Nancy A. Lynch
Alexander A. Shvartsman (Eds.)

LNCS 6343
ARCoSS

Distributed Computing

24th International Symposium, DISC 2010
Cambridge, MA, USA, September 2010
Proceedings

Springer

Lecture Notes in Computer Science 6343

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

Advanced Research in Computing and Software Science
Subline of Lectures Notes in Computer Science

Nancy A. Lynch
Alexander A. Shvartsman (Eds.)

Distributed Computing

24th International Symposium, DISC 2010
Cambridge, MA, USA, September 13-15, 2010
Proceedings

Springer

Volume Editors

Nancy A. Lynch
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
77 Massachusetts Avenue, Cambridge, MA 02139-4307, USA
E-mail: lynch@theory.csail.mit.edu

Alexander A. Shvartsman
University of Connecticut, Computer Science and Engineering
371 Fairfield Way, Unit 2155, Storrs, CT 06269, USA
E-mail: aas@cse.uconn.edu

Library of Congress Control Number: 2010933792

CR Subject Classification (1998): C.2.4, C.2, H.4, D.2, H.3, I.2.11

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

ISSN 0302-9743
ISBN-10 3-642-15762-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-15762-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

DISC, the International Symposium on DIStributed Computing, is an international forum on the theory, design, analysis, implementation and application of distributed systems and networks. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

This volume contains the papers presented at DISC 2010, the 24th International Symposium on Distributed Computing, held on September 13–15, 2010 in Cambridge, Massachusetts. The volume also includes the citation for the 2010 Edsger W. Dijkstra Prize in Distributed Computing, jointly sponsored by DISC and PODC (the ACM Symposium on Principles of Distributed Computing), which was presented at PODC 2010 in Zurich to Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg for their work on failure detectors.

There were 135 papers submitted to the symposium (in addition there were 14 abstract-only submissions). The Program Committee selected 32 contributions out of the 135 full-paper submissions for regular presentations at the symposium. Each presentation is accompanied by a fifteen-page paper in this volume. Every submitted paper was read and evaluated by at least three members of the Program Committee. The committee was assisted by more than 120 external reviewers. The Program Committee made its final decisions during the electronic meeting held on June 18–29, 2010. Revised and expanded versions of several selected papers will be considered for publication in a special issue of the journal *Distributed Computing*.

The program also included three invited lectures by Rachid Guerraoui (EPFL, Switzerland), Barbara Liskov (MIT, USA), and Nitin Vaidya (University of Illinois, USA).

The Best Student Paper Award was presented to François Bonnet for the paper "Anonymous Asynchronous Systems: the Case of Failure Detectors," coauthored with Michel Raynal.

The Program Committee also considered over 30 papers for brief announcements among the papers that generated substantial interest from the members of the committee, but that could not be accepted for regular presentations. This volume contains 14 invited brief announcements. Each two-to-three page announcement presents ongoing work or recent results, and it is expected that these results will appear as full papers in other conference proceedings or journals.

There were three workshops co-located with DISC this year: the *2nd Workshop on Theoretical Aspects of Dynamic Distributed Systems* on September 12th, and the *2nd Workshop on the Theory of Transactional Memory* and the *6th International Workshop on Foundations of Mobile Computing* on September 16th.

September 2010

Nancy Lynch
Alexander Shvartsman

Symposium Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for the presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biennial International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chairs

Nancy Lynch	MIT, USA
Alexander Shvartsman	University of Connecticut, USA

Program Committee

Marcos K. Aguilera	Microsoft Research Silicon Valley, USA
Soma Chaudhuri	Iowa State University, USA
Bogdan Chlebus	University of Colorado Denver, USA
Gregory Chockler	IBM Research, Israel
Rui Fan	The Technion, Israel
Pascal Felber	University of Neuchatel, Switzerland
Paola Flocchini	University of Ottawa, Canada
Pierre Fraigniaud	CNRS and Univ. Paris Diderot, France
Petr Kuznetsov	TU Berlin/Deutsche Telekom Lab., Germany
Dariusz Kowalski	University of Liverpool, UK
Fabian Kuhn	University of Lugano, Switzerland
Victor Luchangco	Sun Microsystems Labs, USA
Yoram Moses	The Technion, Israel
Peter Musial	University of Puerto Rico Rio Piedras, USA
Michel Raynal	IRISA, France
Andrea Richa	Arizona State University, USA
Paul Spirakis	Research Acad. Computer Tech. Inst., Greece
Robbert van Renesse	Cornell University, USA
Jennifer Welch	Texas A&M University, USA
Shmuel Zaks	The Technion, Israel

