebase node v. If there is a path p (of length less than $2n - 2$) from some node x to v, and there was a token on node x that failed, then there must be a RUNNER agent corresponding to this failed token and this RUNNER would traverse all paths (of length at most $2n - 2$). In particular, this RUNNER would traverse path p and reach node v. Thus, for each path starting at node v, agent A could determine if the end point of the path is colored black or white. Notice that an OWNER waits at its homebase until it is visited by every RUNNER during its teaching walk. Thus, once every RUNNER has completed its traversal of the view, every OWNER would obtain the correct bi-coloring of the view. Consider now the last RUNNER to complete its traversal of the view. When this RUNNER returns to its associated OWNER, the OWNER would at that time have obtained the complete bi-colored view. Thus, this RUNNER would be able to construct the minimum-base of the network.

From the previous discussion and due to Lemma 7, the following result holds.

Theorem 2. *Algorithm* RV-General *solves the Rendezvous in any arbitrary network (G, λ, p) whenever the conditions of Lemma 6 are satisfied. Otherwise, every agent terminates with FAILURE.*

Proof. If there is a LEADER and $k - 1$ OWNERS, then the LEADER informs every OWNER about the Rendezvous-location and moves them to the Rendezvous-location. Thus, the theorem holds. Consider now the situation when there some RUNNERS (this implies that there is no LEADER). Due to the previous lemma, at least one RUNNER constructs the minimum-base H of the network. If the conditions of Lemma 6 are satisfied then H is a map of the network and there is a unique Rendezvous-location. The RUNNER traverses the network and changes the status of every OWNER to SUCCESS (and informs them about the Rendezvous-location). Thus, every RUNNER learns about the Rendezvous-location from its associated OWNER and moves to the Rendezvous-location. An OWNER moves to the Rendezvous-location after all its associated Runners have learned the Rendezvous-location. Thus, all agents meet at the Rendezvous-location. In the other case when the conditions of Lemma 6 are not satisfied, at least one Runner learns about this (after computing the minimum-base). This Runner changes the status of every Owner to FAILURE. Every Runner eventually returns to its associated Owner and waits until the status is changed. Thus, every agent terminates with status FAILURE.

Theorem 3. *During the algorithm* RV-General*, the agent performs $O(k\Delta^{2n})$ moves in total, where Δ is the maximum degree of any node in G.*

Proof. Each view computation requires $O(\Delta^{2n})$ agent moves. Each agent initially performs view-computation two times. Every Runner agent performs two more traversals of the view T. Other than that each agent performs a constant number of traversals of the quotient graph H which has size at most n.

4 Conclusions

We presented algorithms for solving Rendezvous with unreliable tokens in a strongly asynchronous environment, when the tokens may fail at any arbitrary time. As long as at least one token remains, we were able to solve the problem under the same conditions as required for Rendezvous without failures. Thus, our results show that the occurrence of $f < k$ faults has no effect on the solvability of the Rendezvous problem. Our algorithm for the case of ring networks requires $O(k\ n)$ agent moves and has therefore the same asymptotic cost as in the absence of failures. Our solution can also be applied to the more general case of an unknown graph, albeit with an exponential cost in terms of agent moves. This cost is same as that of the known solution for the fault-free case [8]. Improving this cost involves finding a more efficient way of traversing and exploring an unknown unlabelled graph.

References

1. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous. Kluwer, Dordrecht (2003)
2. Angluin, D.: Local and global properties in networks of processors. In: Proc. of 12th Symposium on Theory of Computing (STOC 1980), pp. 82–93 (1980)
3. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Can we elect if we cannot compare? In: Proc. Fifteenth Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA 2003), pp. 324–332 (2003)
4. Baston, V., Gal, S.: Rendezvous search when marks are left at the starting points. Naval Research Logistics 38, 469–494 (1991)
5. Boldi, P., Shammah, S., Vigna, S., Codenotti, B., Gemmell, P., Simon, J.: Symmetry breaking in anonymous networks: Characterizations. In: Proc. 4th Israel Symp. on Th. of Comput. and Syst., pp. 16–26 (1996)

6. Chalopin, J., Das, S., Santoro, N.: Rendezvous of Mobile Agents in Unknown Graphs with Faulty Links. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 108–122. Springer, Heidelberg (2007)
7. Das, S.: Mobile Agent Rendezvous in a Ring using Faulty Tokens. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 292–297. Springer, Heidelberg (2008)
8. Das, S., Flocchini, P., Nayak, A., Santoro, N.: Effective elections for anonymous mobile agents. In: Asano, T. (ed.) ISAAC 2006. LNCS, vol. 4288, pp. 732–743. Springer, Heidelberg (2006)
9. De Marco, G., Gargano, L., Kranakis, E., Krizanc, D., Pelc, A., Vaccaro, U.: Asynchronous deterministic rendezvous in graphs. Theor. Comput. Sci. 355(3), 315–326 (2006)
10. Dessmark, A., Fraigniaud, P., Kowalski, D.R., Pelc, A.: Deterministic Rendezvous in Graphs. Algorithmica 46(1), 69–96 (2006)
11. Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F.L., Santoro, N., Sawchuk, C.: Mobile Agents Rendezvous When Tokens Fail. In: Kralovic, R., Sýkora, O. (eds.) SIROCCO 2004. LNCS, vol. 3104, pp. 161–172. Springer, Heidelberg (2004)
12. Flocchini, P., Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Multiple mobile agent rendezvous in a ring. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 599–608. Springer, Heidelberg (2004)
13. Flocchini, P., Roncato, A., Santoro, N.: Computing on anonymous networks with sense of direction. Theor. Comput. Sci. 301, 1–3 (2003)
14. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theor. Comput. Sci. 390(1), 27–39 (2008)
15. Kosowski, A., Klasing, R., Navarra, A.: Taking Advantage of Symmetries: Gathering of Asynchronous Oblivious Robots on a Ring. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 446–462. Springer, Heidelberg (2008)
16. Kowalski, D.R., Pelc, A.: Polynomial Deterministic Rendezvous in Arbitrary Graphs. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 644–656. Springer, Heidelberg (2004)
17. Kranakis, E., Krizanc, D., Markou, E.: Mobile Agent Rendezvous in a Synchronous Torus. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 653–664. Springer, Heidelberg (2006)
18. Norris, N.: Universal covers of graphs: isomorphism to depth $n-1$ implies isomorphism to all depths. Discrete Applied Math 56, 61–74 (1995)
19. Sawchuk, C.: Mobile agent rendezvous in the ring. PhD Thesis, Carleton University (2004)
20. Yamashita, M., Kameda, T.: Computing on anonymous networks: Parts I and II. IEEE Trans. on Parallel and Distributed Systems 7(1), 69–96 (1996)

Solving Atomic Multicast When Groups Crash*

Nicolas Schiper and Fernando Pedone

University of Lugano, Switzerland

Abstract. In this paper, we study the atomic multicast problem, a fundamental abstraction for building fault-tolerant systems. In our model, processes are divided into non-empty and disjoint *groups*. Multicast messages may be addressed to any subset of groups, each message possibly being multicast to a different subset. Several papers previously studied this problem either in local area networks [1,2,3] or wide area networks [4,5]. However, none of them considered atomic multicast when groups may crash. We present two atomic multicast algorithms that tolerate the crash of groups. The first algorithm tolerates an arbitrary number of failures, is genuine (i.e., to deliver a message m, only addressees of m are involved in the protocol), and uses the perfect failures detector $\mathcal{P}$. We show that among realistic failure detectors, i.e., those that do not predict the future, $\mathcal{P}$ is necessary to solve genuine atomic multicast if we do not bound the number of processes that may fail. Thus, $\mathcal{P}$ is the *weakest* realistic failure detector for solving genuine atomic multicast when an arbitrary number of processes may crash. Our second algorithm is non-genuine and less resilient to process failures than the first algorithm but has several advantages: (i) it requires perfect failure detection within groups only, and not across the system, (ii) as we show in the paper it can be modified to rely on unreliable failure detection at the cost of a weaker liveness guarantee, and (iii) it is fast, messages addressed to multiple groups may be delivered within two inter-group message delays only.

1 Introduction

Mission-critical distributed applications typically replicate data in different data centers. These data centers are spread over a large geographical area to provide maximum availability despite natural disasters. Each data center, or *group*, may host a large number of processes connected through a fast local network; a few groups exist, interconnected through high-latency communication links. Application data is replicated locally, for high availability despite the crash of processes in a group, and globally, for locality of access and high availability despite the crash of an entire group.

Atomic multicast is a communication primitive that offers adequate properties, namely agreement on the set of messages delivered and on their delivery order, to implement partial data replication [6,7]. As opposed to atomic broadcast [8], atomic multicast allows messages to be addressed to any subset of the groups in the system. For efficiency purposes, multicast protocols should be *genuine* [9], i.e., only the addressees

* The work presented in this paper has been partially funded by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 481–495, 2008.

of some message m should participate in the protocol to deliver m. This property rules out the trivial reduction of atomic multicast to atomic broadcast where every message m is broadcast to all groups in the system and only delivered by the addressees of m.

Previous work on atomic multicast [1,3,2,4,5] all assume that, inside each group, there exists at least one non-faulty process. We here do not make this assumption and allow groups to entirely crash. To the best of our knowledge, this is the first paper to investigate atomic multicast in such a scenario.

The atomic multicast algorithms we present in this paper use oracles that provide possibly inaccurate information about process failures, i.e., failure detectors [10]. Failure detectors are defined by the properties they guarantee on the set of trusted (or suspected) processes they output. Ideally, we would like to find the *weakest* failure detector $\mathcal{D}_{amcast}$ for genuine atomic multicast. Intuitively, $\mathcal{D}_{amcast}$ provides just enough information about process failures to solve genuine atomic multicast but not more. More formally, a failure detector $\mathcal{D}_1$ is at least as strong as a failure detector $\mathcal{D}_2$, denoted as $\mathcal{D}_1 \succeq \mathcal{D}_2$, if and only if there exists an algorithm that implements $\mathcal{D}_2$ using $\mathcal{D}_1$, i.e., the algorithm emulates the output of $\mathcal{D}_2$ using $\mathcal{D}_1$. $\mathcal{D}_{amcast}$ is the weakest failure detector for genuine atomic multicast if two conditions are met: we can use $\mathcal{D}_{amcast}$ to solve genuine atomic multicast (sufficiency) and any failure detector $\mathcal{D}$ that can be used to solve genuine atomic multicast is at least as strong as $\mathcal{D}_{amcast}$, i.e., $\mathcal{D} \succeq \mathcal{D}_{amcast}$ (necessity) [11].

We here consider *realistic* failure detectors only, i.e., those that cannot predict the future [12]. Moreover, we do not assume any bound on the number of processes that can crash. In this context, Delporte *et al.* showed in [12] that the weakest failure detector $\mathcal{D}_{cons}$ for consensus may not make any mistakes about the alive status of processes, i.e., it may not stop trusting a process before it crashes.[1] Additionally, $\mathcal{D}_{cons}$ must eventually stop trusting all crashed processes. In the literature, $\mathcal{D}_{cons}$ is denoted as the perfect failure detector $\mathcal{P}$. Obviously, atomic multicast allows to solve consensus: every process atomically multicasts its proposal; the decision of consensus is the first delivered message. Hence, the weakest realistic failure detector to solve genuine atomic multicast $\mathcal{D}_{amcast}$ when the number of faulty processes is not bounded is at least as strong as $\mathcal{P}$, i.e., $\mathcal{D}_{amcast} \succeq \mathcal{P}$. We show that $\mathcal{P}$ is in fact the weakest realistic failure detector for genuine atomic multicast when an arbitrary number of processes may fail by presenting an algorithm that solves the problem using perfect failure detection.

As implementing $\mathcal{P}$ seems hard, if not impossible, in certain settings (e.g., wide area networks), we revisit the problem from a different angle: we consider non-genuine atomic multicast algorithms. For this purpose, as noted above, atomic broadcast could be used. This solution, however, is of little practical interest as delivering messages requires all processes to communicate, even for messages multicast to a single group. The second algorithm we present does not suffer from this problem: messages multicast to a single group g may be delivered without communication between processes outside g. Moreover, our second algorithm offers some advantages when compared to our first algorithm, based on $\mathcal{P}$: Wide-area communication links are used sparingly, messages addressed to multiple groups can be delivered within two inter-group message delays,

[1] Intuitively, consensus allows each process to propose a value and guarantees that processes eventually decide on one common value.

and perfect failure detection is only required within groups and not across the system. Although this assumption is more reasonable than implementing $\mathcal{P}$ in a wide area network, it may still be too strong for some systems. Thus, we discuss a modification to the algorithm that tolerates unreliable failure detection, at the cost of a weaker liveness guarantee. The price to pay for the valuable features of this second algorithm is a lower process failure resiliency: group crashes are still tolerated provided that *enough* processes in the whole system are correct.

Contribution. In this paper, we make the following contributions. We present two atomic multicast algorithms that tolerate group crashes. The first algorithm is genuine, tolerates an arbitrary number of failures, and requires perfect failure detection. The second algorithm is non-genuine but only requires perfect failure detection inside each group and may deliver messages addressed to multiple groups in two inter-group message delays. We present a modification to the algorithm to cope with unreliable failure detection.

Road map. The rest of the paper is structured as follows. Section 2 reviews the related work. In Section 3 our system model and definitions are introduced. Sections 4 and 5 present the two atomic multicast algorithms. Finally, Section 6 concludes the paper. The proof of correctness of the algorithms can be found in [13].

2 Related Work

The literature on atomic broadcast and multicast algorithms is abundant [14]. We briefly review some of the relevant papers on atomic multicast.

In [9], the authors show the impossibility of solving genuine atomic multicast with unreliable failure detectors when groups are allowed to intersect. Hence, the algorithms cited below consider non-intersecting groups. Moreover, they all assume that groups do not crash, i.e., there exists at least one correct process inside each group.

These algorithms can be viewed as variations of Skeen's algorithm [1], a multicast algorithm designed for failure-free systems, where messages are associated with timestamps and the message delivery follows the timestamp order. In [3], the addressees of a message m, i.e., the processes to which m is multicast, exchange the timestamp they assigned to m, and, once they receive this timestamp from a majority of processes of each group, they propose the maximum value received to consensus. Because consensus is run among the addressees of a message and can thus span multiple groups, this algorithm is not well-suited for wide area networks. In [2], consensus is run inside groups exclusively. Consider a message m that is multicast to groups $g_1, ..., g_k$. The first destination group of m, g_1, runs consensus to define the final timestamp of m and hands over this message to group g_2. Every subsequent group proceeds similarly up to g_k. To ensure agreement on the message delivery order, before handling other messages, every group waits for a final acknowledgment from group g_k. In [4], inside each group g, processes implement a logical *clock* that is used to generate timestamps, this is g's clock (consensus is used among processes in g to maintain g's clock). Every multicast message m goes through four stages. In the first stage, in every group g addressed by m, processes define a timestamp for m using g's clock. This is g's proposal for m's final timestamp. Groups then exchange their proposals and set m's final timestamp to the maximum among all proposals. In the last two stages, the clock of g is updated to a value bigger than m's final timestamp and m is delivered when its timestamp is the smallest among all messages that are in one of the four stages. In [5], the authors present an optimization of [4] that allows messages to skip the second and third stages in certain conditions, therefore sparing the execution of consensus instances. The algorithms of [4,5] can deliver messages in two inter-group message delays; [5] shows that this is optimal.

To the best of our knowledge, this is the first paper that investigates the solvability of atomic multicast when groups may entirely crash. Two algorithms are presented: the first one is genuine but requires system-wide perfect failure detection. The second algorithms is not genuine but only requires perfect failure detection inside groups.

3 Problem Definition

3.1 System Model

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes which communicate through message passing and do not have access to a shared memory or a global clock. Processes may however access failure detectors [10]. We assume the benign crash-stop failure model: processes may fail by crashing, but do not behave maliciously. A process that never crashes is $correct$; otherwise it is $faulty$. The maximum number of processes that may crash is denoted by f. The system is asynchronous, i.e., messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds. Furthermore, the communication links do not corrupt nor duplicate messages, and are quasi-reliable: if a correct process p sends a message m to a correct process q, then q eventually receives m. We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For each process $p \in \Pi$, $group(p)$ identifies the group p belongs to. A group g that contains at least one correct process is correct; otherwise g is faulty.

3.2 Atomic Multicast

Atomic multicast allows messages to be A-MCast to any subset of groups in Γ. For every message m, $m.dst$ denotes the groups to which m is multicast. Let p be a process. By abuse of notation, we write $p \in m.dst$ instead of $\exists g \in \Gamma : g \in m.dst \wedge p \in g$. Atomic multicast is defined by the primitives A-MCast and A-Deliver and satisfies the following properties: (i) *uniform integrity:* For any process p and any message m, p A-Delivers m at most once, and only if $p \in m.dst$ and m was previously A-MCast, (ii) *validity:* if a correct process p A-MCasts a message m, then eventually all correct processes $q \in m.dst$ A-Deliver m, (iii) *uniform agreement:* if a process p A-Delivers a message m, then all correct processes $q \in m.dst$ eventually A-Deliver m, and (iv) *uniform prefix order:* for any two messages m and m' and any two processes p and q such that $\{p, q\} \in m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.

Let $\mathcal{A}$ be an algorithm solving atomic multicast. We define $\mathcal{R}(\mathcal{A})$ as the set of all admissible runs of $\mathcal{A}$. We require atomic multicast algorithms to be *genuine* [9]:

- *Genuineness*: An algorithm $\mathcal{A}$ solving atomic multicast is said to be *genuine* iff for any run $R \in \mathcal{R}(\mathcal{A})$ and for any process p, in R, if p sends or receives a message then some message m is A-MCast and either p is the process that A-MCasts m or $p \in m.dst$.

4 Solving Atomic Multicast with a Perfect Failure Detector

In this section, we present the first genuine atomic multicast algorithm that tolerates an arbitrary number of process failures, i.e., $f \leq n$. We first define additional abstractions used in the algorithm, then explain the mechanisms to ensure agreement on the delivery order, and finally, we present the algorithm itself.

4.1 Additional Definitions and Assumptions

Failure Detector $\mathcal{P}$. We assume that processes have access to the perfect failure detector $\mathcal{P}$ [10]. This failure detector outputs a list of trusted processes and satisfies the following properties[2]: (i) *strong completeness:* eventually no faulty process is ever trusted by any correct process and (ii) *strong accuracy:* no process stops being trusted before it crashes.

Causal Multicast. The algorithm we present below uses a *causal multicast* abstraction. Causal multicast is defined by primitives *C-MCast(m)* and *C-Deliver(m)*, and satisfies the uniform integrity, validity, and uniform agreement properties of atomic multicast as well as the following *uniform causal order* property: for any messages m and m', if C-MCast(m) → C-MCast(m'), then no process $p \in m.dst \cap m'.dst$ C-Delivers m' unless it has previously C-Delivered m.[3] To the best of our knowledge, no algorithm implementing this specification of causal multicast exists. We thus present a genuine causal multicast algorithm that tolerates an arbitrary number of failures in [13].[4]

Global Data Computation. We also assume the existence of a *global data computation* abstraction [16]. The global data computation problem consists in providing each process with the same vector V, with one entry per process, such that each entry is filled with a value provided by the corresponding process. Global data computation is defined by the primitives propose(v) and decide(V) and satisfies the following properties: (i) *uniform validity:* if a process p decides V, then $\forall q : V[q] \in \{v_q, \bot\}$, where v_q is q's proposal, (ii) *termination:* if every correct process proposes a value, then every correct

[2] Historically, $\mathcal{P}$ was defined to output a set of suspected processes. We here define its output as a set of trusted processes, i.e., in our definition the output corresponds to the complement of the output in the original definition.

[3] The relation → is Lamport's transitive happened before relation on events [15]. Here, events can be of two types, C-MCast or C-Deliver. The relation is defined as follows: $e_1 \rightarrow e_2 \Leftrightarrow e_1, e_2$ are two events on the same process and e_1 happens before e_2 or $e_1 =$ C-MCast(m) and $e_2 =$ C-Deliver(m) for some message m.

[4] The genuineness of causal multicast is defined in a similar way as for atomic multicast.

process eventually decides one vector, (iii) *uniform agreement:* if a process p decides V, then all correct processes q eventually decide V, and (iv) *uniform obligation:* if a process p decides V, then $V[p] = v_p$. An algorithm that solves global data computation using the perfect failure detector $\mathcal{P}$ appears in [16]. This algorithm tolerates an arbitrary number of failures.

4.2 Agreeing on the Delivery Order

The algorithm associates every multicast message with a timestamp. To guarantee agreement on the message delivery order, two properties are ensured: (1) processes agree on the message timestamps and (2) after a process p A-Delivers a message with timestamp ts, p does not A-Deliver a message with a smaller timestamp than ts. These properties are implemented as described next.

For simplicity, we initially assume a multicast primitive that guarantees agreement on the set of messages processes deliver, but not causal order; we then show how this algorithm may incur into problems, which can be solved using causal multicast. To A-MCast a message m_1, m_1 is thus first multicast to the addressees of m_1. Upon delivery of m_1, every process p uses a local variable, denoted as TS_p, to define its proposal for m_1's timestamp, $m_1.ts_p$. Process p then proposes $m_1.ts_p$ in m_1's global data computation (gdc) instance. The definitive timestamp of m_1, $m_1.ts^{def}$, is the maximum value of the decided vector V. Finally, p sets TS_p to a bigger value than $m_1.ts^{def}$ and A-Delivers m_1 when all *pending* messages have a bigger timestamp than $m_1.ts^{def}$—a message m is pending if p delivered m but did not A-Deliver m yet.

Although this reasoning ensures that processes agree on the message delivery order, the delivery sequence of faulty processes may contain *holes*. For instance, p may A-Deliver m_1 followed by m_2, while some faulty process q only A-Delivers m_2. To see why, consider the following scenario. Process p delivers m_1 and m_2, and proposes some timestamp ts_p for these two messages. As q is faulty, it may only deliver m_2 and propose some timestamp ts_q bigger than ts_p as m_2's timestamp—this is possible because q may have A-Delivered several messages before m_2 that were not addressed to p and q thus updated its TS variable. Right after deciding in m_2's gdc instance, q A-Delivers m_2 and crashes. Later, p decides in m_1 and m_2's gdc instances, and A-Delivers m_1 followed by m_2, as m_1's definitive timestamp is smaller than m_2's.

To solve this problem, before A-Delivering a message m, every process p addressed by m computes m's *potential predecessor set*, denoted as $m.pps$. This set contains all messages addressed to p that may potentially have a smaller definitive timestamp than m's (in the example above, m_1 belongs to $m_2.pps$).[5] Message m is then A-Delivered when for all messages m' in $m.pps$ either (a) $m'.ts^{def}$ is known and it is bigger than $m.ts^{def}$ or (b) m' has been A-Delivered already.

The potential predecessor set of m is computed using causal multicast: To A-MCast m, m is first causally multicast. Second, after p decides in m's instance and updates its TS variable, p causally multicasts an *ack* message to the destination processes of m. As

[5] Note that the idea of computing a message's potential predecessor set appears in the atomic multicast algorithm of [3]. However, this algorithm assumes a majority of correct processes in every group and thus computes this set differently.

soon as p receives an *ack* message from all processes addressed by m that are trusted by its perfect failure detector module, the potential predecessor set of m is simply the set of pending messages.

Intuitively, m's potential predecessor set is correctly constructed for the two following facts: (1) Any message m', addressed to p and some process q, that q causally delivers *before* multicasting m's *ack* message will be in $m.pps$ (the definitive timestamp of m' might be smaller than m's). (2) Any message causally delivered by some addressee q of m *after* multicasting m's *ack* message will have a bigger definitive timestamp than m's. Fact (1) holds from causal order, i.e., if q C-Delivers m' before multicasting m's *ack* message, then p C-Delivers m' before C-Delivering m's *ack*. Fact (2) is a consequence of the following. As p's failure detector module is perfect, p stops waiting for *ack* messages as soon as p received an *ack* from all *alive* addressees of m. Hence, since processes update their TS variable after deciding in m's global data computation instance but before multicasting the *ack* message of m, no addressee of m proposes a timestamp smaller than $m.ts^{def}$ *after* multicasting m's *ack* message.

4.3 The Algorithm

Algorithm $\mathcal{A}1$ is composed of four tasks. Each line of the algorithm, task 2, and the procedure ADeliveryTest are executed atomically. Messages are composed of application data plus four fields: dst, id, ts, and $stage$. For every message m, $m.dst$ indicates to which groups m is A-MCast, $m.id$ is m's unique identifier, $m.ts$ denotes m's current timestamp, and $m.stage$ defines in which stage m is. We explain Algorithm $\mathcal{A}1$ by describing the actions a process p takes when a message m is in one of the three possible stages: s_0, s_1, or s_2.

To A-MCast m, m is first C-MCast to its addressees (line 8). In stage s_0, p C-Delivers m, sets m's timestamp proposal, and adds m to the set of pending messages $Pending$ (lines 10-12). In stage s_1, p computes $m.ts^{def}$ (lines 17-19) and ensures that all messages in $m.pps$ are in p's pending set (lines 20-23), as explained above. Finally, in stage s_2, m is A-Delivered when for all messages m' in $m.pps$ that are still in p's pending set (if m' is not in p's pending set anymore, m' was A-Delivered before), m' is in stage s_2 (and thus $m'.ts$ is the definitive timestamp of m') and $m'.ts$ is bigger than

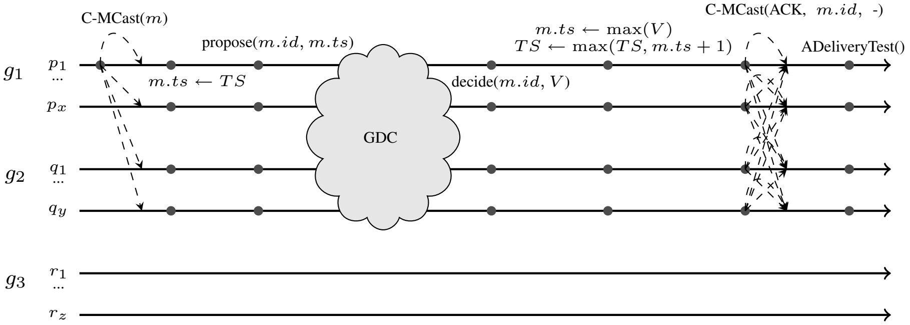


Fig. 1. Algorithm $\mathcal{A}1$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2

$m.ts$ (lines 4-6). Notice that if m and m' have the same timestamp, we break ties using their message identifiers. More precisely, $(m.ts, m.id) < (m'.ts, m'.id)$ holds if either $m.ts < m'.ts$ or $m.ts = m'.ts$ and $m.id < m'.id$. Figure 1 illustrates a failure-free run of the algorithm.

5 Solving Atomic Multicast with Weaker Failure Detectors

In this section, we solve atomic multicast with a non-genuine algorithm. The Algorithm $\mathcal{A}2$ we present next does not require system-wide perfect failure detection and delivers messages in fewer communication steps. We first define additional abstractions used by the algorithm and summarize its assumptions. We then present the algorithm itself and conclude with a discussion on how to further reduce its delivery latency and weaken its failure detection requirements.

5.1 Additional Definitions and Assumptions

Failure Detector $\Diamond\mathcal{P}$. We assume that processes have access to an eventually perfect failure detector $\Diamond\mathcal{P}$ [10]. This failure detector ensures the strong completeness property of $\mathcal{P}$ and the following *eventual strong accuracy* property: there is a time after which no process stops being trusted before it crashes.

Reliable Multicast. Reliable multicast is defined by the primitives R-MCast and R-Deliver and ensures all properties of causal multicast except uniform causal order.

Consensus. In the *consensus* problem, processes propose values and must reach agreement on the value decided. Consensus is defined by the primitives propose(v) and decide(v) and satisfies the following properties [8]: (i) *uniform validity:* if a process decides v, then v was previously proposed by some process, (ii) *termination:* if every correct process proposes a value, then every correct process eventually decides exactly one value, and (iii) *uniform agreement:* if a process decides v, then all correct processes eventually decide v.

Generic Broadcast. Generic broadcast ensures the same properties as atomic multicast except that all messages are addressed to all groups and only *conflicting* messages are totally ordered. More precisely, generic broadcast ensures uniform integrity, validity, uniform agreement, and the following *uniform generalized order* property: for any two conflicting messages m and m' and any two processes p and q, if p G-Delivers m and q G-Delivers m', then either p G-Delivers m' before m or q G-Delivers m before m'.

Assumptions. To solve generic broadcast, either a simple majority of correct processes must be correct, i.e., $f < n/2$, and non-conflicting messages may be delivered in three message delays [17] or a two-third majority of processes must be correct, i.e., $f < n/3$, and non-conflicting message may be delivered in two message delays [18]. Both algorithms require a system-wide leader failure detector Ω [11], and thus the eventual perfect failure detector $\Diamond\mathcal{P}$ we assume is sufficient. Moreover, inside each group, we need consensus and reliable multicast abstractions that tolerate an arbitrary number of failures.

Algorithm $\mathcal{A}1$. Genuine Atomic Multicast using $\mathcal{P}$ - Code of process p

```
Initialization
    $TS \leftarrow 1$, $Pending \leftarrow \emptyset$

procedure ADeliveryTest()
    while $\exists m \in Pending : m.stage = s_2$
            $\forall id \in m.pps : \exists m' \in Pending : m'.id = id \Rightarrow$
                    $m'.stage = s_2 \wedge (m.ts, m.id) < (m'.ts, m'.id)$ do
        A-Deliver($m$)
        $Pending \leftarrow Pending \setminus \{m\}$

To A-MCast message $m$                                    {Task 1}
    C-MCast $m$ to $m.dst$

When C-Deliver($m$) atomically do                         {Task 2}
    $m.ts \leftarrow TS$
    $m.stage \leftarrow s_0$
    $Pending \leftarrow Pending \cup \{m\}$

When $\exists m \in Pending : m.stage = s_0$                  {Task 3}
    $m.stage \leftarrow s_1$
    fork task ConsensusTask($m$)

ConsensusTask($m$)                                        {Task x}
    Propose($m.id$, $m.ts$)          ▷ GDC among processes in $m.dst$
    wait until Decide($m.id$, $V$)
    $m.ts \leftarrow \max(V)$
    $TS \leftarrow \max(TS, m.ts + 1)$
    C-MCast(ACK, $m.id, p$) to $m.dst$
    wait until $\forall q \in \mathcal{P} \cap m.dst$ : C-Deliver(ACK, $m.id, q$)
    $m.pps \leftarrow \{m'.id \mid m' \in Pending \wedge m' \neq m\}$
    $m.stage \leftarrow s_2$
    atomic block
        ADeliveryTest()
```

For this purpose, among realistic failure detectors, $\mathcal{P}$ is necessary and sufficient for consensus [12] and sufficient for reliable multicast [19].[6] Note that in practice, implementing $\mathcal{P}$ within each group is more reasonable than across the system, especially if groups are inside local area networks. We discuss below how to remove this assumption.

5.2 Algorithm Overview

The algorithm is inspired by the atomic broadcast algorithm of [5]. We first recall its main ideas and then explain how we cope with group failures—[5] assumes that there is at least one correct process in every group. We then show how *local* messages to some group g, i.e., messages multicast from processes inside g and addressed to g only, may be delivered with no inter-group communication at all.

[6] In [19], the authors present the weakest failure detector to solve reliable broadcast. Extending the algorithm of [19] to the multicast case using the same failure detector is straightforward.

To A-MCast a message m, a process p R-MCasts m to p's group. In parallel, processes execute an *unbounded* sequence of rounds. At the end of each round, processes A-Deliver a set of messages according to some deterministic order. To ensure agreement on the messages A-Delivered in round r, processes proceed in two steps. In the first step, inside each group g, processes use consensus to define g's bundle of messages. In the second step, groups exchange their message bundles. The set of message A-Delivered by some process p at the end of round r is the union of all bundles, restricted to messages addressed to p.