Steering Committee

Antonio Fernandez Anta	Universidad Rey Juan Carlos, Spain
Chryssis Georgiou	University of Cyprus
Idit Keidar	The Technion, Israel
Andrzej Pelc	University of Quebec, Canada
Sergio Rajsbaum	UNAM, Mexico
Nicola Santoro (Chair)	Carleton University, Canada
Gadi Taubenfeld	IDC Herzliya, Israel

Local Organization

Tigran Anotnyan	University of Connecticut, USA
Seda Davtyan	University of Connecticut, USA
Nancy Lynch	MIT, USA
Peter Musial	University of Puerto Rico Rio Piedras, USA
Nicolas Nicolaou	University of Connecticut, USA
Alexander Shvartsman	University of Connecticut, USA
Ealine Sonderegger	University of Connecticut, USA
Therese Smith	University of Connecticut, USA

External Reviewers

Ittai Abraham
Dan Alistarh
Zakia Asad
James Aspnes
Balasingham Balamohan
Leonid Barenboim
Alysson Bessani
Martin Biely
Paolo Boldi
Borzoo Bonakdarpour
Armando Castaneda
Claris Castillo
Jeremie Chalopin
Ching-Lueh Chang
Ioannis Chatzigiannakis
Hana Chockler
Vicent Cholvi
Hyun-Chul Chung
Alejandro Cornejo
Andrzej Czygrinow
Jurek Czyzowicz
Luke Dalesandro
Shantanu Das
Peleg David
Seda Davtyan
Xavier Defago
Carole Delporte
Partha Dutta
Michael Elkin
Faith Ellen
Yuval Emek
Michael Fischer
Michele Flammini
Dimitris Fotakis
Juan Garay
Leszek Gasieniec
Cyril Gavoille
Chryssis Georgiou
Seth Gilbert
Sarunas Girdzijauskas
Noam Gordon
Maria Gradinariu Potop-Butucaru
Vincent Gramoli
Fabiola Greve

Rachid Guerraoui
Tim Harris
Avinatan Hassidim
Danny Hendler
Maurice Herlihy
Ted Herman
Ezra Hoch
Prasad Jayanti
Colette Johnen
Michal Kapalka
Idit Keidar
Barbara Keller
Roger Keller
Majid Khabbazian
Marek Klonowski
Kishori Konwar
Amos Korman
Miroslaw Korzeniowski
Adrian Kosowski
Rastislav Kralovic
Geetika T. Lakshmanan
Hyunyoung Lee
Vasiliki Liagkou
Andrzej Lingas
Yung-Hsiang Lu
Dahlia Malkhi
Yishay Mansour
Virendra Marathe
Euripides Markou
Maged Michael
Othon Michail
Alessia Milani
Sayan Mitra
Neeraj Mittal
Mark Moir
Angelo Monti
Luca Moscardelli
Thomas Moscibroda
Miguel Mosteiro
Chet Murthy
Georgios Mylonas
Gil Neiger
Calvin Newport
Nicolas Nicolaou
Sotiris Nikoletseas
Nikos Ntarmos
Doron Nussbaum
Edusmildo Orozco
Humberto Ortiz-Zuazaga
Rotem Oshman
Panagiota Panagopoulou
Andrzej Pelc
Dmitri Perelman
Mia Persson
Frank Petit
Giuseppe Prencipe
Vivien Quema
Torvald Riegel
Etienne Riviere
Peter Robinson
Mariusz Rokicki
Jared Saia
Livia Sampaio
Srikanth Sastry
Christian Scheideler
Stefan Schmid
Nir Shavit
Therese Smith
Elaine Sonderegger
Mike Spreitzer
Yannis Stamatiou
Aaron Sterling
Adi Suissa
Gilles Tredan
Yih-Kuen Tsay
Mark Tuttle
Julian Velev
Ymir Vigfusson
Saira Viqar
Marko Vukolic
Jiaqi Wang
Roger Wattenhofer
Prudence Wong
Masafumi Yamashita
Piotr Zielinsky

Sponsoring Organizations

European Association for
Theoretical Computer Science

Computer Science Department of the
University of Puerto Rico Rio Piedras

Booth Engineering Center for
Advanced Technology at UCONN

VeroModo, Inc.