In case of group crashes, this solution does not ensure liveness however. Indeed, if a group g crashes there will be some round r after which no process receives the message bundles of g. To circumvent this problem we proceed in two steps: (a) we allow processes to stop waiting for g's message bundle, and (b) we let processes agree on the set of message bundles to consider for each round.

To implement (a), processes maintain a common *view* of the groups that are trusted to be alive, i.e., groups that contain at least one alive process. Processes then wait for the message bundles from the groups currently in the view. A group g may be erroneously removed from the view, if it was mistakenly suspected of having crashed. Therefore, to ensure that message m multicast by a correct process will be delivered by all correct addressees of m, we allow members of g to add their group back to the view. To achieve (b), processes agree on the sequence of views and the set of message bundles between each view change. For this purpose, we use a generic broadcast abstraction to propagate message bundles and view change messages, i.e., messages to add or remove groups. Since message bundles can be delivered in different orders at different processes, provided that they are delivered between the same two view change messages, we define the message conflict relation as follows: view change messages conflict with all messages and message bundles only conflict with view change messages. As view change messages are not expected to be broadcast often, such a conflict relation definition allows for faster message bundle delivery.

Processes may also A-Deliver local messages to some group g without communicating with processes outside of g. As these messages are addressed to g only, members of g may A-Deliver them directly after consensus, and thus before receiving the groups' message bundles.

We note that maintaining a common view of the alive groups in the system resembles what is called in the literature group membership [20]. Intuitively, a group membership service provides processes with a consistent view of alive processes in the system, i.e., processes "see" the same sequence of views. Moreover, processes agree on the set of messages delivered between each view change, a property that is required for message bundles.[7] In fact, our algorithm could have been built on top of such an abstraction. However, doing so would have given us less freedom to optimize the delivery latency of message bundles.

5.3 The Algorithm

Algorithm $\mathcal{A}2$ is composed of five tasks. Each line of the algorithm is executed atomically. On every process p, six global variables are used:Rnd denotes the current round

[7] Some group membership specifications also guarantee total ordering of the messages delivered between view changes.

number, $Rdelivered$ and $Adelivered$ are the set of R-Delivered and A-Delivered messages respectively, $Gdelivered$ is the sequence of G-Delivered messages, $MsgBundle$ stores the message bundles, and $View$ is the set of groups currently deemed to be alive.

In the algorithm, every G-BCast message m has the following format: $(rnd, g, type, msgs)$, where rnd denotes the round in which m was G-BCast, g is the group m refers to, $type$ denotes m's type and is either $msgBundle$, add, or $remove$, and $msgs$ is a set of messages; this field is only used if m is a message bundle.

To A-MCast a message m, a process p R-MCasts m to p's group (line 5). In every round r, the set of messages that have been R-Delivered but not A-Delivered yet are proposed to the next consensus instance (line 9), p A-Delivers the set of local messages decided in this instance (line 12), and global messages, i.e., non local messages, are G-BCast at line 15 if $group(p)$ belongs to the view. Otherwise, p G-BCasts a message to add $group(p)$ to the view.

Process p then gathers message bundles of the current round k using variable $MsgBundle$: Process p executes the while loop of lines 19-26 until, for every group g, $MsgBundle[g]$ is neither $\bot$, i.e. p is not waiting to receive a message bundle from g, nor $\top$, a value whose signification is explained below. The first message m_g^k of round k related to g of type $msgBundle$ or $remove$ that p G-Delivers "locks" $MsgBundle[g]$, i.e., any subsequent G-Delivered message of round k concerning g is discarded (line 23). If m_g^k is of type $msgBundle$, p stores g's message bundle in $MsgBundle[g]$ (line 26). Otherwise, m_g^k was G-BCast by some process q that suspected g to have entirely crashed, i.e., failure detector $\Diamond\mathcal{P}$ at q did not trust any member of g (lines 33-35), and thus p sets $MsgBundle[g]$ to $\emptyset$ (line 25). Note that q sets $MsgBundle[g]$ to $\top$ after G-BCasting a message of the form $(k, g, remove, \text{-})$ to prevent q from G-BCasting multiple "remove g" messages in the same round.

While p is gathering message bundles for round k, it may also handle some message of type add concerning g, in which case p adds g to a local variable $groupsToAdd$ (line 24). Note that this type of message is not tagged with a round number to ensure that messages A-MCast from correct groups are eventually A-Delivered by their correct addressees. In fact, tagging add messages with the round number could prevent a group from being added to the view as we now explain. Consider a correct group g that is removed from the view in the first round. In every round, members of g G-BCast a message to add g back to the view. In every round however, processes G-Deliver message bundles of groups in the view before G-Delivering these "add g" messages, and they are thus discarded.

After exiting from the while loop, p A-Delivers global messages (line 28), the view is recomputed as the groups g such that $MsgBundle[g] \neq \emptyset$ or $g \in groupsToAdd$ (line 30), and p sets $MsgBundle[g]$ to either $\bot$, if g belongs to the new view, or $\emptyset$ otherwise (p will not wait for a message bundle from g in the next round). Figures 2 and 3 respectively illustrate a failure-free run of the algorithm and a run where group g_3 entirely crashes.

5.4 Further Improvements

Delivery Latency In Algorithm $\mathcal{A}2$, local messages are delivered directly after consensus. Hence, these messages do not bear the cost of a single inter-group message

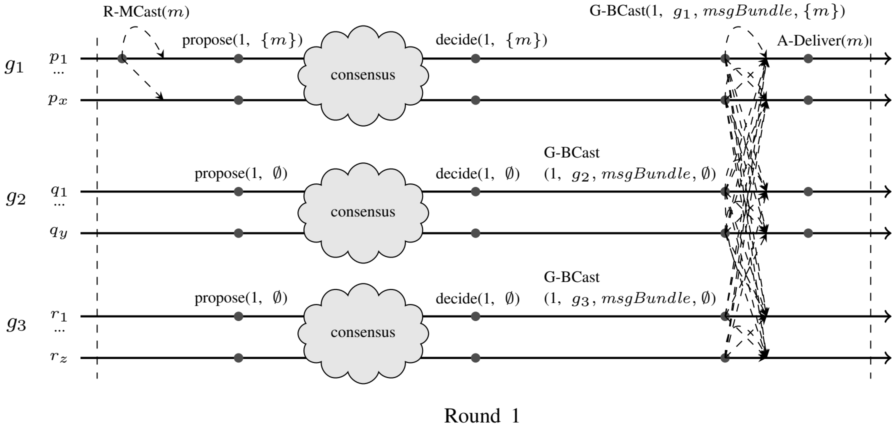


Fig. 2. Algorithm $\mathcal{A}2$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2

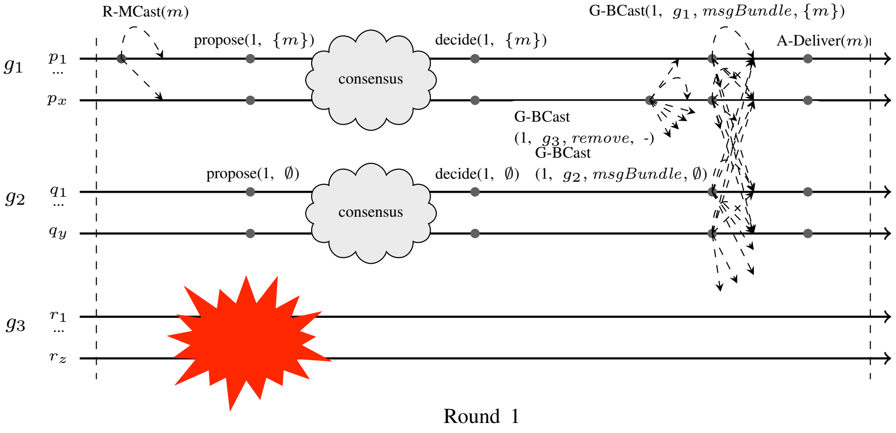


Fig. 3. Algorithm $\mathcal{A}2$ when group g_3 crashes and a message m is A-MCast to groups g_1 and g_2

delay unless: (a) they are multicast from a group different than their destination group or (b) they are multicast while the groups' bundle of messages are being exchanged, in which case the next consensus instance can only be started when message bundles of the current round have been received. Obviously, nothing can be done to avoid case (a). However, we can prevent case (b) from happening by allowing rounds to overlap. That is, we start the next round before receiving the groups' bundle of messages for the current round. Note that to ensure agreement on the relative delivery order of local and global messages, processes inside the same group must agree on when global messages of a given round are delivered, i.e., after which consensus instance. For this purpose, a mapping between rounds and consensus instances can be defined. To control the inter-group traffic, we may also specify that message bundles are sent, say every κ consensus instance. Choosing κ presents a trade-off between inter-group traffic and delivery latency of global messages.

Algorithm $\mathcal{A}2$. Non-Genuine Atomic Multicast - Code of process p

```
Initialization
    $Rnd \leftarrow 1$, $Rdelivered \leftarrow \emptyset$, $Adelivered \leftarrow \emptyset$, $Gdelivered \leftarrow \epsilon$
    $View \leftarrow \Gamma$, $MsgBundle[g] \leftarrow \bot$ for each group $g \in \Gamma$

To A-MCast message $m$                                                    {Task 1}
    R-MCast $m$ to $group(p)$

When R-Deliver($m$)                                                       {Task 2}
    $Rdelivered \leftarrow Rdelivered \cup \{m\}$

Loop                                                                      {Task 3}
    Propose($Rnd$, $Rdelivered \setminus Adelivered$)                ▷ consensus inside group
    wait until Decide($Rnd$, $msgs$)

    $localMsgs \leftarrow \{m \mid m \in msgs \;\wedge\; m.dst = \{group(p)\}\}$
    A-Deliver messages in $localMsgs$ in some deterministic order
    $Adelivered \leftarrow Adelivered \cup localMsgs$

    if $group(p) \in View$ then
        G-BCast($Rnd, group(p), msgBundle, msgs \setminus localMsgs$)
    else
        G-BCast(-, $group(p)$, $add$, -)
    $groupsToAdd \leftarrow \emptyset$

    while $\exists g \in \Gamma : MsgBundle[g] \in \{\bot, \top\}$
        if $\nexists (rnd, g, type, msgs) \in Gdelivered : (rnd = Rnd \;\vee\; type = add)$ then
            wait until G-Deliver($rnd, g, type, msgs$) $\wedge$ $(rnd = Rnd \;\vee\; type = add)$
        $(rnd, g', type, msgs) \leftarrow$ remove first message in $Gdelivered$ s.t.
                                    $(rnd = Rnd \;\vee\; type = add)$
        if $MsgBundle[g'] \in \{\bot, \top\}$ then
            if $type = add$ then $groupsToAdd \leftarrow groupsToAdd \cup \{g'\}$
            else if $type = remove$ then $MsgBundle[g'] \leftarrow \emptyset$
            else $MsgBundle[g'] \leftarrow msgs$
    $globalMsgs \leftarrow \{m \mid \exists g \in \Gamma : MsgBundle[g] = msgs \;\wedge\; m \in msgs\}$
    A-Deliver messages in $globalMsgs$ addressed to $p$ in some deterministic order
    $Adelivered \leftarrow Adelivered \cup globalMsgs$

    $View \leftarrow \{g \mid MsgBundle[g] \neq \emptyset\} \cup groupsToAdd$
    foreach $g \in \Gamma : MsgBundle[g] \leftarrow \bot$ (if $g \in View$) or $\emptyset$ (otherwise)
    $Rnd \leftarrow Rnd + 1$

When $\exists g \in View : MsgBundle[g] = \bot \;\wedge\; \forall q \in g : q \notin \diamond\mathcal{P}$     {Task 4}
    G-BCast($Rnd, g, remove$, -)
    $MsgBundle[g] \leftarrow \top$

When G-Deliver($type, m$)                                                 {Task 5}
    $Gdelivered \leftarrow Gdelivered \oplus (rnd, g, type, msgs)$
```

Failure Detection. To weaken the failure detector required inside each group, i.e., $\mathcal{P}$ in Algorithm $\mathcal{A}2$, we may remove a group g from the view as soon as a majority of processes in g are suspected. This allows to use consensus and reliable multicast algorithms that are safe under an arbitrary number of failures and live only when a majority of processes are correct. Hence, the leader failure detector Ω becomes sufficient. Care should be taken as when to add g to the view again: this should only be done when a majority of processes in g are trusted to be alive. This solution ensures a weaker liveness guarantee however: correct processes in some group g will *successfully* multicast and deliver messages only if g is *maj-correct*, i.e., g contains a majority of correct processes. More precisely, the liveness guaranteed by this modified algorithm is as follows (uniform integrity and uniform prefix order remain unchanged):

- *weak uniform agreement:* if a process p A-Delivers a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m
- *weak validity:* if a correct process p in a maj-correct group A-MCasts a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m.

6 Final Remarks

In this paper, we addressed the problem of solving atomic multicast in the case where groups may entirely crash. We presented two algorithms. The first algorithm is genuine, tolerates an arbitrary number of process failures, and requires perfect failure detection. We showed, in Section 1, that if we consider realistic failure detectors only and we do not bound the number of failures, $\mathcal{P}$ is necessary to solve this problem. The second algorithm we presented is not genuine but requires perfect failure detection inside each group only and may deliver messages addressed to multiple groups within two inter-group message delays. We showed how this latter algorithm can be modified to cope with unreliable failure detection, at the cost of a weaker liveness guarantee.

Figure 4 provides a comparison of the presented algorithms with the related work. The best-case message delivery latency is computed by considering a message A-MCast to k groups ($k \geq 2$) in a failure-free scenario when the inter-group message delay is δ

Algorithm	genuine?	resiliency	failure detector(s)	best-case latency
[2]	yes	majority correct in each group	group-wide Ω	$(k+1)\delta$
[3]	yes	majority correct in each group	group-wide Ω	4δ
[4]	yes	majority correct in each group	group-wide Ω	2δ
[5]	yes	majority correct in each group	group-wide Ω	2δ
$\mathcal{A}1$	yes	$f \leq n$	system-wide $\mathcal{P}$	6δ
$\mathcal{A}2$	no	$f < n/2$	group-wide $\mathcal{P}$ and system-wide $\Diamond\mathcal{P}$ (modification of algorithm with weaker liveness tolerates unreliable failure detection)	3δ
		$f < n/3$		2δ

Fig. 4. Comparison of the presented algorithms and related work

and the intra-group message delay is negligible. Note that we took 2δ as the best-case latency for causal multicast [13] and global data computation [16].

References

1. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. 5(1), 47–76 (1987)
2. Delporte-Gallet, C., Fauconnier, H.: Fault-tolerant genuine atomic multicast to multiple groups. In: Proceedings of OPODIS 2000, pp. 107–122. Suger, France (2000)
3. Rodrigues, L., Guerraoui, R., Schiper, A.: Scalable atomic multicast. In: Proceedings of IC3N 1998. IEEE, Los Alamitos (1998)
4. Fritzke, U., Ingels, P., Mostéfaoui, A., Raynal, M.: Fault-tolerant total order multicast to asynchronous groups. In: Proceedings of SRDS 1998, pp. 578–585. IEEE Computer Society, Los Alamitos (1998)
5. Schiper, N., Pedone, F.: On the inherent cost of atomic broadcast and multicast in wide area networks. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 147–157. Springer, Heidelberg (2008)
6. Frirzke Jr., U., Ingels, P.: Transactions on partially replicated data based on reliable and atomic multicasts. In: Proceedings of ICDCS 2001, pp. 284–291. IEEE Computer Society, Los Alamitos (2001)
7. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
8. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In: Mullender, S.J. (ed.) Distributed Systems, pp. 97–145. Addison-Wesley, Reading (1993)
9. Guerraoui, R., Schiper, A.: Genuine atomic multicast in asynchronous distributed systems. Theoretical Computer Science 254(1-2), 297–316 (2001)
10. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
11. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
12. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A realistic look at failure detectors. In: Proceedings of DSN 2002, pp. 345–353. IEEE Computer Society, Los Alamitos (2002)
13. Schiper, N., Pedone, F.: Solving atomic multicast when groups crash. Technical Report 2008/002, University of Lugano (2008)
14. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36(4), 372–421 (2004)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
16. Delporte-Gallet, C., Fauconnier, H., Helary, J.M., Raynal, M.: Early stopping in global data computation. IEEE Transactions on Parallel and Distributed Systems 14(9), 909–921 (2003)
17. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Thrifty generic broadcast. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 268–283. Springer, Heidelberg (2000)
18. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing 15(2), 97–107 (2002)
19. Aguilera, M.K., Toueg, S., Deianov, B.: Revising the weakest failure detector for uniform reliable broadcast. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 19–33. Springer, Heidelberg (1999)
20. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. 33(4), 427–469 (2001)

A Self-stabilizing Approximation for the Minimum Connected Dominating Set with Safe Convergence

Sayaka Kamei[1] and Hirotsugu Kakugawa[2]

[1] Dept. of Information Engineering, Graduate School of Engineering, Hiroshima University, 1-4-1 Kagamiyama, Higashi-Hiroshima, Hiroshima, 739-8527 Japan
s-kamei@se.hiroshima-u.ac.jp

[2] Dept. of Computer Science, Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama-cyo, Toyonaka, Osaka, 560-8531 Japan
kakugawa@ist.osaka-u.ac.jp

Abstract. In wireless ad hoc or sensor networks, a connected dominating set is useful as the virtual backbone because there is no fixed infrastructure or centralized management. Additionally, in such networks, transient faults and topology changes occur frequently.

A self-stabilizing system tolerates any kind and any finite number of transient faults, and does not need any initialization. An ordinary self-stabilizing algorithm has no safety guarantee and requires that the network remains static during converging to the legitimate configuration. Safe converging self-stabilization is one of the extension of self-stabilization which is suitable for dynamic networks such that topology changes and transient faults occur frequently. The safe convergence property guarantees that the system quickly converges to a safe configuration, and then, it moves to an optimal configuration without breaking safety.

In this paper, we propose a self-stabilizing 7.6-approximation algorithm with safe convergence for the minimum connected dominating set in the networks modeled by unit disk graphs.

Keyword: Self-stabilization, Approximation, Minimum connected dominating set, Mobile ad hoc or sensor networks, Fault-tolerance.

1 Introduction

1.1 Connected Dominating Set

Wireless ad hoc or sensor networks have no fixed physical backbone infrastructure and no centralized administration. Therefore, a *connected dominating set* (CDS) formed by processes is useful as a virtual backbone for the computation of message routing and other network problems for such networks.

In an undirected connected graph, a CDS D is a subset of nodes such that D is a dominating set and the induced subgraph by D is connected. The minimum CDS problem is finding a CDS of the minimum size. Unfortunately, it is known that the minimum CDS problem is NP-hard [1] in unit disk graphs. The unit disk graph is one of the models of ad hoc or sensor networks. In a unit disk graph, there is a link between two nodes if and only if their geographical distance is at most one unit. That is, for the

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 496–511, 2008.

sake of simplicity of analysis, it assumes that each process has the same communication range in ad hoc or sensor networks.

1.2 Self-stabilization with Safe Convergence

Fault-tolerant systems are classified into two categories: masking and non-masking [2]. If liveness property is guaranteed, but safety property is not guaranteed in the presence of faults, it called non-masking. *Self-stabilization* [3] is a theoretical framework of non-masking fault-tolerant distributed algorithms proposed by Dijkstra in 1974. Self-stabilizing algorithms can start execution from an arbitrary (illegitimate) system configuration, and eventually reach a legitimate configuration. By this property, they tolerate any kind and any finite number of transient faults, such as message loss, memory corruption, and topology change, if the network remains static during converging to their legitimate configurations [4]. That is, the system autonomously recovers without the cost of human interventions if transient faults and spontaneous reconfigurations occur.

In mobile ad hoc or sensor networks, message loss and topology change occur frequently. Thus, distributed algorithms for such networks should tolerate such events. Although self-stabilization property works quite well for networks in which the frequency of transient faults is low (compared to the time for convergence to a legitimate configuration), it may not be suitable for dynamic networks, such as mobile ad hoc or sensor networks, in which transient faults and topology changes frequently occur. Because, if a transient fault occurs during the convergence, the self-stabilizing algorithms cannot converge to the legitimate configuration. Additionally, they have not safety guarantee while it is in converging.

In such networks, during converging period, we want to guarantee a safety property by extending self-stabilization, called *safe convergence* [5]. When faults occur, it is better to converge to a safe feasible legitimate configuration as soon as possible. If no fault occurs for enough period of time, it is better to converge to an optimal configuration to provide the best quality of service. The safe convergence property requires that the system should not break safety while a system is moving from a feasible configuration to an optimal configuration.

There are many works on extensions of self-stabilization for quick convergence and guarantee of safe property, for example, superstabilization [6] and safe stabilization [7]. The concept of superstabilization guarantees the system quickly converges to a configuration. It considers only keeping safety in the event of faults or changes of the system in a legitimate configuration and does not consider the safety in converging configurations. The concept of safe stabilization guarantees any k faults in a safe configuration does not lead to an unsafe configuration, for some given constant k. Unfortunately, in both cases, they need very high cost on time and memory performance.

On the other hand, self-stabilization with safe convergence does not require any overhead, and implementation is much easier, because this framework does not guarantee safety when faults occur in a legitimate configuration. The related works for safe convergence are [8], [9], and [10].

1.3 Related Works

Because a CDS can be used for the virtual backbone for routing messages in the ad hoc network, many algorithms for the CDS have been proposed. The literature [11] is a good survey for this problem in ad hoc networks.

There exist some distributed approximation algorithms[1] with constant approximation ratio, for example [12], [13], [14], [15], [16], and [17]. However, these algorithms are not self-stabilizing, and not suitable for dynamic networks.

There are some self-stabilizing algorithms for computing CDSs, for example [18], [19] and [20]. However, [18] and [19] are not approximation algorithms, i.e., their algorithms do not guarantee qualities of their solutions. Additionally, they assume that 2 or 3-hops information can be maintained at each node, i.e., each node can refer to and update the local states of nodes in 2 or 3 hops away in a single step. Unfortunately, an efficient self-stabilizing implementation of such an assumption is not known that is comparable to our algorithm to be presented in this paper. In [20], we proposed the first self-stabilizing distributed approximation algorithm for the CDS. Unfortunately, it is not with safe convergence property. Therefore, it is not suitable for dynamic networks.

1.4 Contribution of This Paper

Self-stabilization with safe convergence is a good property for dynamic networks. We consider the problem to find an approximation of the minimum CDS with safe convergence. Our algorithm guarantees that the size of the solution is at most $7.6|D_{opt}| + 1.4$ in unit disk graphs. In general (topology) networks, our algorithm finds a CDS though it cannot guarantee the approximation ratio.

In general, self-stabilizing approximation algorithms (without safe convergence) need larger time and space complexity to attain better approximation ratio. For example, if we compute an optimal CDS, then it might take longer time. Therefore, we may want to compute a dominating set which is not necessarily optimal in a short time for virtual backbone, even if we know that the optimal CDS is better. Then, if the algorithm is with safe convergence property, it attends to such need.

By our algorithm, a configuration quickly moves to a feasible one in which a safety property is satisfied, i.e., a dominating set is computed. Then, as long as no transient fault occur, a configuration eventually becomes an optimal one in which an approximation of the minimum CDS is computed. By the safe convergence property, each configuration from a feasible one to an optimal one in the computation keeps the safety property, i.e., the set remains a dominating set. To construct the minimum CDS, the members of the dominating set in the feasible configuration leaves the set. Then, it is not trivial to keep the safety property during convergence from the feasible configuration to the optimal configuration.

We assume that all processes are executed in parallel in each step for the execution model. Therefore, a process and its neighbor processes may update their local states simultaneously. However, for safe convergence, each process can update its local state

[1] An approximation algorithm for the minimization problem is an algorithm which guarantees the approximation ratio $|D_{alg}|/|D_{opt}|$, where $|D_{alg}|$ is the size of the solution of the approximation algorithm in the worst case and $|D_{opt}|$ is the size of the optimal solution.

only when the safety property is not broken. On the other hand, when a configuration is feasible but not optimal, at least one process must make a move for convergence to the optimal. Therefore, designing an algorithm with safe convergence is not trivial under such execution model.

This paper is organized as follows. In section 2, we formally describe the system model and the distributed minimum CDS problem. In section 3, we present an outline of a heuristic algorithm of Marathe et al. [21] on which our algorithm is based. In section 4, we propose a safely converging self-stabilizing approximation algorithm for the minimum CDS in unit disk graph. In section 5, we show the proof of the correctness of the proposed algorithm. In section 6, we give a conclusion and discuss future works.

2 Preliminary

2.1 System Model

Let $V = (P_1, P_2, ..., P_n)$ be a set of n processes and $E \subseteq V \times V$ be a set of bidirectional communication links in a distributed system. The number of processes is denoted by n. Then, the topology of the distributed system is represented as an undirected graph $G = (V, E)$. We assume that G is connected and simple. In this paper, we use "graphs" and "distributed systems" interchangeably.

We assume that each process has unique process identifier. Let id be a naming function of processes. By $id(P_i)$, we denote the process identifier of P_i for each process P_i. In discussing process identifier, with abuse of notation, we use P_i to denote $id(P_i)$ when it is clear from the context.

By N_i, we denote the set of neighboring processes of P_i. For each P_i, the set N_i is assumed to be a constant. Let the *distance* between P_i and P_j be the number of the edges on the shortest path between them. For any set $S \subset V$ and any process $P_i \notin S$, let the distance between P_i and S be the minimum distance between P_i and any $P_j \in S$.

As a communication model, we assume that each process can read the local state of neighboring processes without delay. This model is called the *state reading model*. Although a process can read the local state of neighboring processes, it cannot update them; it can only update the local state of itself.

A set of local variables defines the local state of a process. By Q_i, we denote the local state of each process $P_i \in V$. A tuple of the local state of each process $(Q_1, Q_2, ..., Q_n)$ forms a *configuration* of a distributed system. Let Γ be a set of all configurations.

We define a *step* as an atomic execution unit. A step consists from the following three substeps: (1) Read states of all neighbors, (2) Compute the next local state, and (3) Update its local state. We assume that every process has an identical program including some steps, and every process executes the same step in parallel and in a synchronized manner. This assumption seems too strong for the self-stabilization, however, such execution model can be realized on an asynchronous model by using a phase clock synchronizer [22]. We define a *round* as a period from the beginning to the end of an execution of a loop of the program. We say that P_i is *enabled* in a configuration γ at the beginning of a round if and only if P_i executes any steps in the round and changes the value of at least one variable of itself.

2.2 Self-stabilization and Safe Convergence

For any configuration γ, let $\gamma\,'$ be any configuration that follows γ. Then, we denote this transition relation by $\gamma \rightarrow \gamma\,'$. For any configuration γ_0, a *computation E* starting from γ_0 is a maximal (possibly infinite) sequence of configurations $E = \gamma_0, \gamma_1, \gamma_2, ...$ such that $\gamma_t \rightarrow \gamma_{t+1}$ for each $t \geq 0$.

Definition 1. (Self-Stabilization) *Let Γ be a set of all configurations. A system S is* self-stabilizing *with respect to Λ such that $\Lambda \subseteq \Gamma$ if and only if it satisfies the following two conditions:*

- *Convergence: Starting from an arbitrary configuration, a configuration eventually becomes one in Λ, and*
- *Closure: For any configuration $\lambda \in \Lambda$, any configuration γ that follows λ is also in Λ as long as the system does not fail.*

Each $\gamma \in \Lambda$ is called a legitimate *configuration.* □

Definition 2. (Safe converging self-stabilization) *Let Γ be the set of all configurations, and let $\Lambda_O \subseteq \Lambda_F \subseteq \Gamma$. A self-stabilizing system S is* safely converging *with respect to (Λ_F, Λ_O) if and only if it satisfies the following three conditions:*

- *S is self-stabilizing with respect to Λ_F.*
- *Safe convergence: For any execution starting from configuration in Λ_F, configuration eventually reaches one in Λ_O.*
- *S is self-stabilizing with respect to Λ_O.*

Each $\gamma \in \Lambda_F$ is called a feasibly legitimate *configuration, and each $\gamma \in \Lambda_O$ is called an* optimally legitimate *configuration.* □

Definition 3. *Let S be a safely converging self-stabilizing system with respect to (Λ_F, Λ_O). The* first convergence time *is the number of steps to reach a configuration in Λ_F for any starting configuration in Γ. The* second convergence time *is the number of steps to reach a configuration in Λ_O for any starting configuration in Λ_F.* □

2.3 Formal Definition of the Problem

In this section, we give the formal definition of the problem.

Definition 4. *A* dominating set *of a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that there exist $v \in V'$ and $(u, v) \in E$ for any $u \in V \backslash V'$.* □

Definition 5. *An* independent set *of a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that $(u, v) \notin E$ for any $u, v \in V'$. An independent set V' of G is* maximal *if no proper superset of V' is an independent set of G.* □

In [23], the following relationship between dominating sets and independent sets is shown.

```
Arbitrarily pick a node $v_r \in V$.
Construct a BFS tree $T$ of $G$ rooted at $v_r$.
Let $k$ be the depth of $T$.
For each $0 \leq d \leq k$, let $L_d$ denote the set of nodes at distance $d$ from the root in $T$.
Set $\mathrm{I}_0 := \{v_r\};\ \mathrm{S}_0 := \emptyset$.
for $d = 1$ to $k$ do begin
    $\mathrm{D}_d := \{u \in L_d \mid u$ is dominated by some node in $\mathrm{I}_{d-1}\}$.
    Pick an MIS $\mathrm{I}_d$ in $G(L_d \setminus \mathrm{D}_d)$.
    $\mathrm{S}_d := \{u_f \mid u_f$ is the father in $T$ of some $v_i \in \mathrm{I}_d\}$.
end
output $(\cup_{d=0}^{k} \mathrm{I}_d) \cup (\cup_{d=0}^{k} \mathrm{S}_d)$ as the CDS.
```

Fig. 1. Marathe et al.'s algorithm

Theorem 1. *[23] An independent set is maximal if and only if it is independent and dominating.* □

For short, we call maximal independent set MIS.

Definition 6. *A* connected dominating set *of a graph $G = (V, E)$ is a dominating set $V' \subseteq V$ such that an induced subgraph by V' is connected. A connected dominating set V' of G is* minimum *if $|V'| \leq |V''|$ for any connected dominating set V'' of G.* □

We call the members of the CDS *dominators*, and others *dominatees*. Each dominatee is *dominated* by a dominator.

We consider solving the minimum CDS problem in distributed systems in this paper. We assume that each process P_i does not know global information of the network, and they know local information N_i which is a set of neighbors of P_i. We defined the distributed minimum CDS problem as follows.

Definition 7. *Let $G = (V, E)$ be a graph that represents a distributed system, let c_i be a variable that represents whether P_i is in the minimum connected dominating set. The* distributed minimum connected dominating set problem *is a problem defined as follows.*

- *Each process $P_i \in V$ must decide the value of $c_i \in \{0, 1\}$ as output of P_i, and*
- *The set $\{P_i \in V \mid c_i = 1\}$ is the minimum connected dominating set of G.* □

We assume that each process P_i has a local variable c_i.

3 Marathe et al.'s Algorithm

Marathe et al. proposed a sequential heuristic algorithm for the minimum CDS in unit disk graphs [21]. Because our algorithm is based on their algorithm, we present the outline of their algorithm.

The outline is described more formally in Figure 1. By $G(C)$, we denote an induced subgraph of G by a subset C of V.

First, their algorithm selects an arbitrary node v_r from G, and constructs a breadth first spanning (BFS) tree T of G rooted at v_r. For any node v_i, let $dist(v_r, v_i)$ denote the distance from v_r to v_i. Let k denote the height (i.e., the maximum distance) of T on G, and let L_d be the set of nodes which have the distance (i.e., the depth) d from the root ($0 \leq d \leq k$), i.e., $L_d = \{v_i \mid dist(v_r, v_i) = d\}$.

The CDS by the heuristic is the union of two subsets of nodes, i.e., $(\cup_{d=0}^{k} I_d) \cup (\cup_{d=0}^{k} S_d)$.

- The first subset $\cup_{d=0}^{k} I_d$ is an MIS for G. The root v_r definitely joins a set I_0. Let D_d be a set of nodes $v_i \in L_d$ each of which is dominated by some node in I_{d-1}. For each $1 \leq d \leq k$, a set I_d is an MIS of an induced subgraph of G by $L_d \setminus D_d$. That is to say, the heuristic paves the field dominated by members of I in order of increasing of $dist(v_r, v_i)$ from v_r.
- The second subset is $\cup_{d=0}^{k} S_d$, where S_d is a set of nodes which are fathers of some I_d for each $1 \leq d \leq k$. Note that, $S_d \subseteq L_{d-1}$.

We call the above way of construction of an MIS $\cup_{d=0}^{k} I_d$ "*paving* on a BFS tree". For any set $C \subsetneq V$ and any node $v_i \notin C$, let the distance between v_i and C be the minimum distance between v_i and any $v_j \in C$. On the MIS constructed by paving on a BFS tree, the set satisfies the following property.

Theorem 2. *[15] Let I' be the MIS constructed by paving on a BFS tree T. For any v_i in I', the distance between v_i and $I' \setminus \{v_i\}$ is exactly two hops.* □

By Theorem 2, the connectivity of the CDS is ensured. In the MIS constructed by paving on T, each member $v_i \in L_d$ of the MIS has a father on T which is neighbor to at least one member of the MIS in L_{d-1} or L_{d-2}. Therefore, the union of the MIS and a set of fathers of members of the MIS is connected. Because the MIS is also a dominating set by Theorem 1, the union is a CDS.

Definition 8. *Let T be a BFS tree on G, and $dist(v_i)$ be the distance from the root to a node v_i on T. For any MIS I' for G, I' is an* MIS constructed by paving on T, *if each member v_i of I' which has $dist(v_i) = d$ has the following two nodes:*

- *a father $v_j \in N_i$ of v_i on T, and*
- *a neighbor v_k ($\neq v_i$) of v_j which is a member of I' and has $dist(v_k) = d - 1$ or $dist(v_k) = d - 2$.* □

We define such a CDS as *CDS-tree* formally as follows:

Definition 9. *Let I' be any MIS constructed by paving on a BFS tree T for G. Let $S'(\neq \emptyset)$ be a set of nodes each of which is the father of a member in I' on T. A set of nodes $I' \cup S'$ is a* CDS-tree *for G.* □

In [24], Wu et al. prove the following theorem about the relationship between the minimum CDS and MISs in unit disk graphs.

Theorem 3. *[24] For any unit disk graph, the size of an MIS is at most $3.8|D_{opt}| + 1.2$, where D_{opt} is the minimum CDS.* □

By Theorem 3, we proof the following theorem.

Theorem 4. *Let D_{opt} be the minimum CDS. Any CDS-tree is an approximation for the minimum CDS which size is at most $7.6|D_{opt}| + 1.4$ in unit disk graphs.*

Proof. We consider a CDS-tree $I' \cup S'$, where I' be an MIS constructed by paving on a BFS tree and S' be a set of nodes each of which is the father of a member in I' on the BFS tree.

By Theorem 3, the size of I' is at most $3.8|D_{opt}| + 1.2$. Because each member of I' which is not the root of the BFS tree has a father in S', the size of S' is at most $|I'| - 1 = 3.8|D_{opt}| + 0.2$.

Therefore, the size of the CDS-tree is $|I' \cup S'| = |I'| + |S'| \leq 7.6|D_{opt}| + 1.4$. □

4 Proposed Algorithm

Our algorithm SC-CDS is safe converging: we assume that the safety property is "a dominating set is computed". That is, SC-CDS computes a dominating set in the first round, and then, it converges to a CDS-tree. During the converging, the set remains a dominating set in each configuration.

Our algorithm SC-CDS is based on the strategy of Marathe et al.'s algorithm in [21]. First, SC-CDS computes a BFS tree T rooted at P_r [2] for G, i.e., each process P_i computes the distance d_i from P_r. Because an algorithm for computing a BFS tree has been proposed so far, for example [28], we simply adopt it to our system model. For purposes of illustration, let k denote the height of T on G, and let L_d be the set of processes which have $d_i = d$ $(0 \leq d \leq k)$. Next, SC-CDS computes an MIS constructed by paving on T. For constructing an MIS, there exist many self-stabilizing algorithms, for example [29]. However, these algorithms do not ensure that a computed MIS is not the same as the MIS constructed by paving on T. Therefore, we do not use these algorithms in SC-CDS. In SC-CDS, the members of the MIS are selected greedy from the root P_r to leaves on T. Last, SC-CDS selects members of a CDS-tree, i.e., members of the MIS and their fathers.

To guarantee the safety property, SC-CDS computes a larger dominating set, even if the BFS tree is broken. After that, while SC-CDS constructs a BFS tree, it decrease the members carefully to construct a minimum CDS. When a process leaves the set, the set remains a dominating set. Then, the set gradually becomes a union of an MIS and the set of fathers of the MIS.

Formal description of proposed algorithm SC-CDS is shown in Figure 2.

We assume, without loss of generality, the output of each process P_i is following five variables as output.

- d_i — the distance from the root process P_r to P_i.
- f_i — an id of a father of P_i on T.
- m_i — $m_i = 1$ (resp. 0) if P_i is in an MIS (resp. not in an MIS).
- m'_i — $m'_i = 1$ (resp. 0 or 2) if P_i wants to change the value m_i to 1 (resp. 0).
- c_i — $c_i = 1$ (resp. 0) if P_i is a dominator (resp. dominatee).

[2] We assume that P_r is given as a specific process. We can elect it as a leader by leader election algorithms, for example [25], [26] and [27].

Constant

N_i: a set of neighboring processes of P_i.

Local Variable

d_i: the distance from the root process P_r.
f_i: an id of a father of P_i in the BFS tree.
$m_i \in \{0, 1\}$: $m_i = 1$ (resp. 0) if P_i is (resp. is not) in the MIS.
$m'_i \in \{0, 1, 2\}$: $m'_i = 1$ (resp. 0 or 2) if P_i wants to change the value m_i to 1 (resp. 0).
$c_i \in \{0, 1\}$: $c_i = 1$ (resp. 0) if P_i is a dominator (resp. a dominatee).

Macro

$MinDist_i \equiv \min\{d_j \mid P_j \in N_i \wedge m'_j = 2\}$
$Mutex_i \equiv \forall P_j \in N_i[m'_j \neq 2] \vee d_i < MinDist_i \vee$
$\{d_i = MinDist_i \wedge P_i < \min\{P_j \in N_i \mid d_j = d_i \wedge m'_j = 2\}\}$

Algorithm for process $P_i \neq P_r$:

```
do forever{
/* Step 1; Count the distance from $P_r$ for the BFS tree, and decide a father. */
    $d_i := \min\{d_j + 1 \mid P_j \in N_i\}$;
    $f_i := \min\{P_j \in N_i \mid d_j < d_i\}$;
/* Step 2; Declare if $P_i$ wants to join $MIS$ or not. */
    $m'_i := m_i$;
    if $(m_i = 1 \wedge \exists P_j \in N_i[d_j \leq d_i \wedge m_j = 1])$ $m'_i := 2$;
    if $(m_i = 0 \wedge \forall P_j \in N_i[d_j > d_i \vee m_j = 0])$ $m'_i := 1$;
/* Step 3; If $P_i$ wants to leave $MIS$, then $P_i$ decides its value
            by mutually exclusive manner between neighbors. */
    if $(m'_i = 2)$ {
        if $(Mutex_i)$ $m'_i := 0$;
        else $m'_i := 1$;
    }
/* Step 4; Change the value of $m_i$, i.e., construct $MIS$. */
    if $(m'_i = 0 \wedge \forall P_j \in N_i[m'_j = 0])$ $m_i := 1$;  /* for safety */
    else $m_i := m'_i$;
/* Step 5; Change the value of $c_i$, i.e., construct $Doms$.*/
    if $(m_i = 1)$ $c_i := 1$;
    else if $(\exists P_j \in N_i[f_j = P_i \wedge m_j = 1])$ $c_i := 1$;
    else $c_i := 0$;
}
```

Algorithm for process P_r:

```
do forever{
    $d_r := 0$;
    $f_r := P_r$;
    $m'_r := 1$;
    $m_r := 1$;
    $c_r := 1$;
}
```

Fig. 2. SC-CDS: A safe converging self-stabilizing approximation algorithm for the minimum CDS for each process P_i

Definition 10. *For each configuration $\gamma \in \Gamma$, we define $MIS(\gamma) \equiv \{P_i \in V \mid m_i = 1\}$, which is called an independent set in γ, and $Doms(\gamma) \equiv \{P_i \in V \mid c_i = 1\}$, which is called a set of dominator processes in γ.* □

There is only one step for the root process P_r in a round of this algorithm.

- P_r sets the value of $d_r = 0$, $f_r = P_r$, $m_r = m'_r = 1$, and $c_r = 1$.

There are five steps for non-root process $P_i \neq P_r$ in a round of this algorithm.

- Step 1: P_i computes the distance d_i from P_r and a father f_i on T by lines 1 and 2, respectively.
- Step 2: P_i declares whether P_i wants to join MIS.
  - If each neighbor $P_j \in N_i$ with $d_i \geq d_j$ is not a member of MIS, then P_i declares participation in MIS (i.e., $m'_i = 1$) by line 5.
  - If there exists a neighbor P_j with $d_i \geq d_j$ such that P_j is a member of MIS, then P_i declares non-participation in MIS. That is, for the time being, P_i sets $m'_i = 2$ for the next step by line 4.
- Step 3: P_i with $m'_i = 2$ decides if P_i actually leaves MIS.
  - By mutually exclusive manner between P_i and its neighbors P_j with $m'_j = 2$, P_i decides if P_i leaves MIS (i.e., $m'_i = 0$) by line 7 or P_i stays in MIS (i.e., $m'_i = 1$) by line 8.
- Step 4: P_i decides if P_i joins MIS.
  - If P_i has $m'_i = 0$ and all neighbor P_j has $m'_j = 0$, then P_i joins MIS for safety by line 10.
  - Otherwise, P_i decides if P_i joins MIS by the value of m'_i by line 11.
- Step 5: P_i decides if P_i joins $Doms$ or not.
  - If P_i or at least one child of P_i on T is in MIS, then P_i joins $Doms$ by lines 12 and 13.
  - Otherwise, P_i leaves $Doms$ by line 14.

By Γ, we denote a set of all configurations of SC-CDS. A set of legitimate configurations is defined as follows.

Definition 11. *A configuration γ is in a set of* feasibly legitimate *configurations Λ_F iff $Doms(\gamma)$ is a dominating set. A configuration γ is in a set of* optimally legitimate *configurations Λ_O iff $Doms(\gamma)$ is a CDS-tree.* □

5 Proof of Correctness

In this section, we show the proof of correctness of SC-CDS. However, we omit the proof for the limitation of space.

Lemma 1. *(One round convergence to Λ_F) Let γ be any configuration in Γ, and γ' be a configuration at the end of a round execution starting from γ. Then, we have $\gamma' \in \Lambda_F$.*

Proof. For the contrary, we assume that $\mathit{Doms}(\gamma')$ is not a dominating set. This means that there exists a process P_i such that $c_i = 0 \wedge \forall P_j \in N_i[c_j = 0]$ in γ'. It is clear that $P_i \neq P_r$ because $c_r = 1$ by line 5 for P_r in γ'. Then, by the definition of Step 5, each process P_k with $m_k = 1$ holds $c_k = 1$. Therefore, $m_i = 0 \wedge \forall P_j \in N_i[m_j = 0]$ in γ'. Because $m_i = 0$ in γ', $m'_i = 0 \wedge \exists P_j \in N_i[m'_j \neq 0]$ must hold by lines 10 and 11. Let $P_j \in N_i$ be a process such that $m'_j \neq 0$. By the definition of Step 3, each process must hold $m' = 0$ or $m' = 1$ in γ'. That is, $m'_j \neq 2$ in γ'. Therefore, $m'_j = 1$ holds in γ'. By the definition of Step 4, then $m_j = 1$ in γ'. By the definition of Step 5, then $c_j = 1$ in γ'. This is a contradiction. Therefore, $\mathit{Doms}(\gamma')$ is a dominating set. □

Lemma 2. *(Closure of Λ_F) Let γ be any configuration in Λ_F, and γ' be any configuration at the end of a round execution starting from γ. Then, we have $\gamma' \in \Lambda_F$.*

Proof. By the proof of Lemma 1, this lemma trivially holds. □

Lemma 3. *If no process is enabled in configuration γ, $\mathit{MIS}(\gamma)$ is an MIS constructed by paving on a BFS tree.*

Proof. Let γ be a configuration in which no process is enabled. By each line 1 for P_r and P_i, it is clear that the value of d_i, for each P_i, represents the distance from P_r in γ [28]. This means that a BFS tree T is computed in γ. Assume that $\mathit{MIS}(\gamma)$ is not an MIS constructed by paving on T in γ. Then, $\mathit{MIS}(\gamma)$ is not an independent set, is an independent set but it is not maximal, or is an MIS but it is not constructed by paving on T.

- We assume that $\mathit{MIS}(\gamma)$ is not an independent set, i.e., there exist two processes P_i and P_j in $\mathit{MIS}(\gamma)$ such that they are neighbor each other in γ. This means that $m_i = 1 \wedge \exists P_j \in N_i[m_j = 1]$ (resp. $m_j = 1 \wedge \exists P_i \in N_j[m_i = 1]$) holds at P_i (resp. P_j) in γ. If $d_i > d_j$ (resp. $d_j > d_i$, $d_i = d_j$), the condition of line 4 is true at P_i (resp. P_j, P_i and P_j). This is a contradiction.
- We assume that $\mathit{MIS}(\gamma)$ is an independent set but it is not maximal in γ. However, by the proof of lemma 1, there exists no process such that $m_i = 0 \wedge \forall P_j \in N_i[m_j = 0]$ holds, i.e., $\mathit{MIS}(\gamma)$ is a dominating set. Therefore, $\mathit{MIS}(\gamma)$ is a dominating set and an independent set in γ. By Theorem 1, this is a contradiction.
- We assume that $\mathit{MIS}(\gamma)$ is an MIS but it is not constructed by paving on T in γ. Then, there exist two processes $P_i \neq P_r$ and $P_j \in N_i$ such that $m_i = 1$, $m_j = 0 \wedge d_j = d_i - 1$, and there exists no neighbor $P_k \in N_j$ such that $m_k = 1 \wedge d_k \leq d_j$ in γ. Then, $\forall P_k \in N_j[d_k > d_j \vee m_k = 0]$ holds at P_j, and the condition of line 5 is true at P_j. This is a contradiction.

Therefore, $\mathit{MIS}(\gamma)$ is an MIS constructed by paving on T if no process is enabled. □

Lemma 4. *No process is enabled in configuration γ if and only if $\gamma \in \Lambda_O$.*

Proof. First, we show that if no process is enabled in configuration γ, then $\gamma \in \Lambda_O$, that is, $\mathit{Doms}(\gamma)(= \{P_i \mid c_i = 1\})$ is a CDS-tree. By each line 2 for P_r and P_i, it is clear that the value of f_i, for each P_i, represents a father of P_i on a BFS tree T in γ. By Lemma 3, the set $\{P_i \mid m_i = 1\}$ is an MIS constructed by paving on T in γ. Therefore, by the definition of the CDS-tree, $\mathit{Doms}(\gamma)$ is $(\{P_i \mid m_i = 1\} \cup \{P_j \mid f_i = P_j \wedge m_i = 1\})$ in γ iff $\gamma \in \Lambda_O$.

To show the contraposition, we assume that $\gamma \notin \Lambda_O$, i.e., $Doms(\gamma) \neq (\{P_i \mid m_i = 1\} \cup \{P_j \mid f_i = P_j \wedge m_i = 1\})$.

- Assume that $\{P_i \mid m_i = 1\} \not\subseteq Doms(\gamma)$, i.e., there exists a process P_i such that $m_i = 1$ and $c_i = 0$. If $P_i = P_r$, then P_i is enabled by line 5 in γ. If $P_i \neq P_r$, then the condition of line 12 is true in γ. This is a contradiction for the assumption that no process is enabled in γ. Therefore, we have $\{P_i \mid m_i = 1\} \subseteq Doms(\gamma)$. This means that $Doms(\gamma)$ is a dominating set, because $\{P_i \mid m_i = 1\}$ is an MIS by Lemma 3 and an MIS is also a dominating set by Theorem 1.
- Assume that $\{P_j \mid f_i = P_j \wedge m_i = 1\} \not\subseteq Doms(\gamma)$, i.e., there exist two processes P_i and $P_j \in N_i$ such that $m_i = 1$, $c_j = 0$ and $f_i = P_j$. Because $\{P_i \mid m_i = 1\}$ is a MIS and P_j is a neighbor of P_i, $m_j = 0$ holds at P_j. Then, in P_j, $m_j = 0 \wedge \exists P_i \in N_j[f_i = P_j \wedge m_i = 1]$ is true, i.e., the condition of line 13 is true in γ. This is a contradiction for the assumption that no process is enabled in γ. Therefore, $\{P_j \mid f_i = P_j \wedge m_i = 1\} \subseteq Doms(\gamma)$. This means that $Doms(\gamma)$ is a CDS, because $Doms(\gamma)$ is a dominating set and members of $\{P_j \mid f_i = P_j \wedge m_i = 1\}$ connect members of $Doms(\gamma)$.
- Assume that $Doms(\gamma)$ is a CDS, but not a CDS-tree. That is $Doms(\gamma) \supsetneq (\{P_i \mid m_i = 1\} \cup \{P_j \mid f_i = P_j \wedge m_i = 1\})$. Then, there exists a process P_i such that $c_i = 1$, but neither P_i nor its children are members of the MIS. That is, $m_i = 0 \wedge \forall P_j \in N_i[f_j \neq P_i \vee m_j = 0]$ holds at P_i. However, by the definition of lines 12-14, P_i can change the value of c_i by line 14. This is a contradiction for the assumption that no process is enabled in γ.

Therefore, if no process is enabled in configuration γ, $\gamma \in \Lambda_O$, that is, $Doms(\gamma)$ is a CDS-tree.

It is clear that no process is enabled if $\gamma \in \Lambda_O$. □

Lemma 5. *For any configuration $\gamma_0 \in \Lambda_F$ and any computation starting from γ_0, eventually no process is enabled.*

Proof. By the definition of the algorithm, the root process P_r changes values of each variable at most once. Then, P_r decides the value of $d_r = 0$ (resp. $f_r = P_r$, $m'_r = 1$, $m_r = 1$, and $c_r = 1$) only in line 1 (resp. 2, 3, 4 and 5) for P_r. These values never change after that, because they are not changed by other lines. Therefore, we suppose below that values of them are correct at P_r, and we consider each process $P_i \neq P_r$.

By line 1 for each process $P_i \neq P_r$, P_i changes the value of d_i only in line 1, and it is shown that each P_i cannot change the value of d_i infinitely often [28]. Therefore, we assume that the value of d_i is stable and never changes for each P_i in γ_0. By line 2, it is clear that the value of f_i is fixed for each P_i after a round execution following the round in which each value of d_i becomes correct at each P_i. Therefore, we assume that the value of f_i is stable and never changes for each P_i in γ_0.

Suppose that there exists an infinite (non-converging) computation starting from γ_0. Then, there is a process P_i which changes values of m_i, m'_i and c_i infinitely often. By the definition of Step 5, to change the value of c_i infinitely often, P_i must change the value of m_i infinitely often. By the definition of Step 4, to change the value of m_i infinitely often, P_i must change the value of m'_i infinitely often. Therefore, without loss of generality, we assume that there exists a process P_i which changes the value of m'_i infinitely often.

- Suppose that $d_i = 1$. However, because $m_r = 1$ holds at P_r and the value never change, $\exists P_j \in N_i[d_j \leq d_i \wedge m_j = 1]$ is always true at P_i. Therefore, P_i executes line 4 and sets $m'_i = 2$ as long as $m_i = 1$. Then, P_i executes Step 3, and decides m'_i into 0 or 1. P_i eventually executes line 7 at most once. Then, $m'_i = 0$ holds at P_i, and it never change. Because $m'_r = 1$, P_i never executes line 10. That is, P_i eventually holds $m_i = 0$ by line 11. Therefore, P_i with $d_i = 1$ cannot change values of m_i and m'_i infinitely often.
- Suppose that $d_i = d > 1$. For induction, we assume that each P_h with $d_h = d - 1$ never change the value of m_h and m'_h. Then, by the definition of lines 4 and 5, $P_j \in N_i$ with $d_j = d$ must change the value of m_j infinitely often.
  - If there exists a process $P_h \in N_i$ with $d_h = d - 1$ and $m_h = 1$, then P_i cannot execute line 5 by its condition. Then, P_i cannot change the value of m'_i infinitely often. This is a contradiction for the assumption. Therefore, there exist no process $P_h \in N_i$ with $d_h = d - 1$ and $m_h = 1$.
  - If there exist no process $P_h \in N_i$ with $d_h = d - 1$ and $m_h = 1$, P_i can change the value of m'_i from 0 to 1 by line 5 only when all its neighbors P_j with $d_j \leq d$ hold $m_j = 0$ by the condition of line 5. After P_i execute line 5, the condition of line 5 cannot true at all of its neighbors P_j with $d_j = d$. Therefore, P_j cannot change the value of m'_j infinitely often. This means that P_j cannot change the value of m_j infinitely often by the definition of Step 4. This is a contradiction for the assumption. Therefore, P_i cannot change values of m_i and m'_i infinitely often.

Therefore, P_i cannot change the value of its variable infinitely often. □

Theorem 5. SC-CDS *is a self-stabilizing approximation algorithm for the minimum CDS in unit disk graphs, and is safely converging self-stabilizing with respect to* (Λ_F, Λ_O). *The size of the CDS by* SC-CDS *is at most* $7.6|D_{opt}| + 1.4$ *in unit disk graphs, where* D_{opt} *is the minimum CDS. The first convergence time is at most 1 round, and the second convergence time is* $O(n)$ *rounds.*

Proof. The first half of the following theorem clears from Theorem 4 and Lemmas 1-5. By Lemmas 1 and 2, SC-CDS is self-stabilizing with respect to Λ_F, and the first convergence time is at most 1 round. By Lemmas 4 and 5, SC-CDS is self-stabilizing with respect to Λ_O, and satisfies the safe convergence property. Therefore, SC-CDS is safely converging and self-stabilizing with respect to (Λ_F, Λ_O). By Theorem 4, it is clear that the size of the CDS by SC-CDS is at most $7.6|D_{opt}|+1.4$ in unit disk graphs, where D_{opt} is the minimum CDS.

Let us derive the second convergence time. First, we consider the construction of the BFS tree by Step 1. In our system model, the root P_r must decide its variable $d_r = 0$ in the first round, and it never change. Each neighboring process P_i of P_r must decide its variable $d_i = 1$ in the second round, and its value never change. Therefore, each process P_k which is in k hops from P_r must decide its variable $d_k = k$ in the $k + 1$-th round. Therefore, the time for construction of the BFS tree is at most $k + 1$ rounds, where k is the height of the tree. After that, each process fixes its variable of f_i within a round. Therefore, in the $k+2$-th round, values of d_i and f_i are fixed at each P_i.

Next, we consider the construction of the MIS after the construction of the BFS tree, i.e., the execution of Steps 2-4. Let L_d be the set of processes P_i with $d_i = d$ $(0 \leq d \leq k)$. Let l_d be the size of the set L_d for $0 \leq d \leq k$, then $\Sigma_{d=0}^{k} l_d = n$. The root P_r must decide its variable $m_r = m'_r = 1$ in the first round, and it never change. From the second round, $\exists P_j \in N_i[d_j \leq d_i \wedge m_j = 1]$ always holds at each neighboring process P_i of P_r. Then P_i cannot execute line 5. If $m_i = 1$, P_i executes line 4, Step 3 and Step 4 until m'_i and m_i become 0 by lines 7 and 11. Therefore, the time until all processes in L_1 stable is at most l_1 rounds.

We assume that the processes in a set $Q = L_0 \cup L_1 \cup \ldots L_{h-1}$ is stable and never change after the round. In the next round, processes $L_h \cup L_{h+1} \cup \cdots \cup L_k$ execute line 4 or 5. If $P_i \in L_h$ is neighbor to a process P_j with $m_j = 1$ in Q, then P_i can execute only line 4, and execute line 7 at most once. Therefore, P_i executes at most l_h rounds by the above discussion. If $P_i \in L_h$ is not neighbor to such process $P_j \in Q$, then P_i can execute lines 4 and 5. By the proof of Lemma 5, in this case, P_i changes the value of m_i at most twice. Therefore, P_i executes line 4 at most l_h rounds and line 5 at most 1 round. Therefore, each process $P_i \in L_h$ executes at most $l_h + 1$ rounds.

Because $\Sigma_{d=0}^{k} l_d = n$, the time for the construction of the MIS is at most $\Sigma_{d=0}^{k}(l_d + 1) = n + k$ rounds. That is, in at most $n + 2k + 2$ rounds, values of m'_i and m_i are fixed at each P_i.

After that, each process P_i executes Step 5, and fixes the value of c_i within a round.

Therefore, the second convergence time is $O(n)$ rounds.

□

6 Conclusion

In this paper, we proposed a self-stabilizing distributed approximation algorithm for the minimum CDS with safe convergence in unit disk graphs. As an application of the proposed algorithm, a minimum CDS is a virtual backbone in mobile ad hoc or sensor networks. Since our algorithm is self-stabilizing with safe convergence, it is strongly desirable in such dynamic networks. Our algorithm converges to a safe configuration in a round, and to an optimal configuration in $O(n)$ rounds. Our algorithm guarantees that the size of the solution in unit disk graphs is at most $7.6|D_{opt}| + 1.4$. Development of a safely converging self-stabilizing approximation algorithm with better approximation ratio or better time complexity is left for future work.

Acknowledgment

This work is supported in part by Kayamori Foundation of Informational Science Advancement, a Grant-in-Aid for Young Scientists ((B)19700075) of JSPS, a Grant-in-Aid for Scientific Research ((B)17300020, (B)20300012) of JSPS, and “Global COE (Centers of Excellence) Program” of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

1. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit disk graphs. Discrete Mathematics 86(1-3), 165–177 (1990)
2. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Computing Surveys 31(1), 1–26 (1999)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
5. Kakuagwa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: Proceedings IPDPS Workshop on Advances on Parallel and Distributed Computational Model (APDCM), p. 263 (2006)
6. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago Journal of Theoretical Computer Science 3(4), 1–40 (1997)
7. Ghosh, S., Bejan, A.: A framework of safe stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 129–140. Springer, Heidelberg (2003)
8. Cobb, J.A., Gouda, M.G.: Stabilization of general loop-free routing. Journal of Parallel and Distributed Computing 62, 922–944 (2002)
9. Johnen, C., Tixeuil, S.: Routing preserving stabilization. In: Huang, S.-T., Herman, T. (eds.) SSS 2003. LNCS, vol. 2704, pp. 184–198. Springer, Heidelberg (2003)
10. Kamei, S., Kakugawa, H.: A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In: Proceedings of the 1st International Workshop on Reliability, Availability, and Security (WRAS), pp. 57–66 (2007)
11. Blum, J., Ding, M., Thaeler, A., Cheng, X.: Connected dominating set in sensor networks and MANETs. In: Handbook of Combinatorial Optimization, pp. 329–369. Kluwer Academic Publishers, Dordrecht (2004)
12. Wu, J., Lou, W.: Forward-node-set-based broadcast in clustered mobile ad hoc networks. Wireless Networks and Mobile Computing 3(2), 155–173 (2003)
13. Gao, B., Yang, Y., Ma, H.: A new distributed approximation algorithm for constructing minimum connected dominating set in wireless ad hoc networks. International Journal of Communication System 18, 743–762 (2005)
14. Cheng, X., Du, D.Z.: Virtual backbone-based routing in multihop ad hoc wireless networks. Technical report, University of Minnesota (2002)
15. Wan, P.J., Alzoubi, K.M., Frieder, O.: Distributed construction of connected dominating set in wireless ad hoc networks. Mobile Networks and Applications 9(2), 141–149 (2004)
16. Min, M., Du, H., Jia, X., Huang, C.X., Huang, S.C.H., Wu, W.: Improving construction for connected dominating set with Steiner tree in wireless sensor networks. Journal of Global Optimization 35, 111–119 (2006)
17. Li, Y., Thai, M.T., Wang, F., Yi, C.W., Wan, P.J., Du, D.X.: On greedy construction of connected dominating sets in wireless networks. Wireless Communications and Mobile Computing 5, 927–932 (2005)
18. Jain, A., Gupta, A.: A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. In: Proceedings of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pp. 615–619 (2005)
19. Drabkin, V., Friedman, R., Gradinariu, M.: Self-stabilizing wireless connected overlays. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 425–439. Springer, Heidelberg (2006)
20. Kamei, S., Kakugawa, H.: A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. In: Proceedings of the 9th IPDPS Workshop on Advances in Parallel and Distributed Computational Models (APDCM), p. 224 (2007)

21. Marathe, M.V., Breu, H., Hunt III, H.B., Ravi, S.S., Rosenkrantz, D.J.: Simple heuristics for unit disk graphs. Networks 25, 59–68 (1995)
22. Herman, T., Ghosh, S.: Stabilizing phase-clocks. Information Processing Letters 54(5), 259–265 (1995)
23. Berge, C.: Theory of Graphs and its Applications. Methuen (1962)
24. Wu, W., Du, H., Jia, X., Li, Y., Huang, S.C.H.: Minimum connected dominating sets and maximal independent sets in unit disk graphs. Theoretical Computer Science 352(1), 1–7 (2006)
25. Arora, A., Gouda, M.: Distributed reset. IEEE Transactions on Computers 43, 1026–1038 (1994)
26. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems 8(4), 424–440 (1997)
27. Derhab, A., Badache, N.: A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile network. IEEE Transacctions on Parallel and Distributed Systems 19(7), 926–939 (2008)
28. Huang, T.C.: A self-stabilizing algorithm for the shortest path problem assuming the distributed demon. Computers and Mathematics with Applications 50(5-6), 671–681 (2005)
29. Hedetniemi, S.M., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. Computers and Mathematics with Applications 46(5-6), 805–811 (2003)

Leader Election in Extremely Unreliable Rings and Complete Networks

Stefan Dobrev[1], Rastislav Královič[2], and Dana Pardubská[2]

[1] Institute of Mathematics,
Slovak Academy of Sciences,
Bratislava, Slovakia
[2] Dept. of Computer Science
Comenius University,
Bratislava, Slovakia

Abstract. In this paper we investigate deterministic leader election under the *simple threshold* model of omission dynamic faults: The computation is performed in synchronous steps; if the algorithm sends m messages in one particular step, then at most $\max\{c_G - 1, m - 1\}$ of them may be lost without any notification to the sender, where c_G is the edge-connectivity of the communication topology. *Simple threshold* and related models of dynamic faults have been mainly used in the study of information dispersal (broadcasting), while fault-tolerant leader election has been primarily considered in models with static faults.

In this paper we combine these lines of research and present efficient algorithms for leader election on rings and complete networks in the *simple threshold* model. Somewhat surprisingly, obtaining *some* leader election is rather straightforward even in this harsh model. However, getting an *efficient* solution working in general case (arbitrary wake-up, unknown n) involves intricate techniques and careful accounting.