DISC 2010 acknowledges the use of the EasyChair system for handling submissions, managing the review process, and helping compile these proceedings.

Table of Contents

Brief Announcements I (Session 1d)

Wireless Networks (Session 1e)

Brief Announcements II (Session 1f)

Invited Lecture II: Best Student Paper (Session 2a)

Consensus and Leader Election (Session 2b)

Mobile Agents (Session 2c)

Invited Lecture III: Wireless Networks (Session 3a)

Computing in Wireless and Mobile Networks (Session 3b)

Brief Announcements III (Session 3c)

Modeling Issues and Adversity (Session 3d)

Self-stabilizing and Graph Algortihms (Session 3e)

Brief Announcements IV (Session 3f)

The 2010 Edsger W. Dijkstra Prize in Distributed Computing

The ACM-EATCS Edsger W. Dijkstra Prize in Distributed Computing was created to acknowledge outstanding papers on the principles of distributed computing whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade.

The Prize is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). This award is presented annually, with the presentation taking place alternately at PODC and DISC.

The 2010 Dijkstra Prize Committee, composed of Nancy Lynch (Co-Chair), Alexander Shvartsman (Co-Chair), James Anderson, James Aspnes, Pierre Fraigniaud, Rachid Guerraoui, and Maurice Herlihy, has selected

Tushar D. Chandra, **Vassos Hadzilacos**, and **Sam Toueg**,

to receive the 2010 Edsger W. Dijkstra Prize in Distributed Computing for the following two outstanding papers:

> Tushar D. Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems, *Journal of the ACM*, 43(2):225-267, 1996. (The first version appearing in the *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 1991.)
>
> Tushar D. Chandra, Vassos Hadzilacos and Sam Toueg. The Weakest Failure Detector for Solving Consensus, *Journal of the ACM*, 43(4):685-722, 1996. (The first version appearing in the *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, 1992.)

This pair of papers has had a deep impact on research in distributed computing. The work introduces and defines the notion of (unreliable) failure detectors in a distributed system, establishing a theory of failure detectors grounded on a general and precise framework. These papers have greatly influenced how one can reason about and deal with failures in distributed systems.

A failure detector is defined as an abstraction that provides each process with information about failures. The information may have different levels of accuracy and completeness, leading to different failure detectors, each precisely defined by properties relating the pattern of actual failures to the information provided by the failure detector. This modular approach separates definition from use and implementation, and leads to an elegant framework for developing algorithms and failure detector implementations, and for understanding the failure information and synchrony needed to solve distributed problems. The work has identified, in particular, the weakest failure detector for solving the consensus problem in an asynchronous system subject to failures. This work has been very

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 1–2, 2010.

influential. It has led to the development of new failure-detector-based protocols for solving consensus; to the definition of new failure detectors to tackle other distributed computing problems; to the study of various weak forms of partial synchrony as a means to obtain failure information; to the specification of the quality-of-service of failure detection; to the formulation of lower bounds of failure information needed to solve problems; and to a better understanding of the inherent hardness of solving distributed computing problems in the presence of failures. The failure detector approach has been adopted by the community to tackle many distributed problems, including group membership, atomic commit, various broadcast types, reliable communication, set agreement, mutual exclusion, transactional-memory contention management, leader election, and dining philosophers.

To summarize, this pioneering work introduces a general and elegant framework and theory for failure detectors; it establishes failure detectors as a first-class abstraction; and it proposes an approach to investigate problems based on this abstraction. The proposed approach has broadened the scope of research in distributed computing, leading to the study and understanding of an impressive variety of fundamental problems in distributed systems subject to failures.

Dr. Marcos K. Aguilera
Microsoft Research Silicon Valley
USA

Prof. Michel Raynal
IRISA, University of Rennes
France

The Power of Abstraction
(Invited Lecture Abstract)

Barbara Liskov

Massachusetts Institute of Technology

Abstraction is at the center of much work in Computer Science. It encompasses finding the right interface for a system as well as finding an effective design for a system implementation. Furthermore, abstraction is the basis for program construction, allowing programs to be built in a modular fashion. This talk will discuss how the abstraction mechanisms we use today came to be, how they are supported in programming languages, and some possible areas for future research.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, p. 3, 2010.