1 Introduction and Preliminaries

Fault tolerance has been one of the most intensively studied issues in distributed computing over many decades. Indeed, since the goal of the design of distributed systems is to develop highly scalable systems comprising huge amounts of processing elements, the fault of a small number of them is very likely even if extremely reliable devices are used. One of the prominent topics of research in the area is to study the solvability and complexity of communication problems, if various assumptions about the occurrence of faults are adopted. Obviously, if no restrictions are imposed on the occurrence of faults, then no nontrivial results can be obtained. On the other hand, it seems that finding reasonable restrictions on the environment that would at the same time allow non-trivial results is not an easy task.

We consider a point-to-point network of devices communicating via message passing. The simple threshold model of faults we adopt for this paper was introduced in [4]; it is a deterministic model in the sense that some a-priori restrictions

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 512–526, 2008.

on the occurrence of faults are considered, and worst case results are delivered. This is in contrast with probabilistic models, in which the faults occur with some probability distribution, and also the results have probabilistic nature. Next, our model considers only omission faults: a particular message may be lost, but not altered in any way, nor can the adversary introduce new messages. Also, our model is synchronous in the sense that the computation of the whole system is performed in steps synchronized by some global clock. The main feature of the model is *proportionality*: the number of messages that can be lost in a particular step depends on the number of messages that has been sent in that step.

The majority of research in the area of deterministic models of faults (static, dynamic, linearly bounded, fractional, threshold) has been focused on the broadcasting problem (e.g. [1,2,6,7,9,12,16] and many others).

Leader election problem is one of the most prominent coordination problems. All vertices start in the same state (differing only by having distinct identifiers), and the goal of the algorithm is to have exactly one vertex in a distinguished state. This problem has been studied on a number of topologies (e.g. [5,11,15,17] and many others), and also, to some extent in faulty environments, mostly considering static faults (e.g. [10,13,14,18]).

As the simple threshold model is rather punishing (only one message is guaranteed to be delivered, and only if sufficient number of messages has been sent in the first place), the question "Which problems can be solved at all under this model?" is rather pressing. In this paper we give a positive answer for the problem of leader election on bidirectional rings and complete networks. It turns out that while it is rather straightforward to obtain *some* results, novel techniques and intricate analysis were needed in order to obtain *efficient* algorithms that work under general assumptions (non synchronized start-up, size of the network is unknown).

In this paper we consider the leader election problem on bidirectional rings and complete graphs in the simple threshold model. We show upper bounds on the worst case number of time steps needed using various additional assumptions (synchronous start, sense of direction, etc). As a side-effect we improve the broadcasting time on complete graphs with sense of direction from $O(n^2)$ [4] to $O(n \log n)$ thus separating it from the lower bound $\Omega(n^2)$ on complete graphs without s.o.d. The results are summarized in Table 1.

The full version with all proofs included can be found online in [3].

1.1 Model

We consider a synchronous network of n processors communicating in a point-to-point manner in a topology described by a simple undirected graph G. The

Table 1. Summary of the results concerning the leader election problem

rings	simultaneous wakeup arbitrary wakeup	$O(n \log n)$ $O(n^2)$
complete graphs	with s.o.d without s.o.d	$O(n^2 \log n)$ $O(n^3)$

synchronization is performed by a global clock generating (non-numbered) ticks. The computation consists of a series of time steps: at the beginning of a step, each processor reads the incomming messages (possibly from many neighbors). Then every processor performs some local computation and sends messages to (possibly many of) its neighbors (at most one message per neighbor). After this action is performed by all processors (but before any of the messages is delivered) there is a set of m messages that are in transit in the whole networks. Some of these messages may be lost, and the step is conculded by a clock synchronization; i.e. the remaining messages are delivered at the beginning of the next time step.

The *simple threshold* model gives a (weak) restriction on the number of messages that can be lost: if there are m messages in transit after all processors finished sending during a particular time step then at most $\max\{c_G - 1, m - 1\}$ of them can be lost, where c_G is the edge connectivity of the underlying communication topology. In other words, if the algorithm tries to send less than c_G messages (over all processors) during a particular time step, all of them may be lost. Otherwise there is a guarantee that at least one message is delivered. Note, however, that the sender does not receive any kind of notification about which messages have been lost. Also, note that this is only a lower bound on the delivered messages. It may be the case that more messages are delivered.

For the specific case of ring networks we get that if there is only one message sent in a particular step, it may be lost. Otherwise (i.e. at least two messages are sent), at least one of them is delivered. For complete graphs with n vertices, at least $n - 1$ messages must be sent to guarantee the delivery of a single message.

2 Rings with Simultaneous Wake-Up

In this section we consider the case when the algorithm is started by k initiators that start at the same time step. Moreover, all other vertices can be awakened only by receiving a message. The main result of this section is the following theorem:

Theorem 1. *With simultaneous wake-up of k initiators, leader election on n-node rings can be solved in $O(n \log k)$ steps, even if the ring size is unknown to the processors.*

Proof: Our algorithm, SimultStart, is based on a combination of two ideas: a leader election algorithm that elects a leader in a ring (in the standard message-passing model) using $O(n \log k)$ messages, and the idea of *threads* from the broadcasting algorithm of [4].

We first recall a well-known algorithm due to Franklin[8] that elects a leader in a ring using $O(n \log k)$ messages. However, in order to be able to implement this algorithm in the simple threshold model, attention must be given to technical details. Therefore we present a modification of this algorithm in more detail and argue that the message complexity is asymptotically same as in the original Franklin's algorithm. Consider the following election algorithm in asynchronous rings (refer to Algorithm 1 for the handling at vertex w of messages originated from a vertex v): at the beginning, the initiators are *candidates*. Vertices that are not candidates are *defeated*, and they do not participate in the protocol apart from relying messages. A candidate vertex v has a phase number starting from zero. In one phase, v tries to capture at least one of its neighbors in the same phase; if successful, it enters the next phase, otherwise it keeps waiting until made defeated by some other message.

Obviously, the vertex with the highest $[\mathit{phase}, \mathit{id}]$ pair always wins, so there is no deadlock. Moreover, a vertex v can increase its phase only if it encounters some candidate vertex w with the same phase and lower id; this vertex then becomes defeated. Since a particular candidate w can serve as a victim for at most two candidates, at most two thirds of candidates in phase j can be promoted to phase $j+1$. This means that the number Q_j of candidates that reach phase j is at most $k(2/3)^j$, and $p \le 1 + \log_{3/2} k$ phases are sufficient to elect the leader. Note also that the total number of messages in any single phase is at most $3n$: An edge can be crossed by a normal message of phase j in each direction only once, plus possibly once more by a victory message of phase j. Therefore, the overall number of messages is $O(n \log k)$. In the rest of the proof we show how to implement this algorithm in the faulty environment.

Algorithm 1. modified Franklin's algorithm (messages of vertex v)

```
$\mathit{phase} := 0$, $\mathit{state} := \mathit{candidate}$
**loop**
    send messages to both directions using the protocol below
    wait for at least one “victory” answer
    **if** $\mathit{state} = \mathit{candidate}$ **then** $\mathit{phase} := \mathit{phase} + 1$ and continue loop
**end loop**
```

```
\\ *Messages of vertex $v$, being processed at vertex $w$:*
**if** two messages from vertex $v$ meet in a vertex $w$ without bouncing **then**
    $w$ knows the identity of the leader $v$, and launches a final phase
**end if**
**if** $\mathit{msg}$ arrives to a vertex with stored higher $[\mathit{phase}, \mathit{id}]$ pair **then**
    $\mathit{msg}$ dies
**else if** $\mathit{msg}$ arrives to a candidate vertex $w$ with the same phase and lower $\mathit{id}$ **then**
    $\mathit{state}$ of $w$ becomes $\mathit{defeated}$, $v$'s $[\mathit{phase}, \mathit{id}]$ pair is stored in $w$
    $\mathit{msg}$ bounces back as “victory”
**else**
    $\mathit{state}$ of $w$ becomes $\mathit{defeated}$, $v$'s $[\mathit{phase}, \mathit{id}]$ pair is stored in $w$
    $\mathit{msg}$ is relayed
**end if**
```

To implement the messages of Algorithm 1, we use the idea of *threads* from the broadcasting algorithm of [4], which we briefly describe here. Consider a vertex v that tries to send messages to both directions. At any moment of time, there are two chains of consecutive informed vertices, those to the left from v

and those to the right[1]; we call each such chain a thread. Each informed vertex in a thread tries to spread the message in the corresponding direction, until it receives an acknowledgement that the message was successfully delivered. Vertices trying to spread the message are *active*, vertices that have already received the acknowledgement are *passive* (w.r.t. the given thread). The computation is performed in *rounds*: the goal of a round is to ensure that at least one active vertex becomes passive. In [4], the round consists of the following four steps:

1. Each active vertex sends a *forward* message to its possibly uninformed neighbor.
2. Each active vertex in the right part sends a forward message to its possibly uninformed neighbor. Each vertex in the left part that received a message in step 1 replies to this message by sending an *ack* message.[2]
3. Same as step 2, but left and right parts are reversed.
4. Each vertex that received a non-reply message in steps 1–3 replies to that message.

The idea of this protocol is that, if no acknowledgement (an *ack* message) is delivered in the first three steps of a round, then there are two different *ack*s (and no other messages) being sent in step 4 and one of them must be delivered.

The messages in Algorithm 1 have the following structure: a vertex v sends two messages, to its left and right neighbor, and these messages potentially return back to v. To implement this, v starts one left and one right thread, and continues using the protocol above. As the thread spreads, it makes the vertices passive and marks them with the $[phase, id]$ pair of the initiator as in Algorithm 1. Once a growing thread reaches a candidate vertex of the same level, and with smaller identity, it bounces back as "victory", and continues. This bouncing back happens of course only in the front of each thread: the condition for bouncing back is evaluated only when a message arrived to a vertex not yet visited by this thread (as determined by the stored $[phase, id]$ pair). This way it is ensured that a delivered massage from an active vertex inside the thread never iniciates a victory message.

In any time, there may be many active threads; however, since vertices cannot wake up spontaneously, the rounds are synchronized over all threads. Hence, it is easy to see using similar arguments as in [4] that if, during a particular round, the following holds for every time step

1. There is at least one active vertex in a right thread and at least one active vertex in a left thread
2. There is at most one forward message sent over one edge in one direction
3. If there are two messages sent over one edge in opposite directions, they are of the same type (left or right)

Then at least one acknowledgement in some thread is delivered in this round. To see that the first requirement holds, consider the two threads (left and right)

[1] The vertices have no common orientation, so left, and right are local to the initiator.

[2] Note that passive vertices reply to such message, too.

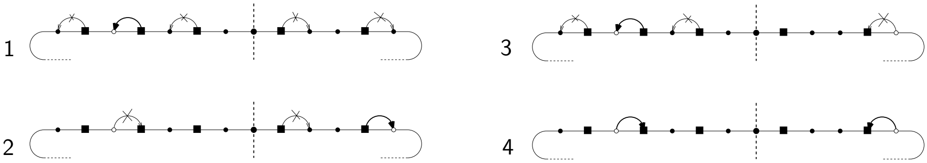


Fig. 1. A sample round of the broadcasting algorithm from [4]. The initially active vertices are squares. In the first step, the only surviving message is delivered to the white vertex on the left, which then replies in the next step. At the end of the round, two white vertices are sending acknowledgement, so at least one acknowledgement is delivered during that round.

with the highest $[phase, id]$ pair. They never die, so each of them has at least one active vertex. The second requirement is easily implemented: if there are more messages ready to be sent over an edge, only the one with the highest $[phase, id]$ pair is sent and all others are discarded. It is easy to see that the corresponding threads would die anyway. The third condition is always true, since there are two messages sent over one edge only when a thread bounces.

There is one exception to this, however, when there is a single candidate (the leader) remaining and its left and right threads meet. In this case, there are one or two vertices that know the identity of the leader. In the final phase, they both wait for a number of steps which is an upper bound on the running time of the algorithm. From the subsequent analysis it follows that after this time there is either a unique vertex knowing the leader's identity, or two such vertices that in addition know each other. In both cases, after the requested time elapses, there are no messages in progress, and the unique vertex (or the one with higher local id) can start a linear-time broadcast with the leader's id using the protocol from [4].

It is not possible to elect more than one leader, because all vertices contained in a thread (apart from the initiator) are defeated, and the leader is elected if two threads of the same vertex meet without bouncing, i.e. all vertices are contained in some thread.

Next, we argue the number of steps: each vertex v can get at most $O(3p)$ acknowledgements[3], because a passive vertex can become active only 3 times per phase: There can be one forward thread from each neighboring candidate, and possibly one bounced victory thread.

Hence, after $O(n \log k)$ acknowledgements are delivered, the algorithm terminates. As at least one acknowledgement is delivered during each 4-step round, the overall time is $O(n \log k)$ as well. □

It is tempting to argue the optimality of this result by a simple argument: comparison-based leader election in synchronous rings requires $\Omega(n \log n)$ messages in the worst case; if there is an adversary delivering only one message in each step, the algorithm must take $\Omega(n \log n)$ steps. However, this argument is wrong, because in our model, vertices can break symmetry not only by using the identifiers, but also by using the timing information. Indeed, if the

[3] where p is the number of phases.

adversary would let only one message in each step to be delivered, it is sufficient that all candidates send one message in the first step: one of them is delivered, so a vertex that receives a message in the second step is elected a leader and perform a broadcast to inform all other vertices about this. Hence, the number of messages delivered by the adversary seems to control the level of symmetry – if all messages are delivered, many symmetries can be forced, but the algorithm progresses fast; on the other hand, slowing the algorithm down by not delivering messages implicitly breaks symmetry.

3 Rings with Arbitrary Wake-Up

In this section we consider the scenario in which vertices can wake up at any time during the execution. The approach from the previous section does not work any more: The starting times of the candidates may be offset in such a way that there is no time step in which only acknowledgements are being sent, and hence no progress can be guaranteed. We show how to adapt the thread technique in order to overcome this problem, at a cost of increasing the time complexity to $O(n^2)$. We first present the algorithm for the case n known, and then explain how this assumption can be dropped. In order to maintain readability, we focus only on the asymptotic analysis, without any attempt to optimize the constant factors involved.

3.1 *N* Known

Consider any algorithm based on the thread approach. We can classify each time step as either *green* (if only acknowledgements are being sent during it), or *red* (at least one node tries to send a forward message).

The first idea is to inflate each round of a thread by appending it with $3n-3$ ack-steps (i.e. the whole round takes $3n+1$ steps). As each candidate tries to send a forward message only during three steps per round, this ensures that in each window of $3n+1$ steps there is at least one green step. Analogously as in the previous section (where the fourth step of each round was the green step) it follows that during each window of $3n+1$ steps at least one acknowledgement is delivered and the algorithm makes progress. As nothing else changes, the same arguments as in the previous section can be used to show correctness.

However, the overall complexity has increased to $O(n^2 \log n)$, since each round now takes $3n+1$ steps instead of 4.

The second idea is to speed up the rounds as more and more candidates are eliminated: A candidate in a higher phase knows that it has eliminated other candidates and can reduce the length of its round, using in essence the time slots of the candidates it eliminated to speed up transmission of its own messages. There are technical problems that have to be resolved, though: (1) A candidate is promoted to the next phase by a victory message coming from one direction. In the opposite direction there may still remain active threads of previous phases, potentially using additional time slots. (2) Similarly, when a candidate is defeated by an incoming message of a stronger candidate, its active threads in the opposite direction are not notified and would still consume time slots (we call those threads *dead tails*).

We solve the first problem by making all round lengths to be a power of two; the time slots of obsolete threads of lower phases are then a subset of the time slots used by highest-phase thread. The second problem is circumvented by choosing a sufficiently large[4] initial round size, so that even assuming the dead tails never die out there are still enough green steps remaining.

The algorithm N-Known for the case of arbitrary wake-up of initiators with n known differs from the algorithm SimultStart only by having vertices of phase j use rounds of length $d_j = n'/2^{\lfloor j/2 \rfloor}$, where n' is the smallest power of two larger than $90n$[5]. This means that as long as there is progress and the algorithm terminates, the correctness of algorithm N-Known follows from correctness of algorithm SimultStart.

The next two lemmas are crucial for proving the time complexity:

Lemma 1. *The number of red time steps during the whole execution of algorithm* N-Known *is at most $t_l/2 + 3n$, where t_l is the last time a progress has been made (an active vertex has turned passive).*

Proof: Consider a fixed execution. Let C denote the set of spontaneously awaken vertices (i.e. candidates). For each $v \in C$ let q_v be the highest phase reached by v and let t_v^0 be the wake-up time of v. Let q denote the highest phase reached by a vertex (i.e. by the leader). Let Q_j be the number of candidates that reached phase j. Note that $Q_j \leq n(2/3)^j$.

Each candidate v can be viewed as a periodic process that marks the time steps $t_v^0 + kd_{q_v}$, $t_v^0 + 1 + kd_{q_v}$ and $t_v^0 + 2 + kd_{q_v}$ as red, for $k = 0, 1, \ldots$ (i.e. during the whole execution). This greatly overestimates the time steps that will indeed become red due to v: In the beginning, v starts with a much longer period d_0, not d_{q_v}; towards the end most candidates are eliminated and do not contribute at all. Note that by ignoring this elimination, we account for the time slots used by the dead tails.

Since each vertex can be viewed as a periodic process (marking 3 steps per period), the total number R of red steps during the whole execution can be bounded as follows (we use the fact that the number of candidates is at most n):

$$R = \sum_{v \in C} 3 \left\lceil \frac{t_l}{d_{q_v}} \right\rceil \leq 3n + \sum_{v \in C} \frac{3t_l}{d_{q_v}} = 3n + t_l \sum_{v \in C} \frac{3}{d_{q_v}} \tag{1}$$

Let us denote the term $\sum_{v \in C} \frac{3}{d_{q_v}}$ as H, and call it *density* of the red steps. Hence, we can rewrite (1) as:

$$R = 3n + t_l H \tag{2}$$

[4] It follows from the subsequent analysis that $90n$ *is* large enough. We did not attempt to optimize this constant.

[5] See the previous footnote. Note that $n' < 180n$.

The *density* H can be bounded as follows:

$$H = \sum_{v\in C} \frac{3}{d_{q_v}} = \sum_{j=0}^{q}\left(\sum_{v:q_v=j}\frac{3}{d_{q_v}}\right) \le \sum_{j=0}^{q}\left(\sum_{v:q_v\ge j}\frac{3}{d_{q_v}}\right) = \sum_{j=0}^{q}\frac{3Q_j}{d_j}$$

As $\sum_{i=0}^{\infty}(\frac{2}{3})^i 2^{\lfloor \frac{i}{2}\rfloor} = 15$, using $Q_j \le n\left(\frac{2}{3}\right)^j$ and substituting for d_j we get

$$H \le \sum_{j=0}^{q} 3n\left(\frac{2}{3}\right)^j \frac{2^{\lfloor\frac{j}{2}\rfloor}}{n'} \le \sum_{j=0}^{q}\frac{3n\left(\frac{2}{3}\right)^j 2^{\lfloor\frac{j}{2}\rfloor}}{90n} \le \frac{1}{30}\sum_{j=0}^{\infty}\left(\frac{2}{3}\right)^j 2^{\lfloor\frac{j}{2}\rfloor}\frac{1}{2}$$

Substituting for H in (2) we obtain $R \le 3n + t_l/2$. Moreover, it holds that $H \le 1/2$ which leads to $R \le 3n + t_l/2$. □

Lemma 2. *Consider an arbitrary execution of the algorithm* N-Known. *Let t_i for $i = 1, 2, \ldots, l$ denote the times when a progress has been made – an active vertex v_i of phase p_i has turned passive. Set $t'_i = t_i + d_{p_i}$ and t''_i be the first green step occurring later then $t'_i + 2$. Then $t_{i+1} \le t''_i$.*

Proof: Assume the contrary, i.e. no progress (a delivered ack message different from the one that caused i-th progress) has been made by the time t''_i. First, note that the current round of v_i is finished by time t'_i at the latest. If no progress has been made up to that moment, in the next three steps either an ack message, or two different forward messages (and in different directions) are delivered (using the same arguments as in the proof of progress of the thread technique). Therefore, at time t''_i at least two ack messages will be sent and the adversary must deliver one of them. □

Note that l is bounded from above by the total number of messages of the underlying modified Franklin's algorithm and is therefore $O(n \log n)$.

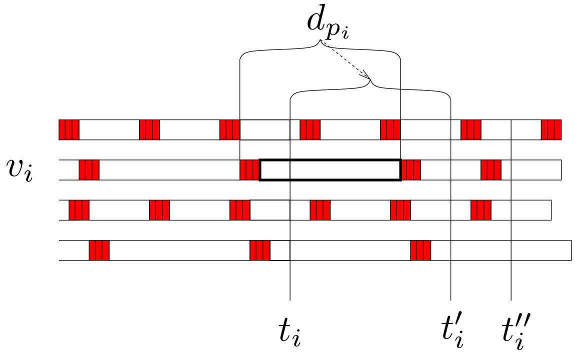


Fig. 2. Illustration of Lemma 2 which assures that progress is made often enough. The x-axis represents time steps; the strips are different processes. Highlighted are red time steps generated by a given process. Vertex v_i uses rounds of length d_{p_i}. If v_i makes progress in step t_i, $t'_i = t_i + d_{p_i}$, and t''_i is the frist green step after $t'_i + 2$, Lemma 2 states that the next time a progress is made is no later that t''_i.

Theorem 2. *With arbitrary wake-up, and the size n of the ring network known to all vertices, leader election can be solved in $O(n^2)$ steps.*

Proof: Recall that t_l is the time when all progresses that are possible have already been done and the leader has been identified, with only the final broadcast remaining. We are going to estimate t_l, as the total execution time is asymptotically the same. Using the notation t_i, t'_i, t''_i from Lemma 2 let $s_i = \min(t_{i+1} - t_i, t'_i - t_i)$ and $s'_i = \max(0, t_{i+1} - t'_i)$. Let $S = \sum_{i=1}^{l-1} s_i$ and $S' = \sum_{i=1}^{l-1} s'_i$. Informally, S corresponds to the overall length of time steps during which the adversary can block progress by delivering the same ack message again and again, while S' corresponds to the overall length time steps wasted due to the adversary preventing the delivery of a new ack message by delivering only forward messages.

By definition, $t_{i+1} - t_i = s_i + s'_i$, therefore $t_l = t_l - t_0 = \sum_{i=1}^{l-1}(t_{i+1} - t_i) = \sum_{i=1}^{l-1}(s_i + s'_i) = S + S'$. Note that (by Lemma 2 and the definition of t''_i) all but the first three steps of the period corresponding to any s'_i are red. Using Lemma 1 we get $S' \leq 3l + t_l/2 + 3n$. S can be estimated as follows:

$$S = \sum_{i=1}^{l-1} s_i \leq \sum_{i=1}^{l-1} d_{p_i} = \sum_{j=0}^{q} \left(\sum_{p_i=j} \frac{n'}{2^{\lfloor p_i/2 \rfloor}} \right)$$

As the number of progresses in each phase is bounded by $4n$ and $n' < 180n$ (recall that n' is the smallest power of 2 not smaller than $90n$), we get $S = O(n^2)$. Summing together: $t_l = S + S' \leq S + 3l + t_l/2 + 3n$; separating t_l gives $t_l \leq 2S + 6l + 6n$, which together with $l \in O(n \log n)$ yields $t_l \in O(n^2)$. □

3.2 *n* Unknown

Algorithm N-UnKnown is based on algorithm N-Known, but this time the vertices need to "guess" a good enough estimate of n. A vertex v, upon wake-up, starts with a constant estimate n'_0. From the proof of Theorem 2 it follows that if $n \leq n'_0$, the algorithm explicitly terminates in at most $c(n'_0)^2$ steps for some fixed constant c. Hence, if after $c(n'_0)^2$ steps the vertex v does not know the leader, it increases its estimate by a factor of 2 (all messages can carry a time in which their originator started, so all vertices participating in v's transmission can increase the estimate of n, and hence the length of a round, synchronously). This doubling of estimates is repeated afterwards until the leader is elected.

Once the estimates of all candidates are larger than n', the algorithm will terminate within $O(nm)$ steps, where m is the largest estimate (follows from the proof of Theorem 2). However, few candidates with low estimates are sufficient to keep the frequency of red steps high and prevent progress, forcing even the candidates with large estimates to further increase their estimates and hence slow down the overall progress. A naive argument (there are n candidates and after wake-up, each of them will reach a correct estimate in $O(n^2)$ steps) results in $O(n^3)$ overall time. In the rest, we will show that the algorithm terminates in time $O(n^2)$.

Consider a vertex v at time t. Let n_v^t and p_v^t denote the estimate of n' used by v and the phase of v at time t, respectively. The round length d_v^t vertex v uses at time t is therefore

$$d_v^t = \frac{n_v^t}{2^{\left\lfloor \frac{p_v^t}{2} \right\rfloor}}$$

Define the *instant density* h_v^t to be 0 if the vertex has not awaken yet, and $3/d_v^t$ otherwise. Define the *overall instant density* at time t as $H^t = \sum_{v \in C} h_v^t$. Note that H^t reflects the density of red steps, overestimating them as it does not take into account that the vertices are being eliminated.

Let us classify all time steps t with $H^t > 1/2$ as *dense*, and all steps in which h_v^t for some v has increased as *special dense*. All other steps are *sparse*. The *special dense* steps are introduced for technical reasons to simplify argumentation. As h_v^t increases only when a vertex spontaneously awakes (this can happen at most n times over all vertices), or when a vertex increases its phase (at most $2n$ times overall), the number of special dense steps is insignificant:

Fact 1. *The number of special dense steps is at most* $3n$.

In order to prove the $O(n^2)$ time complexity, we show that there are $O(n^2)$ green steps, and they comprise a constant fraction of all steps. First, we show that the fraction of dense steps is at most 1/2. Next, we show that in a long enough (at least $12n$) segment of sparse steps, the fraction of green steps is at least 1/4. Finally, we show that the overall length of short segments of sparse steps is short enough.

Let us now estimate the number of dense steps. The idea is to upper-bound the instant densities of the vertices by a well-behaved function; the maximal number of dense steps is then at most twice the area below this function (since each dense step has $H^t > 1/2$) plus the number of special dense steps.

Lemma 3. *Let t_v^0 be the wake-up time of vertex v and let q_v be the maximal phase reached by vertex v. Define*

$$y_v^t = 2^{\lfloor \frac{q_v}{2} \rfloor} \frac{\sqrt{3c}}{\sqrt{t - t_v^0}}$$

for $t \geq t_v^0 + 1$, 0 otherwise, where c is a constant such that the algorithm N-Known *terminates within cn^2 steps.*

Then it holds $\forall t : h_v^t \leq y_v^t$.

Proof: Consider the times t_v^k when v doubles its estimate of n for the k-th time. According to the algorithm,

$$t_v^k = t_v^0 + \sum_{i=0}^{k-1} c(2^k n_0')^2 = t_v^0 + c(n_0')^2 \sum_{i=0}^{k-1} 4^k \leq \frac{4^k}{3} c(n_0')^2 \qquad (3)$$

Let us simplify the notation and use k instead of t_v^k as the time index for the rest of the proof. As q_v is the maximal phase of v, we have

$$h_v^k = \frac{3}{d_v^k} \leq 3\frac{2^{\lfloor \frac{q_v}{2} \rfloor}}{n_v^k} \leq 3\frac{2^{\lfloor \frac{q_v}{2} \rfloor}}{2^k n_0'} \qquad (4)$$

Separating n'_0 from (3), and substituting it into (4) yields the result. □

Summing up over all vertices we obtain:

Corollary 1. $\forall t : H^t \leq \sum_{v \in C} y_v^t = Y^t$

Now we are ready to bound the number of dense steps:

Lemma 4. *There is a constant a such that during the first an^2 time steps, the number of dense time steps is at most $an^2/2$.*

Proof: Let us compute the total area A below Y^t. From Corollary 1, from the definition of dense steps and from Fact 1 follows that the number of dense steps is bounded by $2A + 3n$.

Let us estimate the area A_v below y_v^t for a vertex that reached phase q_v:

$$A_v = \int_{t_v^0+1}^{an^2} y_v^t dt < \int_1^{an^2 - t_v^0 - 1} y_v^{t+t_v^0} dt < \int_1^{an^2} y_v^{t+t_v^0} dt =$$

$$= \left[2^{\lfloor \frac{q_v}{2} \rfloor - 1} \sqrt{3ct} \right]_1^{an^2} < 2^{\lfloor \frac{q_v}{2} \rfloor - 1} \sqrt{3can^2} = (2^{\lfloor \frac{q_v}{2} \rfloor - 1} \sqrt{3ca}) n$$

Note that this bound depends on q_v, but does not depend on v itself; let us thus denote $A^j = (2^{\lfloor \frac{j}{2} \rfloor - 1} \sqrt{3ca}) n$. Clearly, for all v such that $q_v = j$ it holds $A^j \geq A_v$. Let us now bound the total area A:

$$A = \sum_{v \in C} A_v = \sum_{j=0}^{p} \left(\sum_{v : q_v = j} A_v \right) \leq \sum_{j=0}^{p} Q_j A^j \leq \sum_{j=0}^{p} n \left(\frac{2}{3} \right)^j (2^{\lfloor \frac{j}{2} \rfloor - 1} \sqrt{3ca}) n <$$

$$< \frac{n^2 \sqrt{3ca}}{2} \sum_{j=0}^{p} \left(\frac{2}{3} \right)^j 2^{\lfloor \frac{j}{2} \rfloor} < \frac{15}{2} n^2 \sqrt{3ca}.$$

The number of dense steps before the time an^2 can thus be bounded by $15\sqrt{3ca}n^2 + 3n$. By requiring $15\sqrt{3ca}n^2 + 3n \leq an^2/2$ and separating a we obtain $\sqrt{a} \geq 30\sqrt{3c}$ and therefore $a \geq 2700c$. □

The following Lemma is an analogue of Lemma 1:

Lemma 5. *Consider a segment T of sparse steps $t_b, t_b + 1, \ldots, t_e$. Then the number of red steps within T is at most $(t_e - t_b)/2 + 3n$.*

Sketch of the proof: First note that since T does not contain special dense steps, each h_v^t is non-increasing on T (and the same applies to H^t as well). The number of red steps during T can be estimated analogously as in Lemma 1. □

The following counterpart of Lemma 2 holds also for algorithm N-UnKnown; the proof is essentially the same as the proof of Lemma 2:

Lemma 6. *Consider an arbitrary execution of the algorithm* N-UnKnown. *Let t_i for $i = 1, 2, \ldots, l$ denote the times when a progress has been made – an active vertex v_i using a round size $d_v^{t_i}$ has turned passive. Set $t'_i = t_i + d_v^{t_i}$ and t''_i be the first green step occurring later then $t'_i + 2$. Then $t_{i+1} \leq t''_i$.*

Now we are ready for the main result:

Theorem 3. *With arbitrary wake-up, leader election in rings of unknown size n can be solved in $O(n^2)$ steps.*

Proof: Let s_i, s'_i, S and S' be defined as in the proof of Theorem 2, i.e. the total execution time t_l can be expressed as $S + S'$. We will show that $S + S' \leq an^2$, i.e. that the algorithm terminates before the time an^2, where a is the constant from Lemma 6. Let us estimate S'. We may assume that all but the first three steps of each period corresponding to a s'_i are either dense, or sparse but red. We include into S' also the short (of length less then $12n$) sparse segments, as for those Lemma 5 does not guarantee the fraction of red vertices to be below 3/4. By Lemma 4 the number of dense steps is at most $an^2/2$. Using Fact 1 the total number of the short sparse segments is at most $3n \times 12n = 36n^2$. Applying Lemma 5 to the remaining sparse segments (of total length $(a - a/2 - 36)n^2$) yields additional $\frac{3}{4}(a/2 - 36)n^2$ red steps. Summing together we get $S' \leq 3l + an^2/2 + 36n^2 + (3a/8 - 27)n^2 = 3l + (7a/8 + 9)n^2$. S can be estimated similarly as in Theorem 2 to be at most $16\sqrt{a}n^2$, using the fact that for the maximal estimate of n used by the vertices, m, it holds $\frac{4c}{3}(m)^2 < an^2$, as there is not enough time for the estimate to grow bigger. Summing together yields $S + S' < 16\sqrt{a}n^2 + 3l + (7a/8 + 27)n^2 = 3l + (16\sqrt{a} + 7a/8 + 27)n^2 < an^2$ for sufficiently large n. □

4 Complete Graphs

The broadcasting algorithms from [4] can be adapted (essentially by running n broadcasts in parallel, and having the strongest one survive) to leader election, at a cost of additional multiplicative factor of n. This results in $O(n^4)$ algorithm for leader election in unoriented complete graphs, and $O(n^3)$ leader election in complete graphs oriented with chordal sense direction [6].

The insight we gained in Section 3 allows us to improve upon the broadcasting algorithm from [4]

Theorem 4. *With chordal sense of direction, broadcasting can be done in $O(n \log n)$ time.*

which implies an improved leader election as well:

Corollary 2. *There is an $O(n^2 \log n)$ leader election algorithm in complete graphs with chordal sense of direction.*

With maximal use of collected information and careful accounting we are able to improve the case without sense of direction as well:

Theorem 5. *There is an $O(n^3)$ leader election algorithm in complete graphs without sense of direction.*

[6] Select an arbitrary Hamiltonian cycle, both endpoints of each edge are marked by how far ahead in the cycle is the other endpoint.

5 Conclusion

We have shown that despite the rather harsh nature of the simple threshold model, leader election (on bidirectional rings and complete networks) is not only possible, but can be also quite efficient. Interestingly, unlike the fault-free case, whether the candidates start simultaneously or not significantly influences the overall time. While the algorithms are relatively simple, the analysis is rather involved and for the case of arbitrary wake-up does not yield practical constant factors. The first question is whether the analysis can be improved (perhaps also changing the algorithms) to yield practical constant factors for the case of arbitrary wake-up.

An obvious challenge is to find non-trivial lower bounds. The problem is more difficult that it might seem on a first glance, as the timing information is potentially there for the algorithm to use. Even the case of $\Omega(n \log n)$ lower bound for simultaneous wake-up is still open.

More generally, we view this result as a first step into the exploration of more complex problems in the threshold model of dynamic faults. While broadcasting has been investigated before, even such basic problems as leader election in complete networks or arbitrary graphs are still widely open.

References

1. Chlebus, B., Diks, K., Pelc, A.: Broadcasting in synchronous networks with dynamic faults. Networks 27 (1996)
2. Chlebus, B.S., Diks, K., Pelc, A.: Optimal broadcasting in faulty hypercubes. In: FTCS, pp. 266–273 (1991)
3. Dobrev, S., Kralovic, R., Pardubska, D.: Leader election in extremely unreliable rings and complete networks. Technical report of Faculty of Mathematics, Physics, and Informatics, Comenius University, Bratislava, Slovakia, TR-2008-016 (2008), http://kedrigern.dcs.fmph.uniba.sk/reports/display.php?id=16
4. Dobrev, S., Královič, R., Královič, R., Santoro, N.: On fractional dynamic faults with threshold. In: Flocchini, P., Gasieniec, L. (eds.) SIROCCO 2006. LNCS, vol. 4056, pp. 197–211. Springer, Heidelberg (2006)
5. Dobrev, S., Pelc, A.: Leader election in rings with nonunique labels. Fundam. Inform. 59(4), 333–347 (2004)
6. Dobrev, S., Vrťo, I.: Optimal broadcasting in hypercubes with dynamic faults. Information Processing Letters 71(2), 81–85 (1999)
7. Dobrev, S., Vrťo, I.: Optimal broadcasting in tori with dynamic faults. Parallel Processing Letters 12(1), 17–22 (2002)
8. Franklin, R.: On an improved algorithm for decentralized extrema finding in circular configurations of processors. Commun. ACM 25(5), 336–337 (1982)
9. Gasieniec, L., Pelc, A.: Broadcasting with linearly bounded transmission faults. Discrete Applied Mathematics 83(1–3), 121–133 (1998)
10. Goldreich, O., Shrira, L.: Electing a leader in a ring with link failures. Acta Inf. 24(1), 79–91 (1987)
11. Hromkovic, J., Klasing, R., Pelc, A., Ruzicka, P., Unger, W.: Dissemination of Information in Communication Networks: Broadcasting, Gossiping, Leader Election, and Fault-Tolerance. Texts in Theoretical Computer Science. An EATCS Series. Springer, New York (2005)

12. Liptak, Z., Nickelsen, A.: Broadcasting in complete networks with dynamic edge faults. In: Butelle, F. (ed.) OPODIS 2000, Studia Informatica Universalis, pp. 123–142. Suger, France (2000)
13. Mans, B., Santoro, N.: Optimal fault-tolerant leader election in chordal rings. In: FTCS, pp. 392–401 (1994)
14. Mans, B., Santoro, N.: Optimal elections in faulty loop networks and applications. IEEE Trans. Computers 47(3), 286–297 (1998)
15. Marchetti-Spaccamela, A.: New protocols for the election of a leader in a ring. Theor. Comput. Sci. 54, 53–64 (1987)
16. Marco, G.D., Vaccaro, U.: Broadcasting in hypercubes and star graphs with dynamic faults. Information Processing Letters 66(6), 321–326 (1998)
17. Peleg, D.: Time-optimal leader election in general networks. J. Parallel Distrib. Comput. 8(1), 96–99 (1990)
18. Singh, G.: Leader election in the presence of link failures: IEEE Transactions on Parallel and Distributed Systems 7(3), 231–236 (1996)

Toward a Theory of Input Acceptance for Transactional Memories*

Vincent Gramoli[1,2], Derin Harmanci[2], and Pascal Felber[2]

[1] School of Computer and Communication Sciences, EPFL, Switzerland
vincent.gramoli@epfl.ch
[2] University of Neuchâtel, CH-2009, Switzerland
{derin.harmanci,pascal.felber}@unine.ch

1 Introduction

Transactional memory (TM) systems receive as an input a stream of events also known as a *workload*, reschedule it with respect to several constraints, and output a consistent history. In multicore architectures, the transactional code executed by a processor is a stream of events whose interruption would waste processor cycles. In this paper, we formalize the notion of TM workload into classes of input patterns, whose acceptance helps understanding the performance of a given TM.

TMs are often evaluated in terms of throughput (number of commits by time unit). The performance limitation induced by aborted transactions has, however, been mostly neglected. A TM optimistically executes a transaction and commits it if no conflict has been detected during its execution. If there exists any risk that a transaction violates consistency, then this transaction does not commit. Since stopping a thread until possible conflict resolution would waste core cycles, the common solution is to choose one of the conflicting transactions and to abort it.

Interestingly, many existing TMs unnecessarily abort transactions that could commit without violating consistency. For example, consider the input pattern depicted on the left-hand side of Figure 1, whose events are ordered from top to bottom. DSTM [1], a well-known Software Transactional Memory (STM), would detect a conflict and try to resolve it, whatever it costs. Clearly, the read operation applied to variable x could indifferently return value v_1 or the value overwritten without violating serializability [2], or even opacity [3], thus no conflict resolution is needed. This paper focuses on the ability of TMs to commit transactions from given workloads: the input acceptance of TMs.

Contributions. In this paper, we upper-bound the input acceptance of existing TMs by grouping them into the following designs. *(i)* Visible write (VWIR); *(ii)* Invisible write (IWIR); *(iii)* Commit-time relaxation (CTR); *(iv)* Real-time relaxation (RTR). We propose a Serializable STM, namely *SSTM*, that implements the last design in a fully decentralized fashion. Finally, we compare the

* Corresponding author: Vincent Gramoli, EPFL-IC-LPD, Station 14, CH-1015 Lausanne, Switzerland; fax: +41 21 693 7570. This work is supported by the Velox Project and the Swiss National Foundation Grant 200021-118043.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 527–533, 2008.

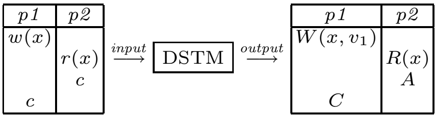


Fig. 1. A simple input pattern for which DSTM produces a commit-abort ratio of $\tau = 0.5$ (e.g., transaction of *p2* aborts with contention manager Polite that kills the transaction detecting the conflict)

four TM designs based on the upper-bound of their input acceptance. We validate our theoretical comparison experimentally under realistic workloads.

Related Work. The question whether a set of input transactions can be accepted without being rescheduled has already been studied by Yannakakis [4]. In contrast here, we especially concentrate on TMs where some operation requests must be treated immediately for efficiency reasons. Some STMs present desirable features that we also target in this paper. All these STMs relax a requirement common to opacity and linearizability to accept a wider set of workloads. As far as we know SSTM is, however, the first of these STMs that is fully decentralized and ensure serializability. CS-STM [5] is decentralized but is not serializable. Existing serializable STMs require either centralized parameters [6] or a global reader table [7] to minimize the number of aborting transactions.

2 Model and Definitions

This section formalizes the notions of workload and history as TM input and TM output, respectively. In this model, we assume that all accesses are executed inside a transaction, each thread executes one transaction at a time, and when a transaction aborts it must be retried later—the retried transaction is then considered as a distinct one.

First, we formalize the workload as the TM input that contains a series of events in some transaction t. These *input events* are a start request, s_t, an operation call on variable x, $\pi(x)_t$ (either a read $r(x)_t$ / ${r^x}_t$ or a write $w(x)_t$ / ${w^x}_t$) or a commit request c_t. We refer to an *input pattern* $\mathcal{P}$ of a TM as a sequence of input events. The sequence order corresponds intuitively to the real-time order, and for the sake of simplicity we assume that no two distinct events occur at the same time. An input pattern is *well-formed* if each event $\pi(x)_t$ of this pattern is preceded by a unique s_t and followed by a unique c_t. Second, we define TM output as the classical notion of history. This history is produced by the TM as a result of a given input. An *output event* is a complete read or write operation, a commit, or an abort. We refer to the complete read operation of transaction t that accesses shared variable x and returns value v_0, as $R(x)_t : v_0$. Similarly, we refer to a complete write operation of t writing value v_1 on variable x as $W(x, v_1)_t$. We refer to C and A as a commit and abort, respectively.

A *history* H of a transactional memory is a pair $\langle O, \prec\rangle$ where O is a set of output events and $\prec$ is a total order defined over O. Two operations π_1 and π_2

conflict if and only if *(i)* they are part of different transactions, *(ii)* they access the same variable x, and *(iii)* at least one of them is a write operation.

We use regular expressions to represent the possible input patterns of a class. In our regular expressions, parentheses, '(' and ')', are used to group a set of events. The star notation, '$*$', indicates the Kleene closure and applies to the preceding set of events. The complement operator, '$\neg$', indicates any event except the following set. Finally, the choice notation, '$|$', denotes the occurrence of either the preceding or the following set of events. Operators are ordered by priority as $\neg, *, |$.

The *commit-abort ratio*, denoted by τ, is the ratio of the number of committing transactions over the total number of complete transactions (committed or aborted). The commit-abort ratio is an important measure of "achievable concurrency" for TM performance. In the remainder of the paper, we say that a TM *accepts* an input pattern if it commits all of its transactions, i.e., $\tau = 1$.

3 On the Input Acceptance of Existing TMs

This section identifies several TM designs and upper-bounds their input acceptance. All the designs considered here use contention manager Polite, i.e., a transaction resolves a conflict by aborting itself.

VWIR Design. The first design is similar to DSTM [1] and TinySTM [8]. If a read request is input, the TM records locally the opened read variable, thus, the set of variables read is visible only to the current thread. Conversely, the write operations are made visible in that when a write request is input the updating transaction registers itself in $x.writer$. The limitations of this design are shown by giving a class of common inputs that it never accepts. For instance, the input pattern depicted in Figure 1 may arise when concurrent operations (searches, insertions) are executed on a linked list. All proofs are deferred to the technical report [9].

Theorem 1. *There is no TM implementing VWIR that accepts any input pattern of the following class:* $\mathcal{C}1 = \pi^*(r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \mid w_j^x \neg c_j^* r_i^x)\pi^*$.

IWIR Design. Here, we outline a second design that accepts patterns of the preceding class, i.e., for which the previous impossibility result does not hold. This design, inspired by WSTM [10] and TL2 [11], uses invisible writes and invisible reads with a lazy acquire technique that postpones effects until commit-time, thus it is called *IWIR*. Even IWIR design does not accept some very common input patterns. Assume that a transaction t_2 writes a variable x and commits after another transaction t_1 reads x but before t_1 commits. Because t_1 has read the previous value, it fails its validation at commit-time and aborts. Such a pattern also arises when performing concurrent operations on a linked list. The following theorem gives a set of input patterns that are not accepted by STMs of the IWIR design.

Theorem 2. *There is no TM implementing IWIR that accepts any input pattern of the following class:* $\mathcal{C}2 = \pi^*(r_i^x \neg c_i^* w_j^x \mid w_j^x \neg c_j^* r_i^x) \neg c_i^* c_j \pi^*$.

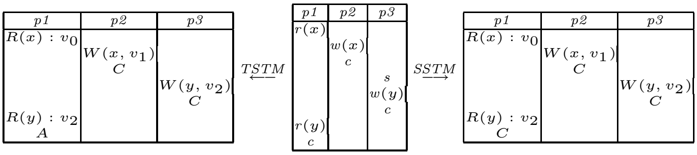


Fig. 2. An input pattern (in the center) that TSTM does not accept as described on the left-hand side. The commit-abort ratio obtained for TSTM is $\tau = \frac{2}{3}$ (transactions of *p2* and *p3* commit but transaction of *p1* aborts with Polite). In contrast, the Serializable STM presented in Section 4) accepts it (the output of SSTM, on the right-hand side, shows a commit-abort ratio of 1).

CTR Design. The following design has, at its core, a technique that makes as if the commit occurred earlier than the time the commit request was received. In this sense, this design relaxes the commit time and we call it *Commit-Time Relaxation (CTR).* To this end, the TM uses scalar clocks that determine the serialization order of transactions. This design is inspired by the recently proposed TSTM [7] in its single-version mode.

TSTM is claimed to achieve conflict-serializability, however, it does not accept all possible conflict-serializations. Figure 2 (center and left-hand side) presents an input pattern that TSTM does not accept since transactions choose their clock depending on the last committed version of the object they access: in this example, transactions of $p2$ and $p3$ choose the same clock and force transaction of $p1$ to abort. This pattern typically happens when a long transaction t runs concurrently with short transactions that update the variables read by t. The following theorem generalizes this result by showing that STMs implementing CTR design does not accept a new input class.

Theorem 3. *There is no TM implementing CTR that accepts any input pattern of the following class:*

$$\mathcal{C}3 = (\neg w^x)^* r_i^x \neg c_i^* w_j^x \neg c_i^* c_j \neg c_i^* s_k \neg(c_i \mid c_k \mid r_k^x)^* w_k^y \neg(c_i \mid c_k \mid r_k^x)^* c_k \neg c_i^* r_i^y \pi^*.$$

Observe that we use the notation s_k in this class definition to prevent transactions t_j and t_k from being concurrent.

4 SSTM, an Implementation of the RTR Design

We propose a new design, the *Real-Time Relaxation (RTR)* design, that relaxes the real-time order requirement. The real-time order requires that given two transactions t_1 and t_2, if t_1 ends before t_2 starts, then t_1 must be ordered before t_2. The design presented here outputs only serializable histories but does not preserve real-time order.

SSTM, standing for *Serializable STM*, implements the RTR design and presents a high input acceptance. Moreover, SSTM is conflict-serializable but not opaque (SSTM accepts a history that is not opaque as illustrated on the right-hand side of Figure 2) and it avoids cascading abort, since whenever a transaction t_1 reads a value from another transaction t_2, t_2 has already committed [12]. Finally, SSTM is also fully decentralized, i.e., it does not use global parameters as opposed to other serializable STMs [6,7] that may experience congestion when scaling to large numbers of cores. Figure 1 presents the pseudocode of SSTM. For the sake of clarity of the presentation, we assume in the pseudocode of the algorithm that each function is atomic and we do not specify how shared variables are updated. We refer to T, X, V, as the sets of transaction identifiers, variable identifiers, and variable values, respectively.

Algorithm 1. SSTM – Serializable STM

```
State of transaction $t$:
    $status \in$ {active, inactive}, initially active
    $read\text{-}set \subset X$, initially $\emptyset$
    $write\text{-}set \subset X \times V$, initially $\emptyset$
    $invisible\text{-}reads \subset X$, initially $\emptyset$
    $cr \subset T$, initially $\emptyset$ // set of concurrent readers

State of variable $x$:
    $read\text{-}fc \subset T$, initially $\emptyset$ // read future conflicts
    $write\text{-}fc \subset T$, initially $\emptyset$ // write future conflicts
    $active\text{-}readers \subset T$, initially $\emptyset$
    $val \in V$, initially the default value

commit()$_t$:
    for all $x \in read\text{-}set$ do
        $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{t\}$
    for all $\langle x, t' \rangle$ such that $x \in read\text{-}set \wedge t' \in x.write\text{-}fc \vee \langle x, * \rangle \in write\text{-}set \wedge t' \in x.write\text{-}fc \cup x.read\text{-}fc$ do
        for all $r' \in t'.cr$ do
            if $r'.status =$ inactive then
                $t'.cr \leftarrow t'.cr \setminus \{r'\}$
                if $t'.cr = \emptyset$ then
                    $x.write\text{-}fc \leftarrow x.write\text{-}fc \setminus \{t'\}$
                    $x.read\text{-}fc \leftarrow x.read\text{-}fc \setminus \{t'\}$
            else if $r' = t$ then abort()
            else $cr \leftarrow cr \cup \{r'\}$
    $status \leftarrow$ inactive
    for all $\langle x, * \rangle \in write\text{-}set$ do
        for all $r \in x.active\text{-}readers$ do
            $cr \leftarrow cr \cup \{r\}$
            $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{r\}$
        $x.write\text{-}fc \leftarrow x.write\text{-}fc \cup \{t\}$
    for all $\langle x, v \rangle \in write\text{-}set$ do
        $x.val \leftarrow v$
    clean()

write$(x, v)_t$:
    $write\text{-}set \leftarrow (write\text{-}set \setminus \{\langle x, * \rangle\}) \cup \{\langle x, v \rangle\}$

read$(x)_t$:
    if $\langle x, v' \rangle \in write\text{-}set$ then
        $read\text{-}set \leftarrow read\text{-}set \cup \{x\}$
        $v \leftarrow v'$
    else
        $invisible\text{-}reads \leftarrow invisible\text{-}reads \cup \{x\}$
        $x.active\text{-}readers \leftarrow x.active\text{-}readers \cup \{t\}$
        for all $t'$ in $x.write\text{-}fc$ do
            for all $r' \in t'.cr \wedge r'.status =$ active do
                if $r' = t$ then abort()
                else
                    $cr \leftarrow cr \cup \{r'\}$
                    $x.read\text{-}fc \leftarrow x.read\text{-}fc \cup \{t\}$
        $v \leftarrow x.val$
    return $v$

abort()$_t$:
    $status \leftarrow$ inactive
    clean()

clean()$_t$:
    for all $y \in invisible\text{-}reads$ do
        $y.active\text{-}readers \leftarrow y.active\text{-}readers \setminus \{t\}$
    for all $x$ such that $\langle x, * \rangle \in write\text{-}set$ or $x \in read\text{-}set$ do
        for all $t' \in x.write\text{-}fc \cup x.read\text{-}fc$ do
            for all $r' \in t'.cr$ do
                if $r'.status =$ inactive then
                    $t'.cr \leftarrow t'.cr \setminus \{r'\}$
                    if $t'.cr = \emptyset$ then
                        $x.write\text{-}fc \leftarrow x.write\text{-}fc \setminus \{t'\}$
                        $x.read\text{-}fc \leftarrow x.read\text{-}fc \setminus \{t'\}$
```

During the execution of SSTM, a transaction records the accessed variables locally and registers itself as a potentially future conflicting transaction in the accessed variables. These records help SSTM keeping track of all potential conflicts. More precisely, a transaction t accessing variable x keeps track of all transactions that may both precede it and follow it. Only transactions that read and that are concurrent with t (namely, the concurrent readers of t) can both precede and follow t. This is due to invisible writes that can only be observed by other transactions after commit. When detected, the preceding transactions are recorded in $t.cr$ (the concurrent readers of transaction t). Transaction t detects those transactions either because they are in *x.active-readers* (Line 27) or because they precede a transaction t' that is in *x.write-fc* (the future conflicts caused by a write access to object x) and they appear in $t'.cr$ (Line 23).Instead of keeping track of the following transactions, transaction t makes sure that any transaction detects it and all its preceding transactions $t.cr$ by recording itself in *x.write-fc* (Line 29)or *x.read-fc* (Lines 47 and 14).

Transaction t may abort for two reasons. First, if a read operation cannot return a value without violating consistency (Line 44).Second, if there exists a transaction that t precedes (Lines 16, 22) but that also precedes t (Line 15).Finally, the clean function is dedicated to garbage collect by emptying the records (Lines 54–63). A transaction t is removed from the *write-fc* and *read-fc* sets only when all its preceding transactions have completed, i.e., their $t.cr = \emptyset$ (Lines 19–21).For the correctness proof, please refer to the full version of this paper [9].

Theorem 4. *SSTM is conflict-serializable.*

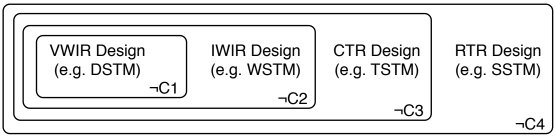


Fig. 3. Comparing the input acceptance of the TM designs. The VWIR design accepts no input patterns of the presented classes, the IWIR design accepts inputs that are neither in $\mathcal{C}1$ nor in $\mathcal{C}2$, and the CTR design accepts input patterns only outside $\mathcal{C}3$. Finally, we have not yet identified single-version patterns not accepted by design RTR.

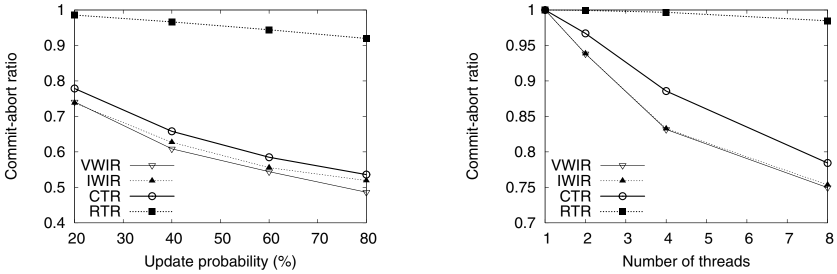


Fig. 4. Comparison of average commit-abort ratio of the various designs on a 256 element linked list: (left) with 8 threads as a function of the update probability; (right) with a 20% update probability as a function of the number of threads. As expected, the distinct commit-abort ratios follow the input acceptances but, interestingly, RTR presents a much better commit-abort ratio than other designs. Note that its heavy mechanism may produce an overhead compared to other designs.

5 Class Comparison and Experimental Validation

The previous section gives some impossibility results on the input acceptance by identifying input classes. Here, we use this classification to compare input acceptance of TM designs. Let $\mathcal{C}4$ be the class of all possible input patterns. Given the input acceptance upper bound, we are able to draw the input acceptance of VWIR, IWIR, CTR, and RTR designs restricted to patterns that are in $\neg\mathcal{C}1$, $\neg\mathcal{C}2$, $\neg\mathcal{C}3$, and $\neg\mathcal{C}4$, respectively. The hierarchy shown in Figure 3 compares the input acceptance of these TM designs.

To validate experimentally the tightness of our bounds on the input acceptance of our TM designs, we have implemented and tested all these designs: VWIR, IWIR, CTR, and RTR on an 8-core Intel Xeon machine using a sorted linked list benchmark. Results are depicted on Figure 4.

6 Conclusion

We upper-bounded the input acceptance of well-known TM designs and we proposed a new TM design with a higher acceptance. Our conclusion is that accepting various workloads requires complex TM mechanisms to test the input and to possibly reschedule it before outputting a consistent history. We expect this result to encourage further research on the best tradeoff between design simplicity and high input acceptance.

References

1. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC 2003, pp. 92–101 (2003)
2. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
3. Guerraoui, R., Kapałka, M.: On the correctness of transactional memory. In: PPoPP 2008, pp. 175–184 (February 2008)
4. Yannakakis, M.: Serializability by locking. J. ACM 31(2), 227–244 (1984)
5. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: PODC 2007, pp. 340–341 (2007)
6. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical Report TR-05-04, Department of Computer Sciences, University of Texas at Austin (2005)
7. Aydonat, U., Abdelrahman, T.S.: Serializability of transactions in software transactional memory. In: TRANSACT 2008. ACM, New York (2008)
8. Felber, P., Riegel, T., Fetzer, C.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008 (February 2008)
9. Gramoli, V., Harmanci, D., Felber, P.: Toward a theory of input acceptance for transactional memories. Technical Report LPD-REPORT-2008-009, EPFL (2008)
10. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not 38(11), 388–402 (2003)
11. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
12. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)

Geo-registers: An Abstraction for Spatial-Based Distributed Computing*

Matthieu Roy[1], François Bonnet[2], Leonardo Querzoni[3],
Silvia Bonomi[3], Marc-Olivier Killijian[1], and David Powell[1]

[1] LAAS-CNRS; Université de Toulouse; F-31077
[2] IRISA; Université Européenne de Bretagne; Rennes, F-35042
[3] Sapienza Università di Roma; Roma, I-00185

Abstract. In this work we present an abstraction that allows a set of distributed processes, aware of their respective positions in space, to collectively maintain information associated with an area in the physical world. This abstraction is a logical object shared between participating processes that provides two operations, namely read and write.

1 Introduction

Motivation. The advent of massively distributed pervasive systems in which every user carries a powerful computing device with communication and positioning capabilities, allows us to envision many new intrinsically decentralized applications and services that are tightly coupled to the position of entities. Two main issues arise when trying to deal with such dynamic and location-aware systems:

- new features of such systems are not represented in traditional distributed models, namely their dynamics and locality, and more specifically the geographical distribution of entities,
- the evolvable nature of such systems imposes that any mechanism built for them must be resilient to mobility- and failure-induced changes in the composition and/or topology of the system.

Our research efforts are focussed on providing suitable abstractions to reason about mobile, large-scale and geographically-aware systems. More specifically, in this paper, we introduce a software abstraction, called a *geo-register*, that can be used to associate some values to a geographic location. Unlike traditional failure mode assumptions, such as process crashes or byzantine behaviors, we solely consider movement as the only source of uncertainty in the system. The paper provides the specification for a *serial writes and concurrent reads* geo-register. A distributed algorithm implementing this specification is provided.

Related Work. Shared storage objects are very attractive abstractions that can be used for indirect process communication and that simplify the development of applications. In mobile environments, a few solutions have been proposed to

* The full version of this work is available as a research report [4]. This work is partially funded by FP6 NoE ReSIST, contract nr. 026764.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 534–537, 2008.

implement such shared objects. In [1,3], atomic memory implementations for mobile ad hoc networks are presented. Both approaches differ from the one presented in this paper because their aim is to build a register maintained by a set of geographic regions while our aim is to build a register in a given geographic region. While previous works focus on using geographic dispersion of nodes to tolerate failures, we are interested in the orthogonal problem of defining a shared storage in an area, in isolation of the remainder of the system.

2 Architecture and Model

Formal system model. The system is composed of entities $(p_i)_{i=1,2,\dots}$ of an infinite set Π that evolve in a 2-dimensional space, or geographic space. The entities are *correct*, i.e., execute correctly and do not crash, and *anonymous*, i.e., execute the same algorithm and do not own a unique identifier.

All entities are equipped with a positioning device and wireless network capabilities. The entities are aware of their position at all times with infinite precision. They can move in the space continuously with a bounded maximal speed V_{max}.

An area A is a geographic surface, i.e., a continuous subset of the space. At every instant t, let $\texttt{active}_A(t)$ be the set of processes in A. Since processes are correct and move continuously, $\texttt{active}_A(.)$ evolves only by additions or removals of entities. The area A is valid if $\forall t$, $\texttt{active}_A(t)$ is a clique w.r.t. communication capabilities, i.e., any two processes in the set can communicate.

Execution Model. To simplify reasoning, in the following we will refer to the starting and the ending of a given operation Op using two operators, $\mathsf{Begin}(Op)$ and $\mathsf{End}(Op)$. By definition, $\mathsf{Begin}(Op)$ corresponds to the time, as perceived by an external observer, at which the caller p_i invokes Op, and $\mathsf{End}(Op)$ is defined by the end of the operation Op from the system's point of view, i.e., the time at which the last action of the Op invocation protocol terminates. Two operations Op_1 and Op_2 in an execution are *non-concurrent* if $\big((\mathsf{End}(Op_1) < \mathsf{Begin}(Op_2)) \vee (\mathsf{End}(Op_2) < \mathsf{Begin}(Op_1))\big)$, else Op_1 and Op_2 are *concurrent.*

Geo-Reliable Broadcast. To abstract away physical parameters of the system, we suppose that the system is equipped with a *geo-reliable broadcast.* A geo-reliable broadcast is a communication primitive that guarantees that all processes located in an area A receive messages broadcasted to that area. From an implementation point of view, this primitive is built on top of wireless communication and positioning capabilities.

Definition 1 ($(\delta, A)-$geo-reliable broadcast). *Let δ be a positive number and A be an area. A $(\delta, A)-$geo-reliable broadcast enjoys the following properties:*

- *every process $p \in A$ can issue a* $\texttt{broadcast}(m)$
- *if m is a message broadcasted at time t by a correct process p that is in the area A from time t to time $t+\delta$, then all correct processes remaining in A between t and $t+\delta$ deliver m by time $t+\delta$.*

This definition is relatively weak, since it does not take into account the processes that may enter or leave the area during the broadcast, and only focuses on processes that stay in the area for the whole duration of a broadcast.

Definition 2 (Core region). *Let A be a valid area equipped with a $(\delta, A)-$geo-reliable broadcast. A* core region *A' associated with A is a subset of A such that every message m sent at time t by any process p in A' using $(\delta, A)-$geo-reliable broadcast will be delivered by every process q that was in A' at time t.*

Notice that this definition abstracts away some physical parameters of the system. In particular, the definition implies that a process that is in A' at time t is guaranteed to be in A at time $t + \delta$.

3 Non Concurrent Write Geo-registers

A *geo-register* is the abstraction of a storage mechanism attached to a particular area, that can be used to collectively store and retrieve pieces of information.

Intuitively, a geo-register implements a temporally-ordered sequence of (traditional) registers. Every element of the sequence corresponds to a temporal interval where entities populate the area. As soon as the area becomes empty, the state of the storage is lost, and when entities reenter the area, a new instance has to be created.

Inspired by the seminal paper of Lamport [2], we provide a specification of a *non-concurrent regular* register. More complex semantics, like multi-writer ones, are explored in [4].