Fast Asynchronous Consensus with Optimal Resilience

Ittai Abraham, Marcos K. Aguilera, and Dahlia Malkhi

Microsoft Research Silicon Valley

Abstract. We give randomized agreement algorithms with constant expected running time in asynchronous systems subject to process failures, where up to a minority of processes may fail. We consider three types of process failures: crash, omission, and Byzantine. For crash or omission failures, we solve consensus assuming private channels or a public-key infrastructure, respectively. For Byzantine failures, we solve weak Byzantine agreement assuming a public-key infrastructure and a broadcast primitive called weak sequenced broadcast. We show how to obtain weak sequenced broadcast using a minimal trusted platform module. The presented algorithms are simple, have optimal resilience, and have optimal asymptotic running time. They work against a sophisticated adversary that can adaptively schedule messages, processes, and failures based on the messages seen by faulty processes.

1 Introduction

In the consensus problem, each process starts with some initial value and must make an irrevocable decision on one of the initial values, such that all correct processes decide on the same value. The challenge lies in solving consensus in the presence of faulty processes. In this paper we consider asynchronous message-passing systems, where processes communicate by sending messages, and there are no bounds on message delays or on the relative speed of processes. In such a system, consensus cannot be solved [21], but it *can* be solved if processes need to terminate only with probability one [3].

Chor, Merritt, and Shmoys [11] gave an algorithm that solves consensus with probability one in constant expected time, in a system with crash failures and $n \geq \frac{3+\sqrt{5}}{2}f+1 \approx 2.62f+1$, where n is the number of processes and f is the maximum number of faulty processes. Subsequently, Attiya and Welch [2] gave an algorithm based on an observation of Gafni, which works with the optimal resiliency of $n \geq 2f+1$, but the algorithm requires a message-independent adversary—one that acts independently of the content of messages.

Our first result is an algorithm that works in a system with $n \geq 2f+1$ and a rather sophisticated adversary that can adaptively schedule messages, processes, and failures based on the message contents seen by faulty processes. The algorithm relies on private channels. It uses a secret sharing scheme and a simple *binding gather* primitive, explained in Section 4.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 4–19, 2010.

Theorem 1. Consensus can be solved by an algorithm with constant expected running time in an asynchronous system with private channels, a sophisticated adversary, and $n \geq 2f+1$ processes, where f processes may crash.

Our result for crash failures can be easily extended to (general) omission failures, assuming the presence of a public-key infrastructure (PKI). Roughly speaking, a PKI provides a public-key cryptographic system in which each process p_i has a secret decryption function D_i and secret signing function S_i, and all processes know the public encryption function E_j and the public one-way signature verification function V_j of all other processes p_j. Except with negligible probability, signatures cannot be forged and encrypted messages cannot be decrypted by a party that does not have the secret signing or decryption functions. A PKI can be implemented under suitable cryptographic assumptions. By using a PKI, we require the adversary to be computationally bounded and our algorithms have a negligible probability of not terminating due to the adversary breaking the PKI.

Theorem 2. Consensus can be securely implemented by an algorithm with constant expected running time in an asynchronous system with a computationally-bounded sophisticated adversary, a public-key infrastructure, and $n \geq 2f+1$ processes, where f processes may experience omission failures.

Our next result concerns Byzantine failures, and it builds upon our result for omission failures. A simple partitioning argument shows that Byzantine agreement (consensus) cannot be solved with $2f+1 \leq n \leq 3f$, even with a PKI. Hence, we augment the system with a primitive called *weak sequenced broadcast*. Roughly speaking, weak sequenced broadcast is a broadcast that ensures that (a) messages from a given sender are delivered by correct processes in the same order; this is an ordering *per-sender*, similar to FIFO broadcast, and (b) if the sender is correct then eventually all processes will receive all its messages. We show one way to implement weak sequenced broadcast by using a minimal trusted platform module, a component which is becoming standard.

Even with weak sequenced broadcast, Byzantine agreement cannot be solved with $2f+1 \leq n \leq 3f$, because correct processes may decide a value that is not one of their initial values, even if their initial values are all the same, violating the Validity property of Byzantine agreement. So we require the Validity property to hold only if all processes are correct, and therefore we solve *weak Byzantine agreement* [26]. The algorithm has constant expected running time in a system with weak sequenced broadcast and $n \geq 2f+1$. It borrows from the approach of Feldman and Micali [18,19] based on verifiable secret sharing (VSS). Using weak sequenced broadcast, we can implement several building blocks like sequenced broadcast (which we define later) and VSS. We are not aware of other work for asynchronous systems with Byzantine failures and $n \geq 2f+1$.