The semantics of a non concurrent write geo-register[1] is defined with respect to 1) the most recently completed write operation and 2) the write operations possibly concurrent with a read operation, that can be defined as follows:

Definition 3. *Let Op be an operation performed on the register. The most recently completed write operation before Op is by definition W_{Op} such that*

$$\mathsf{End}(W_{Op}) = \max\{\mathsf{End}(W_x) : \mathsf{End}(W_x) < \mathsf{Begin}(Op)\}$$

Definition 4. *Let R be a read operation, and W_R the most recently completed write operation before R. Let $\mathcal{CW}$ be the set of write operations that are concurrent with R, and V the set of values written by operations in $\mathcal{CW} \cup \{W_R\}$. V is the set of possible outcomes of the read operation R.*

Definition 5 (geo-register). *Let A be an area, and A' an associated core region. A* geo-register *for (A, A') provides* read *and* write *operations such that:*

- *a* read *operation can be issued at time t by processes in $\mathtt{active}_A(t)$*
- *a* write *operation can be issued at time t by processes in $\mathtt{active}_{A'}(t)$*
- *Let a* read *operation R be issued by a process in $\mathtt{active}_A(\mathsf{Begin}(R))$. Let W_R be the most recently completed write operation before R and V be the set of possible outcomes of R. The value returned by R satisfies:*

[1] Notice that non concurrent write implies that, whenever multiple processes call write operations, no two write operations occur concurrently.

```
Geographically controlled thread:
when $p$ enters $A$:
    $R_p \leftarrow void$;
    wait for
        □ ($W(x)$ is received)  : $R_p \leftarrow x$; exit;
        □ ($2\delta$ time delay elapsed)
    RB_send($REQ$)
    wait for
        □ ($REP(v)$ is received)        : $R_p \leftarrow v$;
        □ ($W(x)$ is received)          : $R_p \leftarrow x$;
        □ ($2\delta$ time delay elapsed) : $R_p \leftarrow \perp$;
when $p$ leaves $A$:
    free($R_p$);
```

```
Communication controlled thread:
upon reception of ($REQ$) : if ($R_p \neq void$)
                            then RB_send($REP(R_p)$)
upon reception of ($W(x)$) : $R_p \leftarrow x$

Read and Write operations:
When $p$ is in $A$:
read() :    wait until ($R_p \neq void$)
            return($R_p$)

When $p$ is in $A'$:
write($x$) : RB_send($W(x)$)
```

Fig. 1. Implementation

(Partial Amnesia): *if, since* End(W_R)*, there exists an instant t such that* $\texttt{active}_{A'}(t) = \emptyset$*, it returns either a value in V or* $\perp$*.*
(Safety) *if* $\texttt{active}_{A'}$ *has never been empty from* End(W_R) *to* Begin(R)*, it returns a value in V.*

Implementation for 1*-hop Communication.* We provide an implementation for the simple one-hop broadcast-based system. In this solution, the parameter δ is a known period of time fixed by the geo-reliable-broadcast primitive from lower parameters; it abstracts the implementation details of the primitive that may include more than one broadcast due to message collisions. The proof is omitted and can be found in [4].

4 Conclusion

In this paper we provided the specification and a simple implementation of a geo-localized storage service for mobile systems. Unlike other similar work, we are interested in providing a local-only abstraction that can be used by applications that require information to be stored only when entities populate the area. Future research directions include the introduction of process failures, and the possibility of providing stronger semantics such as write concurrency.

References

1. Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., Welch, J.: GeoQuorums: Implementing Atomic Memory in Mobile Ad Hoc Networks. In: PODC 2003, pp. 306–320 (2003)
2. Lamport, L.: On Interprocess Communication. Distributed Computing (1985)
3. Tulone, D.: Ensuring strong data guarantees in highly mobile ad hoc networks via quorum systems. Ad Hoc Networks 5(8) (November 2007)
4. Roy, M., Bonnet, F., Querzoni, L., Bonomi, S., Killijian, M.-O., Powell, D.: Geo-registers: an abstraction for spatial-based distributed computing. LAAS report Nr. 08487 (October 2008)

Evaluating a Data Removal Strategy for Grid Environments Using Colored Petri Nets

Nikola Trčka, Wil van der Aalst, Carmen Bratosin, and Natalia Sidorova

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{n.trcka,w.m.p.v.d.aalst,c.c.bratosin,n.sidorova}@tue.nl

Abstract. We use Colored Petri Nets (CPNs) for the modeling and performance analysis of grid architectures. We define a strategy for the optimization of grid storage usage, based on the addition of data removal tasks to grid workflows. We evaluate the strategy by simulating our CPN model of the grid. Experiments show that the strategy significantly reduces the amount of storage space needed to execute a grid application.

1 Introduction

Grid computing has emerged as a powerful platform for executing data and computation intensive applications. Several grid architectures have been proposed to orchestrate available resources (e.g. [1,5,2]). However, as the complexity of grid applications continuously increases, there is always a need for new solutions. These solutions are typically first evaluated on a simulation model. In this paper we propose Colored Petri Nets (CPNs) [6] for the modeling and simulation of grid architectures.

Grid simulation has been an active area of research for quite some time, and several grid simulators have been developed [7,4]. There are, however, several advantages in using CPNs for this purpose: 1) CPNs are graphical, hierarchical, modular, and have a formal semantics[1]. They are executable and thus can model the dynamics of the grid as well. 2) CPNs are supported by CPN tools [6], a powerful framework for modeling, verification and performance analysis. 3) Petri nets have been extensively used for many years to model and analyze concurrent systems. Simple reuse of ideas makes grid modeling a relatively easy task. 4) Most grid workflow languages can be easily converted to Petri nets.

The grid architecture we model is reasonably generic, suitable for executing grid applications that are computationally intensive and manipulate (large) data. It can be seen as a computational grid in which data also plays an important role. We believe that our model covers all relevant aspects with enough detail, and that it can be used to discover trends that remain valid in more complex settings.

Every task in a grid workflow typically specifies the set of its input data files and the data that it generates. We frequently see cases where some data is

[1] The last feature is very relevant and directly used in this work.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 538–541, 2008.

only used at one region of a workflow. This data, although not needed after a certain point in time, stays on the grid until the complete workflow is finished. In an environment with reliable resources this is a waste of storage space and an optimization strategy is needed. In this paper we propose one such strategy, in form of a method that inserts data clean-up tasks to a workflow, at the points from which no further tasks access this data. We perform simulations on our CPN model of the grid, and show that the amount of storage space an application uses during execution is indeed significantly reduced when our strategy is applied.[2]

2 Modeling Grid Architectures

Our grid architecture consists of three layers. On the top is the *application layer*, a workflow environment in which the users describe their applications. On the bottom is the *fabric layer* consisting of grid nodes connected in a network. In between is the *middleware layer*, responsible for the allocation of resources and data handling.

Fig. 1 shows the CPN module of the scheduling system. The module illustrates the complexity of, and the level of detail covered by, our CPN model. It also shows the descriptive power of CPNs, as only basic knowledge of Petri nets is needed to understand how the scheduler works.

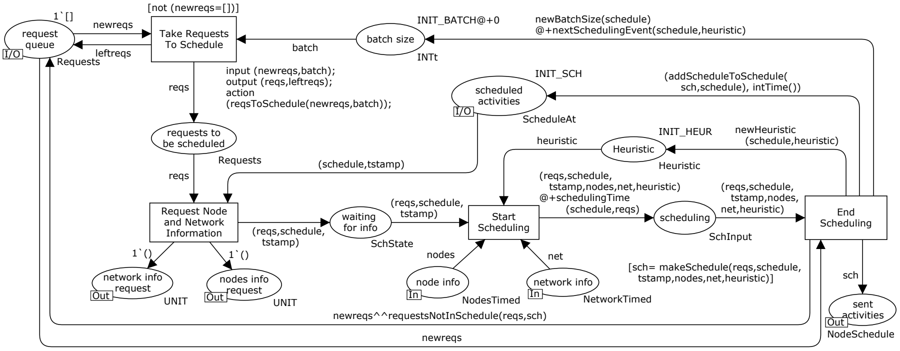


Fig. 1. CPN model of the scheduler

3 Data Removal Strategy

Our optimization strategy is based on the insertion of data clean-up tasks at the points from which this data is no longer needed. Since data elements can be (re)used in loops, in parallel or in alternative branches, the main challenge is to identify these points. However, Petri nets provide ways to achieve this easily. We present the method on the basis of an example.

[2] This short paper presents the main ideas only; full details are given in [8].

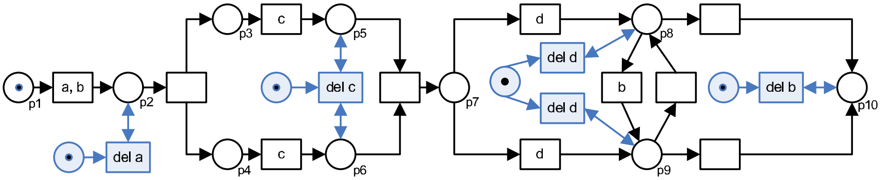


Fig. 2. Data clean-up tasks added to a workflow

Table 1. Percentage of storage space freed

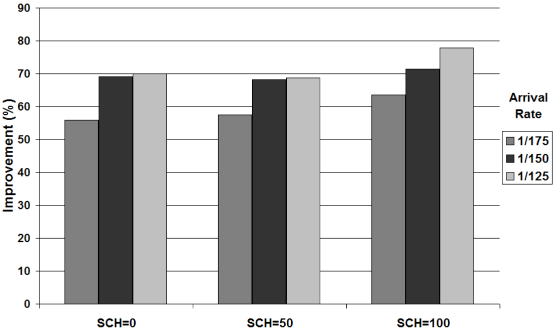


Fig. 2 shows a (sound[3]) grid workflow (ignore the gray elements for the moment), where a, b, c, and d are data elements shared among the tasks. Our method introduces a clean-up task for each element, with one guard place to ensure that the removal request is issued only once. These additions are illustrated in gray color in Fig. 2. The element c is thus deleted when there is a token in place p_5 *and* a token in place p_6. The element d is deleted when there is a token in p_8 *or* a token in p_9. While a can be deleted immediately after the first task, b can only be deleted at the end, as it can be used in the loop between p_8 and p_9.

We evaluate the strategy by means of simulation, using our CPN model. The testbed consists of nine grid nodes, fully connected in a homogeneous network. The input workflow is fixed, but investigated for the case-arrival rates ranging from 1/175 to 1/125. We use periodic Min-Min scheduling with the period varied from 0 to 100.

The results in Table 1 show the percentage of storage space that becomes available when the strategy is applied. The improvement is greater for longer scheduling periods and for higher arrival rates. This is expected, as longer periods enable more effective prioritization of removal tasks, and more cases result in longer unnecessary space occupation.

[3] Soundness property [3] is a sanity check for workflows. It ensures that the workflow can always complete, and that it has no dead transitions.

4 Conclusion

We modeled a grid architecture in terms of Colored Petri Nets. The model is formal, graphical and executable, offering an unambiguous view of how different parts of the grid are structured and how they interact. The model is suitable for performance analysis, it is fully adaptable and extendible.

To solve the problem of data occupying the grid storage space unnecessarily long, we introduced a method for the addition of data clean-up tasks to grid workflows. We evaluated this method by conducting a simulation experiment using our CPN model of the grid. The results showed that the required storage space could be reduced by as far as 80%.

References

1. Condor Project, http://www.cs.wisc.edu/condor/
2. K-WfGrid web site, http://www.kwfgrid.eu
3. van der Aalst, W.M.P.: Verification of Workflow Nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
4. Buyya, R., Murshed, M.M.: GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. Concurrency and Computation: Practice and Experienc 14(13-15), 1175–1220 (2002)
5. Deelman, E., Singh, G., Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Sci. Program. 13(3), 219–237 (2005)
6. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer 9(3-4), 213–254 (2007)
7. Legrand, A., Marchal, L., Casanova, H.: Scheduling Distributed Applications: the SimGrid Simulation Framework. In: 3rd International Symposium on Cluster Computing and the Grid (CCGRID 2003), Washington, pp. 138–146. IEEE, Los Alamitos (2003)
8. Trčka, N., van der Aalst, W.M.P., Bratosin, C., Sidorova, N.: Evaluating a Data Removal Strategy for Grid Environments Using Colored Petri Nets. Technical report, Eindhoven University of Technology (2008), http://w3.win.tue.nl/nl/onderzoek/onderzoek_informatica/computer_science_reports/

Load-Balanced and Sybil-Resilient File Search in P2P Networks

Hyeong S. Kim[1], Eunjin (EJ) Jung[2], and Heon Y. Yeom[1]

[1] School of Computer Science and Engineering
Seoul National University, Seoul, Korea
{hskim,yeom}@dcslab.snu.ac.kr
[2] Department of Computer Science
University of Iowa, Iowa City IA 52242, USA
ejjung@cs.uiowa.edu

Abstract. We present a file search algorithm that is not only Sybil-resilient, but also load-balanced among users. We evaluate our algorithm in terms of bandwidth consumption and the likelihood of bad downloads. In both metrics, our algorithms show balanced overhead among users and small chances of bad download with low bandwidth consumption.

Keywords: File sharing, peer-to-peer networks, Sybil attack, load balancing.

1 Introduction

Peer-to-peer (P2P) networks amount to a large portion of the Internet traffic, and file sharing, the most popular P2P application, accounts for the most traffic in this portion. iPoque reports that 49 to 83% of the Internet traffic in five world-wide regions in August and September 2007 is P2P, and most of them are file sharing applications like eDonkey and BitTorrent [1]. Thus it is important for a P2P file sharing application to be efficient in bandwidth consumption. Bandwidth consumption in file downloads is inevitable, but those in file search and bad downloads, e.g. wrong, corrupted or malicious file downloads, are overheads. In this paper, we target to reduce these overheads while resisting Sybil attacks and ensuring balanced loads among users.

There are two common problems in P2P file search. First, it is difficult to decide which uploader (the user who has the requested file and is willing to share with other users) is more trustworthy than others. P2P users may share their experiences and use reputation systems, e.g. EigenTrust [2]. However, reputation systems often fail under the second problem: Sybil attack [3]. In most distributed systems, including P2P networks, a malicious user may create a large number of identities (Sybil nodes) to unfairly influence the reputation system. There have been efforts to prevent Sybil attacks using social networks, e.g. SybilGuard [4], but these efforts often put unfairly large loads on high-degree nodes. In P2P networks, it is unlikely that any user will make such sacrifice to serve other users.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 542–545, 2008.

In this paper, we present new file search algorithm LBSR which stands for Load-Balanced and Sybil-Resilient. We evaluate our algorithms in terms of bandwidth consumption and inauthentic downloads. In both metrics, the LBSR algorithm shows better balanced load among peers and low chances of inauthentic download with much less network consumption.

2 LBSR File Search Algorithm

Algorithm 1. LBSR file search algorithm for peer i

```
$N_i$: a set of peer $i$'s neighbors, $T_{fwd}$: forwarding threshold
$d_{ij}$: degree of neighbor $j$ in peer $i$'s view, where $j \in N_i$
if $TTL(q) = 0$ or requested file found then
    return { }if the requested file is found, notify the issuer of $q$
else
    sum:=0, TTL(q):=TTL(q)-1
    while $sum < T_{fwd}$ do
        select a random neighbor $j$ from $|N_i|$ based on the probability $p_j = \frac{1}{{d_{ij}}^{\alpha}}$
        forward $q$ to neighbor $j$
        $sum = sum + d_{ij}$
    end while
end if
```

Our intuition for the LBSR algorithm is straightforward from our goals: load balancing from distributing the search queries among all the nodes and Sybil-resilience comes from forwarding search queries over the social network. These two straightforward intuitions are hard to combine into one algorithm. Social networks tend to have skewed distribution of degrees, and efficient file search algorithms based on forwarding tend to stress high degree nodes. In result, efficient search and load balancing are at odds.

We use a combination of probabilistic selection and a weighted sum. Peer i's chances of forwarding requests to neighbor j is proportional to the inverse of the degree of j with discount factor α. When $\alpha = 1$, LBSR highly favors low-degree nodes and slows down the search. When $\alpha = 0$, then the all the neighbors of peer i will have the same probability, which becomes the *random* algorithm presented in Section 3. In Section 3, $\alpha = 0.5$. Peer i also repeats forwarding until the sum of the selected neighbors' degrees reaches T_{fwd}. This way i may replicate the query for more times when i happens to forward to low-degree neighbors. In Section 3, $T_{fwd} = 50$.

3 Experimental Results

We implemented the LBSR algorithm in the Query-Cycle Simulator [5] used in EigenTrust [2]. Our experiment setting followed those in [2]: the network consists of 62 good peers and 40 malicious peers. Each simulation consists of 150 cycles. The specific settings for our experiments include the new attack model where malicious peers respond to every query they receive with a fake file. We ran each setting for five times and averaged the results.

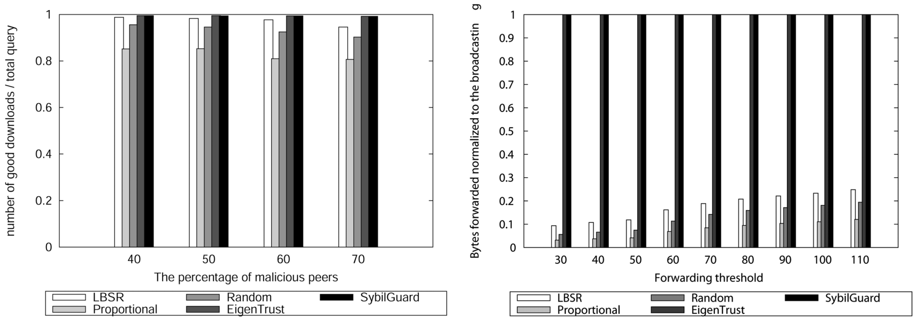
We have experimented with four other algorithms that operate on social networks for comparison. In “proportional” algorithm, the forwarding probability is proportional to the degree of neighbors. This algorithm finds the target file fast, but has more load imbalance and less Sybil-resilience. In “random” algorithm, the forwarding probability is based on uniform distribution. The random algorithm achieves better load balancing but also suffers from higher chances of inauthentic downloads. EigenTrust [2] is a reputation system wherein each peer is assigned a unique global trust value that reflects the experiences of all peers in the network uploaded to or downloaded from this peer. In SybilGuard [4], each node stores random routes in social networks. The intersecting peer of those random routes vouch for the social relationships from the intersecting peer to both peers. In our simulation, peers only download from uploaders with more intersecting nodes than the threshold.

3.1 Sybil Resilience

Fig. 1(a) depicts the fraction of good downloads with varying malicious peer ratio. EigenTrust and SybilGuard achieve almost 100% of good downloads, while LBSR 95% in average, 90% even in the presence of 70% malicious peers. This result is more interesting compared to the bandwidth consumption in Fig. 1(b) where the LBSR algorithm consumes only 10 to 20% of EigenTrust or SybilGuard.

3.2 Load Balancing in Bandwidth Consumption

We show the load balancing comparison in Fig. 2. With each algorithm, we collected the network bandwidth for each peer and computed the share of its load over all the participants. Peers with spikes in other algorithms are high-degree nodes, while LBSR shows not as high spikes.



(a) Ratio of good downloads for each requested file

(b) Forwarded bytes compared to broadcast

Fig. 1. Sybil resilience in terms of download quality

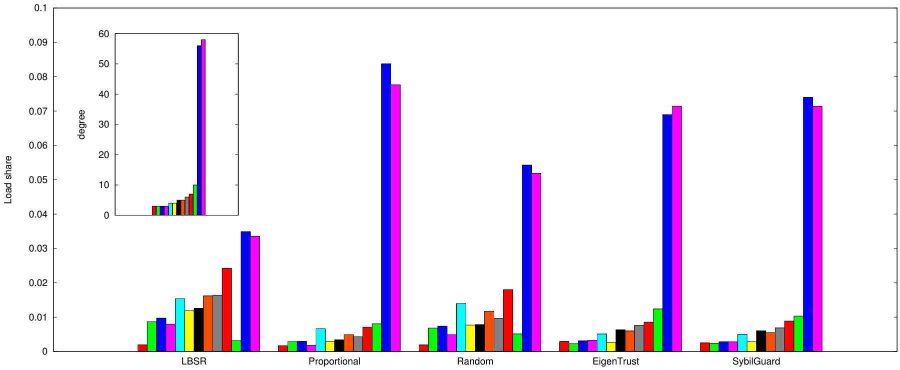


Fig. 2. Load balancing in bandwidth (We sampled every 5th node in the increasing order of degree. The degree of each node is displayed in the inner figure.)

4 Future Work and Conclusion

P2P networks consume a major part of the Internet bandwidth, and most consumption comes from file sharing applications. Our LBSR algorithm shows low inauthentic download rates with low and balanced bandwidth consumption among peers. As future work, we plan to incorporate reputation system into forwarding probability computation for better inauthentic download rate.

References

1. Schulze, H., Mochalski, K.: iPoque internet study (2007), http://www.ipoque.com/userfiles/file/internet_study_2007.pdf
2. Kamvar, S.D., Scholosser, M.T., Garcia-Molina, H.: The eigenTrust algorithm for reputation management in p2p networks. In: Proceedings of the 12th international conference on World Wide Web. ACM Press, New York (2003)
3. Douceur, J.: The sybil attack. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429. Springer, Heidelberg (2002)
4. Yu, H., Kaminsky, M., Gibbons, P.B., Flaxman, A.: SybilGuard: Defending against sybil attacks via social networks. In: ACM SIGCOMM (2006)
5. Schlosser, M.T., Condie, T.E., Kamvar, S.D.: Simulating a file-sharing p2p network. In: Proceedings of Workshop in Semantics in Peer-to-Peer and Grid computing (2003)

Computing and Updating the Process Number in Trees*

David Coudert, Florian Huc, and Dorian Mazauric

Mascotte, INRIA, I3S, CNRS, University of Nice Sophia Antipolis, France
{firstname.lastname}@sophia.inria.fr

Abstract. The process number is the minimum number of requests that have to be simultaneously disturbed during a routing reconfiguration phase of a connection oriented network. From a graph theory point of view, it is similar to the node search number, and thus to the pathwidth, however they are not always equal. In general determining these parameters is NP-complete.

We present a distributed algorithm to compute these parameters and the edge search number, in trees. It can be executed in an asynchronous environment, requires n steps, an overall computation time of $O(n \log n)$, and n messages of size $\log_3 n + 2$. Then, we propose a distributed algorithm to update these parameters on each component of a forest after addition or deletion of any tree edge. This second algorithm requires $O(D)$ steps, an overall computation time of $O(D \log n)$, and $O(D)$ messages of size $\log_3 n + 3$, where D is the diameter of the new connected component.

Keywords: Pathwidth, process number, distributed algorithm.

1 Introduction

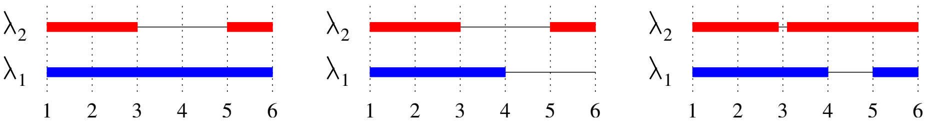
Treewidth and pathwidth have been introduced by Robertson and Seymour [1] as part of the graph minor project. Those parameters are very important since many problems can be solved in polynomial time for graphs with bounded treewidth or pathwidth. By definition, the treewidth of a tree is one, but its pathwidth might be up to $\log n$. A linear time centralized algorithms to compute the pathwidth of a tree has been proposed in [2,3,4], but so far no distributed algorithm exists.

The algorithmic counter part of the notion of pathwidth (denoted pw) is the cops and robber game [5,6,7]. It consists in finding an invisible and fast fugitive in a graph using the smallest set of agents. The minimum number of agents needed gives the node search number (denoted ns). Other graph invariants closely related to the notion of pathwidth have been proposed such as the process number [8,9] (denoted pn) and the edge search number [10] (denoted es). Their determination is in general NP-complete [5].

In this paper, we describe in Sec. 2 the motivation of the problem from a network reconfiguration problem point of view. In Sec. 3, we propose a fully distributed

* This work was partially funded by the European projects IST FET AEOLUS and COST 293 GRAAL, ARC CARMA, ANR JC OSERA, CRC CORSO and Région PACA.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 546–550, 2008.



(a) Initial routing of (1,6), (1,3) and (5,6)

(b) Removal of (1,6) and addition of (1,4)

(c) Solution with (1,3), (1,4), (3,6) and (5,6)

Fig. 1. Starting from the routing of Fig. 1(a), the removal of request (1,6) and addition of request (1,4) gives the routing of Fig. 1(b). Request (3,6) can not be added in Fig. 1(b), although the routing of Fig. 1(c) is possible.

algorithm to compute the process number of trees, which can be executed in an asynchronous environment. Furthermore, with a small increase in the amount of transmitted information, we extend our algorithm to a fully dynamic algorithm allowing to add and remove edges even if the total size of the tree is unknown.

2 Motivation and Modeling

The process number of a (di)graph has been introduced to model a routing reconfiguration problem in connection oriented networks such as WDM, MPLS or wireless backbone networks [8,9]. In such networks, and starting from an optimal routing of a set requests, the routing of a new connection request can be done greedily using available resources (e.g. capacity, wavelengths) thus avoiding to reroute existing connections. Some resources might also be released after the termination of some requests. In fine, such traffic variations may lead to a poor usage of resources with eventual rejection of new connections. For example, in Fig. 1 where the network is a 6 nodes path with two wavelengths, a new connection request from 3 to 6 would be rejected in Fig. 1(b) although the routing of Fig. 1(c) is possible. To optimize the number of granted requests, the routing has to be reconfigured regularly.

In this context, routing reconfiguration problem consists in going from a routing, R_1, to another, R_2, by switching requests one by one from the original to the destination route. This yield to a scheduling problem. Indeed, resources assigned to request r in R_2 might be used by some request r' in R_1 might, thus request r' has to be rerouted before r. We represent these constraints by a digraph $D = (V, A)$ in which each node corresponds to a request, and there is an arc from vertex u to vertex v if v must be rerouted before u. When the digraph D is acyclic, the scheduling is straightforward, but in general, it contains cycles. To break them, some requests have to be temporarily interrupted, thus removing incident arcs in D, and so, the optimization problem is to find a scheduling minimizing the number of requests simultaneously interrupted. When the digraph is symmetric, the problem can be solved on the underlying undirected graph G, and we will restrict our study to this case in the following.

As for the pathwidth, our problem can be expressed as a cops and robber game. An interruption is represented by placing an agent on the corresponding node in G, a node is said *processed* when the corresponding request has been rerouted, and we call a *process strategy* a series of the three following actions allowing to reroute all requests with respect to the constraints represented by the graph.

(1) put an agent on a node (*interrupt a connection*).
(2) remove an agent from a node if all its neighbors are either processed or occupied by an agent (*release a connection to its final route when destination resources are available*). The node is now processed (*connection has been rerouted*).
(3) process a node if all its neighbors are occupied by an agent.

A *p-process strategy* is a strategy which process the graph using p agents and the *process number*, pn(G), is the smallest p such that a p-process strategy exists. For example, a star has process number 1, a path of more than 4 nodes has process number 2, a cycle of size 5 or more has process number 3, and a $n \times n$ grid, $n \geq 3$, has process number $n+1$. Moreover, it has been proved in [8,9] that $\mathrm{pw}(G) \leq \mathrm{pn(G)} \leq \mathrm{pw(G)} + 1$, where $\mathrm{pw}(G)$ is the *pathwidth* of G [1], and that determining the process number is in general NP-complete.

The node search number [5], $\mathrm{ns}(G)$, can be defined similarly except that we only use rules (1) and (2). It was proved by Ellis *et al.* [2] that $\mathrm{ns}(G) = \mathrm{pw}(G)+1$, and by Kinnersley [11] that $\mathrm{pw}(G) = \mathrm{vs}(G)$, where $\mathrm{vs}(G)$ is the *vertex separation* of G. Those results show that vertex separation, node search number and pathwidth are equivalent, but so far it is not known when equivalence also holds with the process number.

3 Distributed Algorithms

We propose an algorithm, `algoHD`, to compute the process number of a tree with an overall of $O(n \log n)$ operations. The principle of `algoHD` is to perform a hierarchical decomposition of the tree. Each node u of degree $d(u)$ collects a compact view of the subtree rooted at each of its sons ($d(u) - 1$ neighbors), computes a compact view of the subtree it forms and sends it to its father (last neighbor), thus constructing a hierarchical decomposition. The algorithm is initialized at the leaves, and the node receiving messages from all its neighbors (the root) concludes on the process number of the tree. Notice that our algorithm is fully distributed and that it can be executed in an asynchronous environment assuming that each node knows its neighbors.

The message sent by a node v to its father v_0 describes the structure of the *subtree* T_v *rooted at* v, that is the connected component of T minus the edge vv_0 containing v. More precisely, the message describes a decomposition of T_v into a set of smaller disjoint trees, each of them being indexed by its root. See [12] for more details.

Lemma 1. *Given a n-nodes tree T, `algoHD` computes* pn(T) *in n steps and overall $O(n \log n)$ operations, sending n messages each of size* $\log_3 n + 2$.

We propose a dynamic algorithm that allows to compute the process number of the tree resulting of the addition of an edge between two trees. It also allows to delete any edge. To do it efficiently, it uses one of the main advantage of the hierarchical decomposition: the possibility to change the root of the tree without additional information.

Lemma 2. *Given two trees $T_i = (V_i, E_i)$ rooted at $r_i \in V_i$, $i = 1, 2$, its hierarchical decompositions, and $r'_i \in V_i$, we can compute the hierarchical decomposition of $T = (V_1 \cup V_2, E_1 \cup E_2 \cap (r'_1, r'_2))$, and so compute its process number in $O(D)$ steps of time complexity $O(\log n)$ each, using $O(D)$ messages of size $\log n + 3$ (D is the diameter of T).*

The best and worst cases of the incremental algorithm (`IncHD`) are:

- Worst case: T consists of two subtrees of size $n/3$ and process number $\log_3 n/3$ linked via a path of length $n/3$. Edges are inserted alternatively in each opposite subtrees. Thus `IncHD` requires an overall of $O(n^2 \log n)$ operations.
- Best case: edges are inserted in the order induced by `algoHD` (inverse order of a breadth first search). `IncHD` needs an overall of $O(n \log n)$ operations.

4 Conclusion

In this paper we have proposed a distributed algorithm to compute the process number of a tree, as well as the node and edge search numbers, changing only the values of the initial cases of our algorithm. Then, we have proposed a dynamic algorithm to update these invariants after addition or deletion of any tree edge. Finally we have adapted the algorithm to compute the process number of a tree if its size is unknown and we have characterized the trees for which the process number (resp. edge search number) equals the pathwidth [12]. A challenging task is to characterize other classes of graphs where equality holds or to prove it is NP-hard to decide it in the general case.

References

1. Robertson, N., Seymour, P.D.: Graph minors. I. Excluding a forest. J. Combin. Theory Ser. B 35, 39–61 (1983)
2. Ellis, J., Sudborough, I., Turner, J.: The vertex separation and search number of a graph. Information and Computation 113, 50–79 (1994)
3. Scheffler, P.: A linear algorithm for the pathwidth of trees. In: Bodendiek, R., Henn, R. (eds.) Topics in Combinatorics and Graph Theory, pp. 613–620. Physica-Verlag, Heidelberg (1990)
4. Skodinis, K.: Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. Journal of Algorithms 47, 40–59 (2003)
5. Kirousis, M., Papadimitriou, C.: Searching and pebbling. Theoretical Computer Science 47, 205–218 (1986)

6. Fomin, F., Thilikos, D.: An annotated bibliography on guaranteed graph searching. Theor. Comput. Sci. 399, 236–245 (2008)
7. Díaz, J., Petit, J., Serna, M.: A survey on graph layout problems. ACM Computing Surveys 34, 313–356 (2002)
8. Coudert, D., Perennes, S., Pham, Q.C., Sereni, J.S.: Rerouting requests in wdm networks. In: AlgoTel 2005, Presqu'île de Giens, France, pp. 17–20 (2005)
9. Coudert, D., Sereni, J.S.: Characterization of graphs and digraphs with small process number. Research Report 6285, INRIA (2007)
10. Megiddo, N., Hakimi, S.L., Garey, M.R., Johnson, D.S., Papadimitriou, C.H.: The complexity of searching a graph. J. Assoc. Comput. Mach. 35, 18–44 (1988)
11. Kinnersley, N.G.: The vertex separation number of a graph equals its pathwidth. Information Processing Letters 42, 345–350 (1992)
12. Coudert, D., Huc, F., Mazauric, D.: A distributed algorithm for computing and updating the process number of a forest. Research Report 6560, INRIA (2008)

Redundant Data Placement Strategies for Cluster Storage Environments*

André Brinkmann and Sascha Effert

University of Paderborn, Germany
{brinkman,fermat}@upb.de

Abstract. The continued exponential increase in stored data as well as the high demand for I/O performance is imposing high pressure on the scalability properties of storage environments. The resulting number of disk drives in huge environments does not only lead to management, but also to reliability problems. The foundations of scalable and reliable storage systems are data distribution algorithms, which are able to scale performance and capacity based on the number of disk drives and which are able to efficiently support multi-error correcting codes. In this paper, we propose data distribution strategies, which are competitive concerning the number of data movements required to optimally adapt to a changing number of heterogeneous disk drives under these constraints.

1 Introduction

The ability to scale storage environments from a small number of hard disks up to hundreds or even thousands of disks requires the ability to adapt the underlying data distribution and error-correcting codes to new infrastructures. Not adapting the data layout would lead to a segmentation of the data. A promising approach to scale storage environments without sacrificing performance is the use of pseudo-randomized hash functions, which are able to adapt the data layout with nearly minimum overhead [4] [2]. In case of pseudo-random hash functions, the address of each block of a virtual disk has to be mapped to a physical disk. The mapping to a sector number on that physical disk has to be performed by a second instance, which can either be integrated into the physical disk (e.g. in case of object based storage devices) or can be part of a volume management solution. However, storing just a single copy is not sufficient to ensure data reliability. A simple alternative is to use Consistent Hashing to store multiple copies of each data block or to use it to assign the data to a disk cluster [3]. The error correcting code can then be mapped on top of this pseudo-random hash functions. Nevertheless, these simple approaches are not able to efficiently use the available storage capacity.

* This work has been partially supported by the EU within the 6th Framework Programme under contract IST-511531 *Highly Predictable Cluster for Internet Grids (HPC4U)* and under contract IST-2005-034891 *Hydra*.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 551–554, 2008.

1.1 The Model

The applied model is based on an extension of the standard balls into bins model. Let $\{0, \ldots, (m-1)\}$ be the set of all identifiers for the balls and $\{0, \ldots, (n-1)\}$ be the set of identifiers for the bins. We will often assume for simplicity that the balls and bins are numbered in a consecutive way starting with 0. Furthermore, we will use the terms bins and disks interchangeable inside this paper.

Suppose that bin i with a unique identifier b_i can store up to cap_i (copies of) balls. In some cases, we will use the name b_i of a bin and i interchangeable. We define the relative capacity of the bin as $c_i = cap_i / \sum_{j=0}^{n-1} cap_j$. We require that for every ball k copies have to be stored in the system for some fixed k and that all copies of a ball have to be stored in different bins. In this case, a trivial upper bound for the number of balls the system can store while preserving fairness and redundancy is $\sum_{j=1}^{n} cap_j / k$, but it can be much less in certain cases [1].

A placement strategy will be called c-competitive concerning the insertion or removal of a bin, if it induces the (re-)placement of (an expected number of) at most c times the number of copies an optimal strategy would need for this operation. The redistribution of balls is necessary to ensure an even distribution of requests among the bins. To bound the competitiveness of the proposed algorithm, we will introduce the notion of an (α, β)-restricted environment:

Definition 1. *Inside an (α, β)-restricted environment it holds for all bins i with $0 \leq i \leq (n-1)$ that $\frac{1}{\alpha \cdot n} \leq c_i \leq \frac{\beta}{n}$ for arbitrary $\alpha, \beta \in \mathbb{R}^+ \geq 1$.*

The definition of an (α, β)-restricted environment will help us to characterize the homogeneity of a set of bins and to provide bounds depending on this homogeneity.

1.2 Related Work

Data reliability is mostly achieved by using RAID encoding schemes, which divide data blocks into specially encoded sub-blocks that are placed on different disks [6]. RAID encoding schemes are normally implemented by striping data blocks according to a pre-calculated pattern across all the available storage devices, achieving a nearly optimal performance in small environments.

In the following, we just focus on randomized data placement strategies which are able to cope with dynamic changes of the capacities or the set of storage devices (or bins) in the system. Good strategies are known for uniform capacities without replication. In this case, it only remains to cope with situations in which bins enter or leave the system [4].

Adaptive data placement schemes that are able to cope with arbitrary heterogeneous capacities have been introduced in [2]. First methods with dedicated support for replication are described in [3][7]. The proposed *Rush*-strategy maps replicated objects to a scalable collection of storage servers according to user-specified server weighting. Brinkmann et al. have shown that the known approaches for data replication are not sufficient to ensure capacity and performance efficiency in heterogeneous storage environments [1]. The proposed

Redundant Share strategy is the first strategy that is able to support replication inside dynamic environments for a heterogeneous set of disks without sacrificing storage capacity. The drawback of Redundant Share is a competitiveness for the insertion or removal of a disk drive that is not independent of the replication degree k and the number of bins n. A first strategy with a competitiveness independent of the number of copies k and the number of bins n is Spread [5].

1.3 Contributions of This Paper

Inside this paper, we propose new data distribution strategies for cluster storage environments. The strategies are based on a combination of Redundant Share with the Share strategy proposed in [2] that is able to overcome the restrictions of Redundant Share. The new strategy has a competitiveness of $O(1)$ compared to an optimal algorithm, if the capacity difference of the disks can be bounded by an arbitrary constant. If this difference can not be bounded, the strategy still has a competitiveness of at most $O(\ln n)$ for all operations and of $O(1)$ for many important operations. The main idea of this new strategy is to reduce the problem of distributing copies over a set of heterogeneous disks with different capacities to the problem of distributing copies over a set of homogeneous disks, where each disk has the same capacity. Compared to previous strategies, both implementation and analysis of the new strategies are much simpler [5].

2 Algorithmic Approach and Analysis Sketch

The algorithmic idea of the proposed redundant data placement strategy for heterogeneous bins is to combine one strategy that is able to reduce the problem of distributing balls over a set of heterogeneous bins to the problem of distributing them over a set of homogeneous bins and one strategy that is able to efficiently replicate data over this set of homogeneous bins.

For reducing the problem of heterogeneous bins to homogeneous bins, we have selected the *Share*-strategy [2]. For distributing copies of a ball over a homogeneous set of bins, we have chosen the *Redundant Share*-strategy [1]. Redundant Share in its original form has some drawbacks, e.g. it is not able to adjust the capacities of some of the bins without an unacceptable overhead concerning the replacement of data blocks. Redundant Share over Share works very similar to the original version of Share. The only difference is that Share calls Redundant Share as sub-function to distribute the k balls instead of calling a uniform data distribution strategy for a single ball.

Inside this section, we will shortly cover the most important aspects of Redundant Share as well as for Redundant Share over Share. We will start by showing some matching bounds for Redundant Share for homogeneous bins:

Theorem 1. *The lower bound for the competitiveness as well as the expected competitiveness of Redundant Share for inserting new, homogeneous bins for a k-replication scheme is $1/2 \cdot (k+1)$ and it is $\ln n$-competitive concerning the insertion or removal of arbitrary bins.*

In any case, Redundant Share keeps the ordering of the copies. This ordering is important, if each copy of a ball has a different meaning. We will show in the following theorem that Redundant Share is even 1-competitive concerning the insertion of a new bin in this setting for homogeneous bins.

Theorem 2. *Redundant Share is 1-competitive concerning the insertion of a new homogeneous bin, if the ordering of copies can be changed.*

For the combined strategy of Redundant Share over Share to work correctly, we require that every point $x \in [0, 1)$ is covered by at least k intervals I_i w.h.p. from different bins. This is the case for a stretch factor $s \geq 5 \cdot k \cdot \ln n$. In the next theorem we will show that the combination of Share and Redundant Share has a constant competitive ratio for arbitrary n.

Theorem 3. *Share in combination with Redundant Share is O(1)-competitive concerning the insertion and deletion of bins for (α, β)-restricted environments.*

We will show in the following theorem that Redundant Share over Share is highly competitive concerning the insertion of a new biggest bin for non-restricted environments.

Theorem 4. *Redundant Share over Share is in the expected case $\frac{1+\ln n}{2}$ -competitive for the insertion of a biggest bin.*

References

1. Brinkmann, A., Effert, S., Meyer auf der Heide, F., Scheideler, C.: Dynamic and Redundant Data Placement. In: Proceedings of the 27th IEEE International Conference on Distributed Computing Systems, ICDCS (2007)
2. Brinkmann, A., Salzwedel, K., Scheideler, C.: Compact, adaptive placement schemes for non-uniform distribution requirements. In: Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA) (2002)
3. Honicky, R.J., Miller, E.L.: Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In: Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2004)
4. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of the 29th ACM Symposium on Theory of Computing (STOC) (1997)
5. Mense, M., Scheideler, C.: Spread: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In: Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA) (2008)
6. Patterson, D.A., Gibson, G., Katz, R.H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). In: Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD) (1988)
7. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: CRUSH: Controlled, Scalable And Decentralized Placement Of Replicated Data. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC) (2006)

An Unreliable Failure Detector for Unknown and Mobile Networks

Pierre Sens[1], Luciana Arantes[1], Mathieu Bouillaguet[1], Véronique Simon[1], and Fabíola Greve[2]

[1] LIP6, Université Pierre et Marie Curie, INRIA, CNRS, France
{pierre.sens,luciana.arantes,mathieu.bouillaguet,veronique.simon}@lip6.fr
[2] DCC - Computer Science Department / Federal University of Bahia, Brazil
fabiola@dcc.ufba.br

Abstract. This paper presents an asynchronous implementation of a failure detector for unknown and mobile networks. Our approach does not rely on timers. Neither the composition nor the number of nodes in the system are known. Our algorithm can implement failure detectors of class $\Diamond S$ when behavioral properties and connectivity conditions are satisfied by the underlying system.

1 Introduction

Unreliable failure detector, namely FD, is a fundamental service, able to help in the development of fault-tolerant distributed systems. FD can informally be seen as a per process oracle, which periodically provides a list of processes suspected of having crashed. In this paper, we are interested in the class of FD denoted $\Diamond S$ [1]. They ensure that (i) eventually each crashed process is suspected by every correct process (*strong completeness*), and (ii) there is a time after which some correct processes are never suspected (*eventual weak accuracy*).

We propose a new asynchronous FD algorithm for dynamic systems of mobile and unknown networks. It does not rely on timers to detect failures and no knowledge about the system composition nor its cardinality are required. The basic principle of our FD is the flooding of failure suspicion information over the network. Initially, each node only knows itself. Then, it periodically exchanges a QUERY-RESPONSE [2] pair of messages with its neighbors. Based only on the reception of these messages and on the partial knowledge about its neighborhood, a node is able to suspect other processes or revoke a suspicion in the system. A proof that our implementation provides a FD of class $\Diamond S$ is available at the research report [3].

2 Model and Behavioral properties

Model. We consider a dynamic distributed system composed of a finite set Π of $n > 1$ mobile nodes, $\Pi = \{p_1, \ldots, p_n\}$. Each process knows its own identity and it knows only a subset of processes in Π. It does not know n. There is one

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 555–559, 2008.

process per node which communicates with its 1-hop neighbors by sending and receiving messages via a packet radio network. There are no assumptions on the relative speed of processes or on message transfer delays, thus the system is asynchronous. A process can fail by crashing. Communications between 1-hop neighbors are considered to be reliable. Nodes are mobile and they can keep continuously moving and pausing. A faulty node will eventually crash. Nonetheless, we assume that there are no network partitions in the system in spite of node failures and mobility. We also assume that each node has at least d neighbors and that d is known to every process. Let f_i denote the maximum number of processes that may crash in the neighborhood of any process. We assume that the local parameter f_i is known to every process p_i and $f_i + 1 < d$.

Behavioral properties. Let us now define some behavioral properties that the system should satisfy in order to ensure that our algorithm implements a FD of class $\Diamond S$. In order to implement any type of FD with an unknown membership, processes should interact with some others to be known. According to [4], if there is some process in the system such that the rest of processes have no knowledge whatsoever of its identity, there is no algorithm that implements a FD with weak completeness. Thus, the following *membership property*, namely $\mathcal{MP}$, should be ensured by all nodes in the system. This property states that, to be part of the membership of the system, a process p_m (either correct or not) should interact at least once with other processes in its neighborhood by broadcasting a QUERY message when it joins the network. Moreover, this query should be received and kept in the state of at least one correct process in the system, beyond the process p_m itself.

Let p_m be a mobile node. Notice that a node can keep continuously moving and pausing, or eventually it crashes. Nonetheless, we consider that, infinitively often, p_m should stay within its target range destination for a sufficient period of time in order to be able to update its state with recent information regarding failure suspicions and mistakes. Hence, in order to capture this notion of “sufficient time of connection within its target range”, the following *mobility property*, namely $\mathcal{M}obi\mathcal{P}$, has been defined. This property should be satisfied by all mobile nodes. Thus, $\mathcal{M}obi\mathcal{P}$ for p_m at time t ensures that, after reaching a target destination, there will be a time t at which process p_m should have received QUERY messages from at least one correct process, beyond itself. Since QUERY messages carry the state of suspicions and mistakes in the membership, this property ensures that process p_m will update its state with recent informations.

Let us define another important property in order to implement a $\Diamond S$ FD. It is the *responsiveness property*, namely $\mathcal{RP}$, which denotes the ability of a node to reply to a QUERY among the first nodes. This property should hold for at least one correct node. The $\mathcal{RP}(p_i)$ property states that after a finite time u, the set of responses received by any neighbor of p_i to its last QUERY always includes a response from p_i. Moreover, as node can move, the $\mathcal{RP}(p_i)$ also states that neighbors of p_i eventually stop moving outside p_i's transmission range. $\mathcal{RP}$ property should hold for at least one correct stationary node. It imposes that eventually there is some “stabilizing” region where the neighborhood of some correct “fast” node p_i does not change.

Properties $\mathcal{MP}$ and $\mathcal{RP}$ may seem strong, but in practice they should just hold during the time the application needs the strong completeness and eventual weak accuracy properties of FD of class $\diamond S$, as for instance, the time to execute a consensus algorithm.

3 Implementation of a Failure Detector of Class $\diamond S$

The following algorithm describes our protocol for implementing a FD of class $\diamond S$ when the underlying system satisfies $\mathcal{MP}$ and $\mathcal{M}obi\mathcal{P}$ for all participating nodes and the $\mathcal{RP}$ for at least one correct node. We use the following notations:

- $susp_i$: denotes the current set of processes suspected of being faulty by p_i. Each element of this set is a tuple of the form $\langle id, ct\rangle$, where id is the identifier of the suspected node and ct is the tag associated to this information.
- $mist_i$: denotes the set of nodes which were previously suspected of being faulty but such suspicions are currently considered to be a mistake. Similar to the $susp_i$ set, the $mist_i$ is composed of tuples of the form $\langle id, ct\rangle$.
- rec_from_i: denotes the set of nodes from which p_i has received responses to its last QUERY message.
- $known_i$: denotes the current knowledge of p_i about its neighborhood. $known_i$ is then the set of processes from which p_i has received a QUERY message.
- $Add(set, \langle id, ct\rangle)$: is a function that includes $\langle id, ct\rangle$ in set. If an $\langle id, -\rangle$ already exists in set, it is replaced by $\langle id, ct\rangle$.

The algorithm is composed of two tasks. Task $T1$ is made up of an infinite loop. At each round, a QUERY message is sent to all nodes of p_i's range neighborhood (line 5). Node p_i waits for at least $d - f_i$ responses, which includes p_i's own response (line 6). Then, p_i detects new suspicions (lines 7-12). It starts suspecting each node p_j, not previously suspect, which it knows ($p_j \in known_i$), but from which it does not receive a RESPONSE to its last QUERY. If a previous mistake information related to this new suspected node exists in the mistake set $mist_i$, it is removed from it (line 10) and the suspicion information is then included in $susp_i$ with a tag which is greater than the previous mistake tag (line 9). If p_j is not in the $mist$ set (i.e., it is the first time p_j is suspected), p_i suspected information is tagged with 0 (line 12).

Task $T2$ allows a node to handle the reception of a QUERY message. A QUERY message contains the information about suspected nodes and mistakes kept by the sending node. However, based on the tag associated to each piece of information, the receiving node only takes into account the ones that are more recent than those it already knows. The two loops of task $T2$ respectively handle the information received about suspected nodes (lines 18–24) and about mistaken nodes (lines 25–30). Thus, for each node p_x included in the suspected (respectively, mistake) set of the QUERY message, p_i includes the node p_x in its $susp_i$ (respectively, $mist_i$) set only if the following condition is satisfied: p_i received a more recent information about p_x status (failed or mistaken) than the ones it has in its $susp_i$ and $mist_i$ sets. Furthermore, in the first loop of task $T2$, a new mistake is detected if the receiving node p_i is included in the suspected set of the QUERY message (line 20) with a greater tag. At the end of the task (line 31), p_i sends to the querying node a RESPONSE message.

When a node p_m moves to another destination, p_m will start suspecting the nodes of its old destination since they are in its $known_m$ set.

```
1  init:
2   $susp_i \leftarrow \varnothing; mist_i \leftarrow \varnothing$ ; $known_i \leftarrow \varnothing$
3  Task T1:
4  Repeat forever
5      broadcast QUERY($susp_i$, $mist_i$)
6      wait until RESPONSE received from at least $(d - f_i)$ processes
7      For all $p_j \in known_i \setminus rec\_from_i \mid \langle p_j, -\rangle \notin susp_i$ do
8          If  $\langle p_j, ct\rangle \in mist_i$
9              $Add(susp_i, \langle p_j, ct+1\rangle)$
10             $mist_i = mist_i \setminus \{\langle p_j, -\rangle\}$
11         Else
12             $Add(susp_i, \langle p_j, 0\rangle)$
13 End repeat
14
15 Task T2:
16 Upon reception of QUERY ($susp_j$,$mist_j$) from $p_j$ do
17 $known_i \leftarrow known_i \cup \{p_j\}$
18 For  all $\langle p_x, ct_x\rangle \in susp_j$ do
19 If $\langle p_x, -\rangle \notin susp_i \cup mist_i$ or ($\langle p_x, ct\rangle \in susp_i \cup mist_i$ and $ct < ct_x$)
20     If $p_x = p_i$
21         $Add(mist_i, \langle p_i, ct_x + 1\rangle)$
22     Else
23         $Add(susp_i, \langle p_x, ct_x\rangle)$
24         $mist_i = mist_i \setminus \{\langle p_x, -\rangle\}$
25 For  all $\langle p_x, ct_x\rangle \in mist_j$ do
26 If $\langle p_x, -\rangle \notin susp_i \cup mist_i$ or ($\langle p_x, ct\rangle \in susp_i \cup mist_i$ and $ct < ct_x$)
27     $Add(mist_i, \langle p_x, ct_x\rangle)$
28     $susp_i = susp_i \setminus \{\langle p_x, -\rangle\}$
29     If ($p_x \neq p_j$)
30         $known_i = known_i \setminus \{p_x\}$
31 send RESPONSE to $p_j$
```

Lines 29–30 allow the updating of the *known* sets of both the node p_m and of those nodes that belong to the original destination of p_m. For each mistake $\langle p_x, ct_x\rangle$ received from a node p_j such that node p_i keeps an old information about p_x, p_i verifies whether p_x is the sending node p_j. If they are different, p_x should belong to a remote destination. Thus, process p_x is removed from the local set $known_i$.

References

1. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM 43(2), 225–267 (1996)
2. Mostefaoui, A., Mourgaya, E., Raynal, M.: Asynchronous implementation of failure detectors. In: DSN (June 2003)
3. Sens, P., Arantes, L., Bouillaguet, M., Greve, F.: Asynchronous implementation of failure detectors with partial connectivity and unknown participants. Research Report 6088, INRIA (January 2007)
4. Fernández, A., Jiménez, E., Arévalo, S.: Minimal system conditions to implement unreliable failure detectors. In: PRDC, pp. 63–72. IEEE Computer Society, Los Alamitos (2006)

Efficient Large Almost Wait-Free Single-Writer Multireader Atomic Registers

Andrew Lutomirski[1] and Victor Luchangco[2]

[1] Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, MA 02139 and AMA Capital Management, LLC, Los Angeles, CA 90067
[2] Sun Microsystems Laboratories, Burlington, MA 01803

Abstract. We present a nonblocking algorithm for implementing single-writer multireader atomic registers of arbitrary size given registers only large enough to hold a single word. The algorithm has several properties that make it practical: It is simple and has low memory overhead, readers do not write, write operations are wait-free, and read operations are almost wait-free. Specifically, to implement a register with w words, the algorithm uses $N(w + O(1))$ words, where N is a parameter of the algorithm. Write operations take amortized $O(w)$ and worst-case $O(Nw)$ steps, and a read operation completes in $O(w(\log(k + 2) + Nk \cdot 2^{-N}))$ steps, where k is the number of write operations it overlaps.

1 Introduction

Consider a system in which one process updates data that is read asynchronously by many processes. For example, the data may represent a value from some sensor. If the data is larger than can be handled by primitive operations of the system, some synchronization is necessary to ensure the consistency of data read by any process, so that, for example, reads do not interleave data from multiple writes. In such a system, readers should get the latest data available, no process should cause other processes to wait, and overhead should be minimized.

The abstract specification of this system is an *atomic single-writer multireader register* [3], which provides a *write operation* that only one process may invoke and a *read operation* that any process may invoke, such that each operation can be thought of as happening atomically at some point between its invocation and response. We present a simple new nonblocking algorithm for implementing such a register of arbitrary size using only registers large enough to hold a single word. In this algorithm, readers never write shared data and thus do not interfere with the writer or with other readers. Write operations always complete in a bounded number of steps, as do read operations, unless the number of write operations that they overlap is exponential in a parameter that determines the space overhead of the algorithm. Specifically, for a positive integer N, a w-word register uses $N(w+O(1))$ words. A write operation takes $O(Nw)$ worst-case and $O(w)$ amortized steps. A read operation takes at most $O(w(\log(k+2)+Nk\cdot2^{-N}))$ steps, where k is the number of write operations it overlaps. Thus, so long as it overlaps no more than $O(2^N)$ writes, a read operation will complete in $O(Nw)$ or fewer steps.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 560–563, 2008.

2 Algorithm Description and Analysis

The key idea in our algorithm is a recursive construction of a *regular register* (i.e., one in which reads may return the value written by the last write that completed before the read began or by any overlapping write [3]). Code for this construction appears in Fig. 1. Although the embedded register is the same size as the implemented register, it is accessed only by every second write operation and by read operations that overlap writes, so fewer operations on the embedded register overlap and conflict. By using this construction repeatedly to implement the embedded register, we can exponentially reduce the number of operations that access the innermost register. The depth at which we bound the recursion is the parameter N of our algorithm.

```
Fields
    data: an array of $w$ words
    tag: a natural number, initially 0
    next: regular register of size $w$, initially data$[1..w]$

Write(d): // d is an array of $w$ words
    if tag $= 2 \bmod 4$ then
        next.Write(d)
    tag $\leftarrow$ tag $+ 1$
    for i $= 1..w$ do
        data[i] $\leftarrow$ d[i]
    tag $\leftarrow$ tag $+ 1$

TryRead():
    origTag $\leftarrow$ tag
    if origTag is even then
        for i $= 1..w$ do
            d[i] $\leftarrow$ data[i]
        if tag $=$ origTag then
            return d
    return failed

Read():
    value $\leftarrow$ TryRead()
    if value $\neq$ failed then
        return value
    return next.Read()

ReadAtomic():
    value $\leftarrow$ TryRead()
    if value $\neq$ failed then
        return value
    n $\leftarrow$ next.Read()
    value $\leftarrow$ TryRead()
    if value $\neq$ failed then
        return value
    return n
```

Fig. 1. An implementation of a regular (with Read) or atomic (with ReadAtomic) single-reader multireader register of size w using a preexisting regular register

In addition to the embedded register, the algorithm maintains an array of words containing the data and updated by every Write, as well as a "tag." A Read that does not overlap a Write can simply read and return the data in the array. However, if the writer is updating the array then the Read must not return data from the array, which may include words from both before and after the write. To achieve this, the writer uses the tag to indicate when the array is being written, incrementing tag before starting to write the array and again after the array is written. Thus, a reader that sees the same even tag both before and after it reads the array is assured that the array was not written in that interval, so it can safely return the data read. On the other hand, a Read that sees an odd tag or sees tag change, must overlap a Write. In this case, it reads the embedded regular register and returns the value it gets.

To see that the resulting register is regular, note that tag is even whenever no Write is in progress and that whenever tag is even, the value in data was written by the most recently completed Write. Thus, a Read that does not read next returns the value written by the last Write completed before the Read began, as required. A Read that does read next must overlap with some Write, and either that Write or the one immediately preceding it wrote its value into next before the Read began to read next. Therefore, because (by assumption) next is regular, such a Read returns either the value written by the last Write that completed before the Read began or by a Write that overlaps the Read, also as required.

The register is not atomic, however, because a Read, seeing an odd tag due to an overlapping Write, may return the value being written by a subsequent Write. If the latter Write has only written next and not yet incremented tag, then a subsequent Read may return the value of the preceding Write, violating atomicity.

We can avoid this problem and thus guarantee atomicity by rereading tag and data (if tag is even) after reading next, as in the ReadAtomic function in Fig. 1. This prevents any Read from returning the value of a Write that has not incremented tag before the Read returns. It is easy to verify that the value returned by ReadAtomic was the abstract value of the register at some point during its operation, where the abstract value is the value in data when tag is even and the value last written into next when tag is odd. Because next is written by even-numbered Writes, the abstract value of the register changes when even-numbered Writes increment tag the first time and when odd-numbered Writes increment tag the second time.

Finally, we can bound the construction to N levels by simply retrying any Read that fails on the Nth level. That is, if $\texttt{next}_0$ is the outermost register (for which we use ReadAtomic rather than Read), and $\texttt{next}_{i+1} = \texttt{next}_i\texttt{.next}$, then we restart $\texttt{next}_0$.ReadAtomic whenever $\texttt{next}_N$.Read would be called and ignore calls to $\texttt{next}_N$.Write.

Every 2^ith Write recurses to a depth of i or more, with a maximum depth of N for any Write. Thus, Write has $O(w)$ amortized and $O(Nw)$ worst-case step complexity. A Read that calls $\texttt{next}_i$.Read must overlap two calls to $\texttt{next}_{i-2}$.Write: the call that causes $\texttt{next}_{i-2}$.TryRead to fail and the subsequent call that causes $\texttt{next}_{i-1}$.TryRead to fail; thus, it overlaps at least $2^{i-2} + 1$ calls to $\texttt{next}_0$.Write. Thus, a Read that overlaps k Writes retries $O(k \cdot 2^{-N})$ times and succeeds on its final try after making $O(w \log(k + 2))$ word-sized accesses, for a step complexity of $O(w(\log(k + 2) + Nk \cdot 2^{-N}))$.

3 Discussion

The efficiency of the algorithm described above can easily be improved. For example, the nested data structures can be implemented as arrays. Since Read is tail-recursive, it can easily be made iterative, using only $O(w)$ (rather than

$O(Nw)$) local space. Similarly, Write is “head-recursive,” so it can compute how far to recurse and then iterate over all the writes.

Although we have only presented code for operations that read or write all w words of the register with step complexity linear in w, we can easily read or write a subset of those words more efficiently. For Read, we can simply read the relevant words of `data` (recursively if necessary). For Write, a small amortized amount of bookkeeping is needed to propagate partial changes into each `next` register as it is written. If a typical operation only access a few words of a large register, this can greatly reduce the cost of these operations.

To ensure that Reads detect when `tag` is changed, `tag` is unbounded. Otherwise, the value of `tag` might wrap around and have the same value both time TryRead reads it, an instance of the ABA problem. However, a `tag` that wraps around to 0 whenever it reaches $2K$ is sufficient to avoid the ABA problem as long as no Read overlaps K or more Writes. Thus, a 64-bit `tag` would be sufficient provided that Reads overlap fewer than $2^{63} > 10^{18}$ Writes.

Although we have assumed sequential consistency and atomic access to single words, our algorithm remains correct even under much weaker guarantees. In particular, accesses to `data` can be completely unsynchronized and reordered, and reads of `data` that overlap with writes can even return arbitrary values, as long as the accesses to `tag` serve as memory barriers. Thus, each operation requires relatively little synchronization, improving performance on large registers, especially on systems that allow out-of-order execution.

There is a rich literature on register constructions (see [1] for some examples), but most of these are not practical. In real systems, nonblocking implementations of large registers typically involve expensive synchronization primitives (e.g., [4]). In contrast, our algorithm supports an unlimited number of readers, requires few memory barriers, and can take advantage of hardware-enforced read-only memory when available to protect against buggy or malicious readers. Kopetz and Reisinger [2] propose an algorithm similar to ours in that it uses only reads and writes, is parameterized by a factor N for the space overhead, and provides wait-free writes. However, in their algorithm, a reader can starve if it is only N times slower than the writer, whereas in our algorithm, it must be 2^N times slower.

References

1. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, Amsterdam (2008)
2. Kopetz, H., Reisinger, J.: The non-blocking write protocol NBW: A solution to a real-time synchronization problem. In: Real-Time Systems Symposium, 1993, Proceedings, pp. 131–137 (1993)
3. Lamport, L.: On interprocess communication, part I: Basic formalism. Distributed Computing 1(2), 77–85 (1986)
4. Sorenson, P.G., Hamacher, V.C.: A real-time system design methodology. In: INFOR, vol. 13, pp. 1–18 (1975)

A Distributed Algorithm for Resource Clustering in Large Scale Platforms

Olivier Beaumont, Nicolas Bonichon, Philippe Duchon,
Lionel Eyraud-Dubois, and Hubert Larchevêque

Universit de Bordeaux, INRIA Bordeaux Sud-Ouest, Laboratoire Bordelais de Recherche en Informatique

Abstract. We consider the resource clustering problem in large scale distributed platforms, such as BOINC, WCG or Folding@home. In this context, applications mostly consist in a huge set of independent tasks, with the additional constraint that each task should be executed on a single computing resource. We aim at removing this last constraint, by allowing a task to be executed on a (small) set of resources. Indeed, for problems involving large data sets, very few resources may be able to store the data associated to a task, and therefore may be able to participate to the computations. Our goal is to propose a distributed algorithm for a large set of resources that enables to build clusters, where each cluster will be responsible for processing a task and storing associated data. From an algorithmic point of view, this corresponds to a bin covering problem with an additional distance constraint. Each resource is associated to a weight (its capacity) and a position in a metric space (its location, based on network coordinates such as those obtained with Vivaldi), and the aim is to build a maximal number of clusters, such that the aggregated power of each cluster (the sum of the weights of its resources) is large enough and such that the distance between two resources belonging to the same cluster is kept small (in order to minimize intra-cluster communication latencies). In this paper, we describe a generic 2-phases algorithm, based on resource augmentation and whose approximation ratio is 1/3. We also propose a distributed version of this algorithm when the metric space is $\mathbb{Q}^D$ (for a small value of D) and the L_∞ norm is used to define distances. This algorithm takes $O((4^D)\log^2 n)$ rounds and $O((4^D)n\log n)$ messages both in expectation and with high probability, where n is the total number of hosts.