Theorem 3. Weak Byzantine agreement can be securely implemented by an algorithm with constant expected running time in an asynchronous system with a computationally-bounded sophisticated adversary, a public-key infrastructure, weak sequenced broadcast, and $n \geq 2f+1$ processes, where f processes may be Byzantine.

Weak sequenced broadcast itself may be implemented in many ways, one of which is demonstrated here by using a minimal trusted platform module (TPM). The TPM has a register which can be updated only by invoking guarded functions of the module, and stores a private key which is used for signing the register value upon request. This system model is useful when one can trust the hardware platform but the system is otherwise vulnerable (say, to malware). But in some other settings, the hardware of some machines may not be trusted. In this case, we can use an external service to aid in implementing weak sequenced broadcast, e.g., using a public certification authority [22] or other means.

All the algorithms we present have optimal resilience and asymptotically optimal (constant) expected running time. They are based on a suitable combination of known techniques as we explain below. Due to space limitations, the proofs of our results are omitted; they will be included in the full version.

Related work. Our work touches one of the most active areas of research in distributed computing, and it is therefore beyond our scope to cover all related work. Below, we cover the most relevant results and we refer the reader to [28,2] for an extensive treatment.

Byzantine failures. There have been proposals to solve Byzantine agreement in a system with $n \geq 2f+1$ using strong primitives such as an *append-only log* [12] or trusted increment [27]. The append-only log is similar to weak sequenced broadcast. However, these works differ from ours because they consider partially synchronous systems, that is, the liveness of the algorithms is conditional on eventual synchrony. In contrast, we consider a fully asynchronous system and provide randomized algorithms that terminate with probability one (except for a negligible probability that the cryptosystem is broken). Our implementation of weak sequenced broadcast using TPM's is similar to the scheme by [25]. A formal specification of TPM's is given in [31]. The idea of using strong primitives to boost the resilience of Byzantine agreement algorithms has been suggested in the wormhole approach of Correia et al [13]. Our solution for the Byzantine case uses a simple asynchronous VSS protocol for $n \geq 2f+1$, which is based on zero knowledge and weak sequenced broadcast. Our asynchronous VSS protocol is similar to the PVSS protocol of [30] (but they have not considered asynchronous VSS or use weak sequenced broadcast). Other asynchronous VSS protocols work only for $n \geq 3f+1$ [9,6].

A public-key infrastructure has often been used for reaching agreement. In *synchronous* systems with Byzantine failures, Dolev and Strong [15] show how to solve terminating reliable broadcast for $n \geq f+1$. Katz and Koo [24] show how to solve Byzantine agreement for $n \geq 2f+1$ in constant expected time [24] (also see [17]). Our asynchronous randomized probabilistic agreement primitive is similar to the leader based approach of [24]. In *asynchronous systems*, [8] shows how to solve Byzantine agreement for $n \geq 3f+1$ in constant expected time with asymptotically optimal message complexity.

Without computational assumptions, Byzantine agreement with probability one in an asynchronous system is given by Ben-Or [3] for $n \geq 5f+1$ and by

Bracha [5] for $n \geq 3f+1$. However, these algorithms take expected exponential time for decision. Our solution is based on the approach of [3].

Algorithms for solving Byzantine agreement in constant expected time are given by Feldman and Micali [18,19] for $n \geq 3f+1$ for the synchronous model and $n \geq 4f+1$ for the asynchronous model [17]. For $n \geq 3f+1$ and an asynchronous model, [9,29] solve Byzantine agreement with probability $1-\epsilon$ and, conditional on success, processes terminate in constant expected number of rounds, while [1] solves Byzantine agreement with probability one termination and a polynomial expected running time. If $n \leq 3f$, randomized Byzantine agreement is impossible in an asynchronous system even with a PKI. It is also impossible in a synchronous system without a PKI [20]. [4] solves Byzantine agreement in expected constant rounds in an asynchronous system with $n \geq 5f+1$. This protocol has $O(\log n)$ communication complexity per message; however, it assumes a trusted dealer and does not obtain optimal resilience. Our solution, like those of [18,11,19,9,29,1], require polynomial communication complexity per message.

Benign failures. [11] addresses the problem of solving consensus in constant expected time with crash or omission failures. There are algorithms for *synchronous systems* and $n \geq 2f+1$, and for *asynchronous systems* and $n \geq \frac{3+\sqrt{5}}{2}f+1$. An algorithm for asynchronous systems and $n \geq 2f+1$ is given in [2] using a *get-core* primitive suggested by Gafni, but this algorithm requires a message-independent adversary. Our solutions use a similar structure and primitives as [11,2]. The main differences are the following: (1) to handle stronger adversaries we need a stronger *binding gather* primitive; (2) we use probabilistic agreement to solve multi-valued consensus while [11,2] use a common coin to just solve binary consensus; (3) to handle stronger adversaries, we use verifiable secret sharing; (4) we give an algorithm that tolerates Byzantine failures. For asynchronous systems, it is conjectured that the protocol of [1] solves consensus for $n \geq 3f+1$ in constant expected time (without a PKI) for a message-dependent adversary and omission failures. Our work differs because we give algorithms for *asynchronous systems* and $n \geq 2f+1$. Consensus can also be solved with $n \geq 2f+1$ in partially synchronous systems [14,16], or in systems with failure detectors [10].

2 Model

We consider a system with n processes denoted $p_1, \ldots, p_n$ that can communicate with each other via point-to-point messages. The system is *asynchronous* meaning that there are no bounds on message delays or on the relative speed of processes. Processes have access to a source of uniformly random bits. The system is subject to process failures, and we consider several possibilities:

1. Crash failures. A process may fail by crashing, that is, it stops taking steps. Communication between every pair of processes is reliable. More precisely, the following properties are satisfied for every processes p_i and p_j:

- *Integrity.* If p_i receives a message m from p_j exactly k times by time t then p_j sent m to p_i at least k times before time t.

– *No Loss.* If p_j does not crash and p_i sends m to p_j exactly k times by time t then p_j eventually receives m from p_i at least k times.

2. Omission failures. A faulty process may experience omission failures, in which it fails to send or receive messages. More precisely, for every processes p_i and p_j, the Integrity property above holds, but the No Loss property is guarantees to hold only if p_i and p_j are not faulty.

3. Byzantine failures. A faulty process may behave arbitrarily, including deviating from its code. The Integrity and No Loss properties hold for every pair of *correct* processes p_i and p_j. If p_i or p_j is Byzantine, neither property may hold.

If a process never becomes faulty we say that it is *correct*.

Power of adversary. When designing fault-tolerant algorithms, we often assume that an intelligent adversary has some control of the system: it may be able to control the occurrence and the timing of process failures, the message delays, and the scheduling of processes. Adversaries may have limitations on their computing power and on the information that they can obtain from the system. Different algorithms are designed to defeat different types of adversaries. The simplest adversary is the *message-oblivious adversary*, which cannot look at the internal state of processes or the contents of messages. Our algorithms can defeat a stronger adversary, called the *sophisticated adversary*, which we now describe. At any point in the execution, the adversary may choose to make a process faulty, provided that at most $f < n/2$ processes are faulty. A faulty process may not exhibit faulty behavior immediately: with Byzantine failures, a faulty process may continue to behave well for a while; with omission failures, a faulty process may continue to handle messages without loss; with crash failures, a faulty process may continue to execute for a while. The adversary has full knowledge of the internal state of a faulty process. In particular, it knows all the messages that it sends and receives. With this information, at any time in the execution, the adversary can dynamically select which process takes the next step and which message this process receives (if any). The adversary, however, operates under the following restrictions: the final schedule must fair, meaning that all correct processes take infinitely many steps, and the messages sent and received must satisfy the Integrity and No Loss property according to the type of failure considered (crash, omission, or Byzantine) as described above. In some cases, we consider a computationally-bounded adversary, which has the additional requirement that its computation is limited to polynomially-bounded functions.

3 Problem

We are interested in solving consensus and probabilistic agreement, which we now define.

Consensus. Each process starts with some initial value and must decide on a single value. In the classical consensus problem, the following must hold:

- *Validity.* If a correct process decides on a value v then v is one of the initial values.
- *Agreement.* No two correct processes decide differently.
- *Termination.* Every correct process eventually decides.

Consensus cannot be solved in an asynchronous system subject to failures [21], so we consider the following weakening of termination:

- *Termination with probability one.* With probability one, all correct processes eventually decide.

We are interested in *fast* algorithms, which we define to be algorithms in which correct processes decide in *constant expected time*.