Introduction

The past few years have seen the emergence of a new type of high performance computing platforms. These highly distributed platforms, such as BOINC [1], Folding@home [2] and WCG [3] are characterized by their high aggregate computing power, their heterogeneity in terms of resource performances and by the dynamism of their topology, due to node arrivals and departures. Until now, all the applications running on these platforms (Seti@home [4], Folding@home [2],...) consist in a huge number of independent tasks, and all data necessary to process

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 564–567, 2008.

a task must be stored locally in the processing node. The only data exchanges take place between the master node and the slaves, what strongly limits the set of applications that can be performed on these platforms.

Two kind of applications fit in this model. The first one consists in those, such as Seti@home, where a huge set of data can be arbitrarily split into arbitrarily small amounts of data that can be processed independently on participating nodes. The second one corresponds to Monte-Carlo simulations. In this case, all slaves work on the same data, except a few parameters that drive the simulation. This is for instance the model corresponding to Folding@home.

In this paper, our aim is to extend this last set of applications. More precisely, we consider the case where the data set needed to perform a task is possibly too large to be stored at a single node. This situation is very likely to occur in large scale platforms based on the aggregation of strongly heterogeneous resources. In this case, both processing and storage must be distributed on a small set of nodes that will collaborate to perform the task. The nodes involved in the cluster should have an aggregate capacity (memory, processing power,...) higher than a given threshold, and they should be close enough (the latencies between those nodes should be small) in order to avoid high communication latencies.

In this context, the aim is the following: given a set of weighted items (the weights are the storage capacity of each node), and a metric (based on latencies), to create a maximum number of groups so that the maximal latency between two hosts inside any group is lower than a given threshold, and so that the total storage capacity of any group is greater than a given storage threshold. This problem turns out to be difficult, even if one node knows the whole topology (*i.e.* the available memory at each node and the latency between each pair of nodes). Indeed, even without the distance constraint, this problem is equivalent to the classical NP-complete bin covering problem [5]. Similarly, if the constraint about storage capacity is removed, but the distance constraint is kept, the problem is equivalent to the NP-Complete disk cover problem [6].

Results

Due to the lack of space, we refer the interested reader to the companion research report [7] where all the proofs and algorithms are provided in details.

In this paper, we propose a generic greedy 2-phases algorithm, based on resource augmentation and whose approximation ratio is $\frac{1}{3}$. More precisely, we use resource augmentation in the following way. We compare the number of clusters (or bins) created by our algorithm with diameter constraint d to the optimal number of bins that could be created with distance $d_{\max}$, where $d > d_{\max}$. This resource augmentation is both efficient and realistic. Indeed, if the aggregated memory of the cluster should be larger than a given threshold in order to be able to process the task, the threshold on the maximal latency between two nodes belonging to the same cluster is weaker, and mostly states that nodes belonging to the same cluster should not be too far from each other. Moreover, this resource augmentation enables to prove a constant approximation ratio ($\frac{1}{3}$) whereas approximation ratio without resource augmentation would be exponential in the dimension of the metric space .

The basic structure of this 2-phases greedy algorithm is the following:

Phase 1 Greedily create bins of diameter at most $d_{\max}$
Phase 2 Greedily create bins of diameter at most $3d_{\max}$.

Theorem 1. *The 2-phases greedy algorithm provides a* $\frac{1}{3}$*-approximation algorithm of* max_DCBC *problem, using a resource augmentation of factor* $2 + \frac{d}{d_{\max}}$ *on the maximal diameter of a bin.*

An extension of the generic 2-phases greedy algorithm with approximation ratio $\frac{2}{5}$ with the same resource augmentation is also possible. These results are to be compared to some classical results for bin covering in centralized environment without the distance constraint. In this (much easier) context, a $PTAAS$ (polynomial-time asymptotic approximation scheme) has been proposed for bin covering [8], *i.e.* algorithms A_ϵ such that for any $\epsilon > 0$, A_ϵ can perform, in a polynomial time, a $(1-\epsilon)$-approximation of the optimal when the number of bins tends towards the infinite. Many other algorithms have been proposed for bin covering, such as [5], that provides algorithms with approximation ratio of $\frac{2}{3}$ or $\frac{3}{4}$, still in a centralized environment.

This paper is a follow-up to [9], where the case of a one-dimensional metric space is considered. In order to estimate the positions of the nodes involved in the large scale platform, we rely on mechanisms such as Vivaldi [10,11] that associate to each node a set of coordinates in a low dimension metric space, so that the distance between two points approximates the latency between corresponding hosts. Here, we consider the case where resource locations are given by their coordinates in a metric space with arbitrary dimension. Moreover, in a large scale dynamic environment such as BOINC, where nodes connect and disconnect with a high churn, it is unrealistic to assume that a node knows all platform characteristics. Therefore, in order to build the clusters, we need to rely on fully distributed schemes, where a node makes the decision to join a cluster based on its position, its weight, and the weights and positions of its neighbor nodes. Therefore, we also propose a distributed version of this algorithm when the metric space is $\mathbb{Q}^D$ (for a small value of D) and the infinity norm is used to define distances.

Theorem 2. *There exists an algorithm, running in parallel for* 4^D *disjoint intervals, that uses* $O(4^D n \log n)$ *messages and, in a synchronous execution model where each message takes unit time,* $O(4^D \log^2 n)$ *rounds, both in expectation and with high probability, where* n *is the total number of hosts.*

Moreover, we claim that this algorithm can be used in practice, since its implementation only relies on classical distributed data structures, such as skip graphs [12].

In future works, we plan to adapt the algorithm to the case where several characteristics must be satisfied simultaneously (for instance, a task may require both a large aggregated memory and a large disk storage capacity). Another interesting work is to compare the performances of the distributed algorithm we propose with the gossip-based approach. Gossip-based algorithm complexities are usually very difficult to establish, but these algorithms have been proved very efficient to exploit locality [13],[14]. At last, we need to adapt the algorithm to the case where the metric space is not $\mathbb{Q}^D$. Indeed, if network coordinates systems based on landmarks [15] used $\mathbb{Q}^D$ (for values of D of order 10) as underlying metric space, more recent coordinate systems, such as Vivladi [10] rely on much more sophisticated metric spaces (but based on 3 coordinates only).

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: GRID 2004: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, Washington, DC, USA, pp. 4–10. IEEE Computer Society, Los Alamitos (2004)
2. (Folding@home), http://folding.stanford.edu/
3. (World community grid), http://www.worldcommunitygrid.org
4. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Commun. ACM 45, 56–61 (2002)
5. Assmann, S., Johnson, D., Kleitman, D., Leung, J.: On a dual version of the one-dimensional bin packing problem. Journal of algorithms (Print) 5, 502–525 (1984)
6. Franceschetti, M., Cook, M., Bruck, J.: A geometric theorem for approximate disk covering algorithms (2001)
7. Beaumont, O., Bonichon, N., Duchon, P., Eyraud-Dubois, L., Larcheveque, H.: A dsitributed algorithm for resource clustering in large scale platforms. Research report, INRIA Bordeaux Sud-Ouest, France, 15 pages (2008)
8. Csirik, J., Johnson, D., Kenyon, C.: Better approximation algorithms for bin covering. In: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, pp. 557–566 (2001)
9. Beaumont, O., Bonichon, N., Duchon, P., Larcheveque, H.: Distributed approximation algorithm for resource clustering. In: Shvartsman, A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058. Springer, Heidelberg (2008)
10. Cox, R., Dabek, F., Kaashoek, F., Li, J., Morris, R.: Practical, distributed network coordinates. ACM SIGCOMM Computer Communication Review 34, 113–118 (2004)
11. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. In: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 15–26 (2004)
12. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 384–393 (2003)
13. Ganesh, A., Kermarrec, A., Massoulié, L.: Peer-to-Peer Membership Management for Gossip-Based Protocols (2003)
14. Voulgaris, S., Gavidia, D., van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. Journal of Network and Systems Management 13, 197–217 (2005)
15. Ng, T., Zhang, H.: Predicting internet network distance with coordinates-based approaches. In: IEEE (ed.) Proceedings of INFOCOM 2002, pp. 170–179 (2002)

Reactive Smart Buffering Scheme for Seamless Handover in PMIPv6

Hyon-Young Choi, Kwang-Ryoul Kim, Hyo-Beom Lee, and Sung-Gi Min*

Dept. of Computer Science and Engineering, Korea University, Seoul, Korea
sgmin@korea.ac.kr

Abstract. PMIPv6 is proposed as a new network-based local mobility management. Even if PMIPv6 exploits the locality of Mobile Nodes (MNs) at the mobility management, it still has the packet loss problem during the handover period like MIPv6. We propose a new reactive network-based scheme for seamless handover in PMIPv6. The scheme prevents packet loss during the handover period by buffering packets which are expected to be lost by MN's movement. All decisions related with early packet buffering are made at the access routers without any MN's involvement.

1 Introduction

Mobility management protocols are widely researched with the advance of the wire communication technology. IETF NetLMM working group proposed the Proxy MIPv6 (PMIPv6) [1] as a network-based local mobility management protocol. The beauty of PMIPv6 is that mobile nodes do not involve in any mobility functionality. The mobility of any standard IPv6 device can be achieved without any user device modification. Many network service providers keep an eye on the PMIPv6.

However, PMIPv6 has the same problem with MIPv6 [2]: the loss of packet during the handover period. FMIPv6 [3] provides fast and seamless handover for MIPv6. The power of FMIPv6 comes from the information given by mobile nodes (MNs). Because MIPv6 performs the host-based handover, FMIPv6 can notify the serving access router (AR) with MN's and target AR's information about the impending handover. The main difficulty of applying FMIPv6 to PMIPv6 is how an AR knows the beginning of MN's handover and target AR information.

Recently, several fast handover methods based on FMIPv6 for PMIPv6 have been proposed [4][5]. In [4], "Layer 2 (L2) HO signaling" is used to detect the MN's handover decision. The "L2 HO signaling" contains the information of the MN identifier and the new AP identifier. In case of IEEE 802.16e, the MOB_HO_IND message may act as "L2 HO signaling." However, if L2 layer does not provide such a message like IEEE 802.11, this procedure will not work. In [5], the dependency of L2 technology is avoided by using Context Transfer Protocol. In this protocol, an MN sends a REPORT message which includes MN

* Corresponding author.

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 568–571, 2008.

identifier and the new AP identifier to a serving Mobile Access Gateway (MAG). The role of REPORT message is the same as the FBU message in FMIPv6. The problem of this approach is that an MN must support Context Transfer Protocol. However, using L2 handover signaling or Context Transfer Protocol can be seen as another form of MN's involvement in the handover procedure.

2 The Smart Buffering Scheme for PMIPv6

The Smart Buffering scheme predicts an MN's movement using the network-side information and starts buffering packets to be expected to lose. The detection of the attachment of the MN after the MN's movement is done by the new MAG. After detecting the attachment, the new MAG notifies the attachment to the previous MAG. The previous MAG forwards the buffered packets to the new MAG.

The movement prediction is based on the receiving signal strength indication (RSSI) of an MN at the serving MAG. If the RSSI crosses the given threshold, the MAG decides that the MN movement is imminent and it starts packet buffering as well as forwarding packets to the MN.

To avoid excessive buffering by a premature handover decision, buffered packets are time-stamped. If the lifetime of a buffered packet is expired, the packet is discarded. The lifetime of buffered packets is the maximum expected handover time of the current PMIPv6 domain. To fetch the buffered packets in a previous MAG, the new MAG after the MN's attachment must find the previous MAG. The new MAG multicasts a discovery message (Flush Request) to its neighbor MAGs. When the previous MAG receives the discovery message , it replies the acknowledge message (Flush Acknowledgement) to the new MAG and starts forwarding buffered packets. IP-in-IP tunnel is used between two MAGs.

Fig. 1 shows the sequence diagram of the Smart Buffering when an MN hands over from MAG1 to MAG2 while communicating with the CN. Except buffering and Flush Request/Acknowledgement messages, the handover sequence of Smart Buffering follows the standard PMIPv6 procedure.

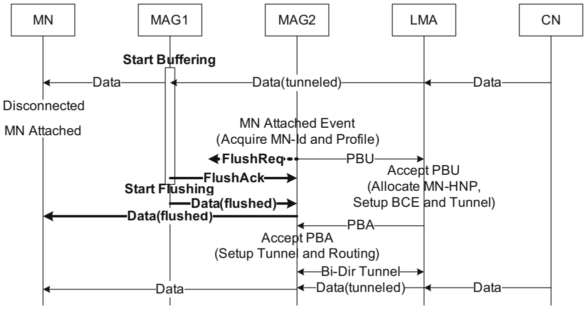


Fig. 1. Sequence diagram of the proposed scheme

Table 1. The comparison of FMIPv6, FMIPv6-based PMIPv6s, and Smart Buffering

	FMIPv6	[4]	[5]	Smart Buffering
Operation	Proactive+Reactive			Reactive
Movement Detection	FBU (Active)	L2 signal (Active [L2])	CXTP (Active)	RSSI (Passive)
Buffering	Yes	Yes	Yes	Yes
Handover Manage	MN-based	MN-based	MN-based	Network-based

Table 1 compares FMIPv6, [4], [5], and the Smart Buffering. As mentioned above, FMIPv6 based proposals depend on the MN-based handover initiation with different kinds of handover indication, whilst the Smart Buffering does not depend on any MN's assistance. This complete independency from MN's aid during the handover period comes from that the Smart Buffering is based on the network-side proactivity, early buffering, rather than the host-side proactivity, handover indication.

3 Experiments

We performed simulation using ns-2 network simulator with NIST-modified-ns-2.29 in IEEE 802.11 environment. The simulation network topology is shown by Figure 2.

The LMA manages two MAGs which support the smart buffering, and the MN moves from the MAG1 to the MAG2 while communicating with the CN. The link delay of all wired links is 10 ms and the link capacity is 100 Mbps for the wired link and 11 Mbps for the wireless link. The CN communicates with the MN through CBR over UDP with rates, 300 Kbps, 500 Kbps, 1 Mbps, and 2 Mbps. Each scenario ran 100 times.

Table 2 shows the average packet loss at the MN. The results proves that the Smart Buffering prevents packet loss during the handover period. Fig. 3 shows the total handover latency for PMIPv6 only and PMIPv6 with Smart Buffering. We calculate the time between the last packet from old MAG and the first packet from new MAG as a handover latency. PMIPv6 with Smart Buffering scheme slightly improves the total handover latency.

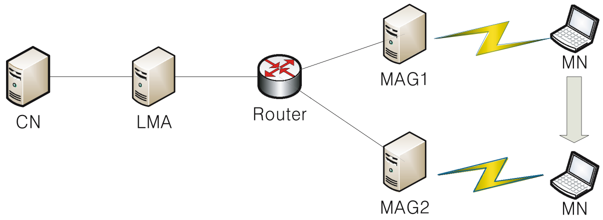


Fig. 2. Simulation topology

Table 2. Lost packets count

Rate	PMIPv6 only	PMIPv6 with Smart Buffering
300 Kbps	16.33	0
500 Kbps	28.53	0
1 Mbps	56.80	0
2 Mbps	114.87	0

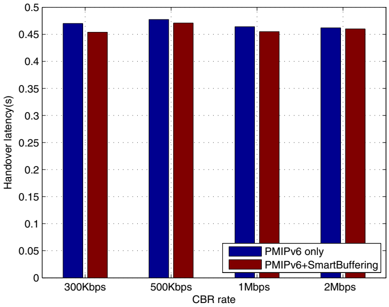


Fig. 3. Handover latency

4 Conclusion

In this paper, we proposed the Smart Buffering to support seamless handover in PMIPv6. The scheme buffers packets in a previous MAG without knowing target MAG and the time of the detachment of the MN. It also prevents over buffering by introducing lifetime to the buffered packets. All buffering and forwarding processes between the previous MAG and the new MAG are solely based on network-side information. So it well confirms the principle of PMIPv6 - the exclusion of MNs from the handover procedure. Simulation results show that the Smart Buffering prevents packet loss during the handover period and slightly improves the total handover latency.

References

[1] Gundavelli, S., Leung, K., Devarapalli, V., Chowdhury, K., Patil, B.: Proxy Mobile IPv6. draft-ietf-netlmm-proxymip6-17 (work in progress) (2008)

[2] Johnson, D., Perkins, C., Arkko, J.: Mobility Support in IPv6. IETF, RFC 3775 (2007)

[3] Koodli, R. (ed.): Fast Handovers for Mobile IPv6. IETF, RFC 4068 (2005)

[4] Xia, F., Sarikaya, B.: Mobile Node Agnostic Fast Handovers for Proxy Mobile IPv6. draft-xia-netlmm-fmip-mnagno-00 (work in progress) (2007)

[5] Yokota, H., Chowdhury, K., Koodli, R., Patil, B.: Fast Handovers for PMIPv6. draft-yokota-mipshop-pfmipv6-00 (work in progress) (2007)

Uniprocessor EDF Scheduling with Mode Change

Björn Andersson

IPP Hurray Research Group,
Polytechnic Institute of Porto, Portugal

Abstract. Consider the problem of scheduling sporadically-arriving tasks with implicit deadlines using Earliest-Deadline-First (EDF) on a single processor. The system may undergo changes in its operational modes and therefore the characteristics of the task set may change at run-time. We consider a well-established previously published mode-change protocol and we show that if every mode utilizes at most 50% of the processing capacity then all deadlines are met. We also show that there exists a task set that misses a deadline although the utilization exceeds 50% by just an arbitrarily small amount. Finally, we present, for a relevant special case, an exact schedulability test for EDF with mode change.

1 Introduction

Many real-time systems must reconfigure themselves during operation and thereby change the characteristics of their tasks. It is therefore crucial to (i) design a protocol (called a *mode-change protocol*) that prescribes how tasks are allowed to arrive during the reconfiguration and (ii) design a method (a schedulability test) for proving that deadlines are met during the reconfiguration. Unfortunately, no analysis of mode change with EDF scheduling on a single processor is available.

In this paper, we present a new solution to the mode change problem. We use the previously known, and well established, mode-change protocol designed by Sha et al. [1]. But we use EDF and we present a schedulability analysis for EDF with mode changes. The schedulability analysis is simple; if the utilization of every mode is at most 50% then all deadlines are met. We also show that there exists a task set that misses a deadline although the utilization exceeds 50% by just an arbitrarily small amount. Finally, we present, for a relevant special case, an exact schedulability test for EDF with mode change.

2 System Model

Consider a task set $\tau = \{\tau_1, \tau_2, \ldots\}$ and a mode set $modes = \{mode^1, mode^2, \ldots\}$. Also, consider a sequence of times of transition requests $< tr[1], tr[2], tr[3], \ldots >$ and corresponding new modes $< newmode[1], newmode[2], newmode[3], \ldots >$ where each of those modes are in the set modes. These two sequences have the

T.P. Baker, A. Bui, and S. Tixeuil (Eds.): OPODIS 2008, LNCS 5401, pp. 572–577, 2008.

interpretation that the current mode of the task set τ is requested to become $newmode[j]$ at time $tr[j]$. We assume that the request of the transition of the task set to mode $newmode[j]$ at time $tr[j]$ is unknown to the scheduling algorithm and mode change protocol before time $tr[j]$.

A task τ_i generates a (potentially infinite) sequence of jobs. We consider the sporadic model, that is, the time of the arrival of a job is unknown before the job arrives and the arrival time of a job cannot be controlled by the scheduling algorithm. A task τ_i has a current mode at time t; this mode is one of the modes in the set modes. A task τ_i is characterized by the minimum inter-arrival time of task τ_i in mode k (denoted T_i^k) and the execution time of task τ_i in mode k (denoted C_i^k). The parameters T_i^k and C_i^k have the following interpretation. If task τ_i is in mode $mode^k$ at time t and s denotes the latest time not exceeding t when task τ_i has arrived then it holds that the next arrival of task τ_i occurs at time $s + T_i^k$ or later. If task τ_i is in mode $mode^k$ at time t and task τ_i has never arrived before time t then it is possible for τ_i to arrive at time t. Let us consider a job of task τ_i that arrives at time s and the job is in mode k at that time, time s. Then the deadline of the job is $s + D_i^k$. If the job performs C_i^k time units of execution by its deadline then we say that the job meets its deadline; otherwise it misses its deadline. A task τ_i is said to meet its deadlines if all of its jobs meet their deadlines; otherwise the task τ_i is said to miss a deadline. A task set τ is said to meet its deadlines if all tasks in τ meet their deadlines otherwise we say that the task set τ misses a deadline. We assume $\forall i, k : D_i^k = T_i^k$ and we assume that preemptive Earliest-Deadline-First (EDF) scheduling is used to schedule jobs on a single processor

We say that the system is in steady state at time t if it holds that all tasks are in the same mode at time t. We say that the system is in transient state at time t if it is not in steady state at time t. We let $latest\_arrival(t, \tau_i)$ denote the maximum time such that (i) this time is no greater than t and (ii) task τ_i arrives at time t. If task τ_i has not yet arrived at time t, then $latest\_arrival(t, \tau_i)$ is undefined. Let us assume that the system has a variable $pending\_changes$, a set which is initialized to the empty set when the system boots.

If the system is in steady state at time t and all tasks are in mode k and t is one of the elements in the sequence $< tr[1], tr[2], \ldots >$, say $tr[j]$, then a mode change protocol will switch the mode of the tasks from mode k to mode $newmode[j]$. The tasks do not necessarily switch to the new mode immediately. The general rule for mode change when the system is in steady state at time t is as follows. If task τ_i arrives at time $tr[j]$ then task τ_i switches from mode k to $newmode[j]$ immediately on its arrival; otherwise task τ_i switches from mode k to $newmode[j]$ at time $latest\_arrival(t, \tau_i) + T_i^k$.

If the system is in transient state at time t and the task set τ is in mode k and t is one of the elements in the sequence $< tr[1], tr[2], \ldots >$, say $tr[j]$, then no mode change is performed when the system is in a transient state; instead the tuple ($tr[j]$, $newmode[j]$) becomes member of the set $pending\_changes$ and then immediately when the system enters steady state, the run-time system selects one (the application developer can choose which one) of the tuples in *pending_changes* (let us say that ($tr[q]$, $newmode[q]$) was selected) and then acts as if it was requested that the system changes to mode $newmode[q]$ at the time when the system entered steady state. And then the set *pending_changes* is assigned the empty set.

3 The Utilization Bound of EDF with Mode Change

In proofs, we will find it useful to discuss an algorithm called Processor-Sharing (PS). It operates as follows. Consider a time interval of duration $\epsilon > 0$ and assume that task τ_i is in the mode k during the entire time interval. Then it holds that τ_i executes for $(C_i^k/T_i^k) \cdot \epsilon$ time units during the time interval of duration ϵ.

Lemma 1. *Let $current\_modes(i, \tau)$ denote the set of modes of tasks in the task set τ at time t. It holds that $\forall t : current\_modes(i, \tau) \leq 2$.*

Proof. The lemma follows from the fact we do not (as stated in Section 2) allow a mode change when the system is in transient state.

Lemma 2. *If $\forall modes^k \in modes$ it holds that:*

$$\sum_{\tau_j \in \tau} \frac{C_j^k}{T_j^k} \leq \frac{1}{2} \tag{1}$$

and PS is used to schedule tasks then all deadlines are met.

Proof. From Lemma 1 and Equation 1 it follows that PS meets all deadlines.

Theorem 1. *If $\forall modes^k \in modes$ it holds that:*

$$\sum_{\tau_j \in \tau} \frac{C_j^k}{T_j^k} \leq \frac{1}{2} \tag{2}$$

and EDF is used to schedule tasks then all deadlines are met.

Proof. Follows from Lemma 2 and the fact that EDF is an optimal scheduling algorithm for a set of jobs [2]. (A scheduling algorithm is said to be optimal if it meets deadlines when it is possible to do so.)

The utilization bound expressed by Theorem 1 is tight; it can be seen by considering two tasks with T_i such that $T_1^1 + T_1^2 = T_2^1 + T_2^2$ and $T_1^1 = T_2^2$ and $T_1^2 = T_2^1$.

4 Schedulability Analysis of EDF with Mode Change

In this section, we will study schedulabiltiy analysis for the special case where (i) |modes|=2 and (ii) T_i^1,C_i^1, T_i^2,C_i^2 and $tr[1]$ are integers and arrivals occur only at times which are integers and (iii) only one mode change request can occur during a busy interval. (A busy interval is an interval such that the processor is busy during this interval and just before the interval, the processor is idle and just after the interval, the processor is idle as well.) We believe this limitation is reasonable for systems where reconfiguration is performed not too often but when reconfiguration is required, the reconfiguration must be completed quickly, for example reconfiguration after the occurrence of a fault.

Let us define $dbf(\ \tau,\ L)$ as:

$$dbf(\tau, L) = \sum_{\tau_j \in \tau} dbf(\tau_j, L) \tag{3}$$

We need to check whether for every $L > 0$, it holds that if $dbf(\ \tau,\ L) \leq L$.

Consider a time interval $[t_0,t_1)$ with $t_1 - t_0 = L$. The system is in $mode^1$ at time t_0 and at time $tr[1]$, there is a request to change to $newmode[1]$. Let $transition_j$ denote the time when task τ_j switches to mode $tr[1]$. We have that:

$$dbf(\tau, L) = \max_{t_0 \leq tr[1] \leq t_1} dbf(\tau, L, tr[1]) \tag{4}$$

where

$$dbf(\tau, L, tr[1]) = \sum_{\tau_j \in \tau} \Big(\lfloor \frac{transition_j - t_0}{T_j^1} \rfloor \cdot C_j^1 + \lfloor \frac{t_1 - transition_j}{T_j^2} \rfloor \cdot C_j^2 \Big) \tag{5}$$

and

$$\forall j : t_0 \leq transition_j \leq t_1 \tag{6}$$

and

$$\forall j : tr[1] \leq transition_j \tag{7}$$

and

$$\forall j : transition_j < tr[1] + T_j^1 \tag{8}$$

and

$$t_0 \leq tr[1] \leq t_1 \tag{9}$$

It is possible to compute $dbf(\tau, L)$ for a fixed L by solving the optimization problem expressed by Inequality 4 - Inequality 9. But recall that we need to calculate $dbf(\tau, L)$ for all positive values of L. For uniprocessor scheduling of EDF without mode change, it was shown [3] that if the utilization of the task set is known then one can find an upper bound on L such that values of L above this bound does not need to be checked. We will now develop a similar approach for EDF with mode change.

Lemma 3. *It holds that:*

$$dbf(\tau, L, tr[1]) \leq \Big(\sum_{j \in \tau} C_j^1 \Big) + L \cdot \max \Big(\sum_{j \in \tau} \frac{C_j^1}{T_j^1}, \sum_{j \in \tau} \frac{C_j^2}{T_j^2} \Big) \tag{10}$$

Proof. From Inequality 5,Inequality 7 and Inequality 8 it follows that:

$$dbf(\tau, L, tr[1]) \leq \sum_{\tau_j \in \tau} \Big(\lfloor \frac{tr[1] + T_j^1 - t_0}{T_j^1} \rfloor \cdot C_j^1 + \lfloor \frac{t_1 - tr[1]}{T_j^2} \rfloor \cdot C_j^2 \Big) \tag{11}$$

Using the facts that (i) $t_1{=}t_0{+}L$ and (ii) for every $x > 0$ it holds that $\lfloor x \rfloor \leq x$ and rewriting gives us:

$$dbf(\tau, L, tr[1]) \leq (tr[1] - t_0) \cdot \Big(\sum_{\tau_j \in \tau} (\frac{C_j^1}{T_j^1} - \frac{C_j^2}{T_j^2}) \Big) + (\sum_{\tau_j \in \tau} C_j^1) + L \cdot (\sum_{\tau_j \in \tau} \frac{C_j^2}{T_j^2}) \tag{12}$$

By observing the two cases where $\Big(\sum_{\tau_j \in \tau} (\frac{C_j^1}{T_j^1} - \frac{C_j^2}{T_j^2}) \Big)$ is non-positive and positive, we obtain that: Consider the term

$$\Big(\sum_{\tau_j \in \tau} (\frac{C_j^1}{T_j^1} - \frac{C_j^2}{T_j^2}) \Big) \tag{13}$$

If it is negative or zero then Inequality 12 is maximized for $tr[1] = t_0$ and hence for that case, an upper bound on Inequality 12 is:

$$(\sum_{\tau_j \in \tau} C_j^1) + L \cdot (\sum_{\tau_j \in \tau} \frac{C_j^2}{T_j^2}) \tag{14}$$

If Inequality 13 is positive then Inequality 12 is maximized for $tr[1] = t_0 + L$ and hence for that case, an upper bound on Inequality 12 is:

$$L \cdot \Big(\sum_{\tau_j \in \tau} (\frac{C_j^1}{T_j^1} - \frac{C_j^2}{T_j^2}) \Big) + (\sum_{\tau_j \in \tau} C_j^1) + L \cdot (\sum_{\tau_j \in \tau} \frac{C_j^2}{T_j^2}) \tag{15}$$

which can be rewritten to:

$$(\sum_{\tau_j \in \tau} C_j^1) + L \cdot (\sum_{\tau_j \in \tau} \frac{C_j^1}{T_j^1}) \tag{16}$$

Combining these cases gives us that:

$$dbf(\tau, L, tr[1]) \leq \Big(\sum_{j \in \tau} C_j^1 \Big) + L \cdot \max \Big(\sum_{j \in \tau} \frac{C_j^1}{T_j^1}, \sum_{j \in \tau} \frac{C_j^2}{T_j^2} \Big) \tag{17}$$

This states the lemma.

Lemma 4. *It holds that:*

$$dbf(\tau, L) \leq \Big(\sum_{j \in \tau} C_j^1 \Big) + L \cdot \max \Big(\sum_{j \in \tau} \frac{C_j^1}{T_j^1}, \sum_{j \in \tau} \frac{C_j^2}{T_j^2} \Big) \tag{18}$$

Proof. Follows from Inequality 4 and Lemma 3.

Lemma 5. *If*

$$\max\left(\sum_{j\in\tau}\frac{C_j^1}{T_j^1},\sum_{j\in\tau}\frac{C_j^2}{T_j^2}\right) < 1 \tag{19}$$

then for all $L > 0$ *such that*

$$\frac{\sum_{j\in\tau} C_j^1}{1-\max\left(\sum_{j\in\tau}\frac{C_j^1}{T_j^1},\sum_{j\in\tau}\frac{C_j^2}{T_j^2}\right)} \leq L \tag{20}$$

it holds that:

$$dbf(\tau, L) \leq L \tag{21}$$

Proof. Follow the spirit of the proof in [3] but use Lemma 4.

We now know that only values of L that do not exceed the left-hand side of Inequality 20 must be checked. We can enumerate all values of L from 1 up to this bound. We can also assume (with no loss of generality) that $t_0 = 0$ and $t_1 = L$ and hence enumerate $tr[1]$ from 0 up to L. For each of these we would like to compute $dbf(\ \tau,L,tr[1])$, using Inequality 5. This can be done by iterating, for each task τ_i, through all values of $transition_j$ from $tr[1]$ to $tr[1]+T_j^1-1$ and compute the term in the sum of Inequality 5; for convenience this term is stated below. to $tr[1]+T_j^1-1$ and compute the term in the sum of Inequality 5.

Acknowledgements

This work was partially funded by the Portuguese Science and Technology Foundation (Fundação para a Ciência e a Tecnologia - FCT) and the ARTIST2 Network of Excellence on Embedded Systems Design.

References

[1] Sha, L., Rajkumar, R., Lehoczky, J., Ramamritham, K.: Mode Change Protocol for Priority-Driven Preemptive Scheduling. Carnegie-Mellon University Pittsburgh, Pennsylvania, Software Engineering Institute (November 1988)

[2] Dertouzos, M.L.: The Procedural Control of Physical Processes. IFIP Congress, Stockholm, Sweden (1974)

[3] Baruah, S., Howell, R., Rosier, L.: Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems 2, 301–324 (1990)

Author Index