For a system with Byzantine failures, we consider a weakening of consensus called weak Byzantine agreement, obtained by replacing the validity property with the following:

- *No-Failure Validity.* If all processes are correct and a correct process decides on a value v then v is one of the initial values.[1]

Probabilistic agreement. Probabilistic agreement is a variant of consensus in which processes may decide different values with some probability smaller than one. More precisely, we require the Validity and Termination properties as defined above, as well as the following:

- *Uncertain agreement.* With probability at least $\rho > 0$, no two correct processes decide differently.

Here $\rho > 0$ is some constant. For practical purposes it should be large, say $1/3$.

For a system with Byzantine failures, we consider a variant called *certified probabilistic agreement*, in which the initial value of a process (whether correct or Byzantine) is certified by some computationally unforgeable means, as we later explain. Algorithms will use this certification to provide the validity property.

Cryptographic primitives. The use of cryptography in algorithms can create a negligible probability of failure of the algorithm, including non-termination, due to the adversary breaking the cryptographic primitives by randomly guessing keys. Technically, we say that an algorithm *securely implements* consensus (instead of "implements consensus") to indicate this negligible probability of failure.

4 Binding Gather

Our fast algorithms for benign failures are based on a simple primitive called *binding gather*. All correct processes invoke *binding gather*(v) with some input value v, and the primitive returns as output a *set* of values, such that the following holds:

[1] This property implies the standard validity property of weak Byzantine agreement [26], which states that if all processes are correct and they have the same initial value v then no correct process decides on a value other than v.

- *Validity.* Every value in every output set is the input value of some process.
- *Binding Commonality.* There exists some value v such that v is in every output set, and v can be determined by an external observer when the first process outputs its set.
- *Termination.* All correct processes eventually output some non-empty set of values.

A value that is in every output set is called a *common value*. The binding gather primitive is easy to implement in a system with crash or omission failures using three asynchronous rounds of a full-information protocol (Section 6.3 extends the primitive and implementation to Byzantine failures):

Round 1. p_i sends its input to all, waits for $n-f$ values, and stores it in set $seen_i$.

Round 2. p_i sends set $seen_i$ to all, waits for $n-f$ sets, and stores their union in set $seenmore_i$.

Round 3. p_i sends set $seenmore_i$ to all, waits for $n-f$ sets, and returns the union of these sets.

By using another round of a full-information protocol (a total of four rounds), we can ensure that there exists a set of $n-f$ common values. A similar three round protocol called *get-core* appeared in [2]: it obtains a set of $n-f$ common values where each common value has the first and third properties of *binding gather* but does not provide Binding Commonality, a property needed against sophisticated adversaries. If we used *get-core*, a sophisticated adversary could influence which values appear in the common set, and this would break our running time guarantees.

5 Algorithms for Crash Failures

We now present algorithms for consensus and probabilistic agreement that tolerate crash failures and terminate in constant expected time, assuming $n \geq 2f+1$. The algorithm for consensus uses probabilistic agreement as a building block, so we start with the latter.

5.1 Fast Probabilistic Agreement

The fast algorithm for probabilistic agreement is similar to the algorithms in [18,24], except that we use binding gather to make it work for $n \geq 2f+1$ with crash failures. Each process p_i starts with an initial value v_i. The rough idea of the algorithm is that each process p_i picks a random rank for itself (a rank is a number) and sends to all a message containing its rank and v_i. A process collects $n-f$ such pairs and calls binding gather to share its set of $n-f$ pairs. Binding gather will return a set of such sets, with the property that at least one set C of $n-f$ pairs is returned to every process. Each process looks at all pairs in all sets that it gets from binding gather, finds the largest rank, and decides on the value associated with that rank. If the ranks are uniformly random, the true largest rank M will be in C with probability at least $(n-f)/n \geq 1/2$, which

will cause all processes to pick M and hence decide on the same value. One technicality is that the ranks need to be picked from a bounded interval, and there is a probability that two different processes pick the same rank. We must choose the interval to be large enough $(0 \ldots n^4{-}1)$ so that this collision happens with probability less than $1/n^2$. The algorithm is given in Figure 1.

Process p_i has initial value v_i and executes the following code:

1. *rank*$_i$:= random number between 0 and $n^2 - 1$
2. send $(\textit{rank}_i, v_i)$ to all
3. wait to receive $(\textit{rank}_j, v_j)$ from $n{-}f$ processes j
4. R_i := set of received $(rank_j, v_j)$'s
5. $V_i := \cup$*binding-gather*(R_i) (* *binding-gather* outputs a set of sets; flatten it out to V_i *)
6. *max_rank* := $\max\{\textit{rank} : (\textit{rank}, *) \in V_i\}$
7. choose v such that $(\textit{max\_rank}, v) \in V_i$
8. *decide*(v)

Fig. 1. Fast algorithm for probabilistic agreement with crash failures, $n \geq 2f{+}1$, and a message-oblivious adversary

Process p_i has initial value v_i and executes the following code:

(* *part 1: send values and shares* *)
1. $r_{i1}, \ldots, r_{in}$:= pick n random numbers between 0 and $n^4 - 1$
2. for $j = 1..n$ do *dealer-share*$^{ij}(r_{ij})$
3. for $j, k = 1..n$ do fork *share*$^{jk}()$ (* run share protocol; it may not return if dealer crashes, so fork *)
4. wait for *share*^{ji} to return for $n{-}f$ distinct j's, and let S_i be the set of those j's (* processes in S_i are those that contribute to p_i's rank *)
5. send (S_i, v_i, i) to all
6. wait to receive (S_j, v_j, j) from $n{-}f$ processes; let J_i be the set of such messages (* J_i has data of processes whose rank can be retrieved *)

(* *part 2: gather* *)
7. *bigJ* := $\cup$*binding-gather*(J_i) (* *binding-gather* outputs a set of sets; flatten it *)
8. $P := \{j : (*, *, j) \in \textit{bigJ}\}$ (* P contains processes whose rank can be retrieved *)
9. for each $(S, v, j) \in$ *bigJ* do $(S_j, v_j) := (S, v)$ (* extract S_j and v_j *)

(* *part 3: recover secrets* *)
10. for each $j, k = 1..n$ do fork { wait for *share*^{jk} to return; r_{jk} := *recover*$^{jk}()$ } (* *share*^{jk} or *recover*^{jk} may not return, so do in background *)
11. for each $j \in P$ do
12. wait for *recover*^{kj} to return for all $k \in S_j$
13. *rank*$_j$:= $\sum_{k \in S_j} r_{kj} \bmod n^4$ (* recover j's rank *)

(* *part 4: choose winner and decide* *)
14. *winner* := *argmax*$_j\{\textit{rank}_j : j \in P\}$ (* winner is process with highest rank *)
15. *decide*$(v_{\textit{winner}})$

Fig. 2. Fast algorithm for probabilistic agreement with crash failures, $n \geq 2f{+}1$, and a sophisticated adversary

This algorithm works with the simple message-oblivious adversary, but it does not quite work with the sophisticated adversary: after the adversary learns the largest rank M, it can coordinate the execution of binding gather to ensure that C does not include M. To solve this problem, we use secret sharing to hide the values of the ranks from the adversary until the common value of binding gather has been determined.

The full algorithm is given in Figure 2. It uses binding gather and n^2 copies of the secret sharing primitives denoted *dealer-share*^{ij}, *share*^{ij}, and *reconstruct*^{ij} where $i, j = 1..n$ The protocol has four parts. In the first part, p_i picks n random values $r_{i1}, \ldots, r_{in}$. The idea is that we want to hide p_i's rank from p_i itself, because p_i could be a faulty process that the adversary has access to. We use the idea in Feldman and Micali's protocol [18]: each process will pick a uniform random value between 0 and n^4-1 and p_i's rank will be the sum of $n-f$ such values (modulo n^4), which is also uniformly distributed. Thus, the random value r_{ij} that p_i picks is p_i's contribution to the rank of p_j. Process p_i then uses secret sharing to distribute shares of the r_{ij}'s to all processes. It then waits to receive shares of r_{ji} for $n-f$ values of j; we let S_i denote those values of j. Intuitively, S_i are those j's that will contribute to the rank of p_i. Process p_i then sends (S_i, v_i, i) to all, where v_i is its initial input; other processes will use S_i to reconstruct the rank of p_i later, and v_i to decide in case p_i is the process with the highest rank. Finally, p_i waits for such triples (S_j, v_j, j) from $n-f$ processes, and stores them in J_i. Intuitively, J_i has the data (S_j, v_j) of processes whose rank can be retrieved: they did not crash too early in the protocol. In the second part of the algorithm, p_i calls binding gather with its set J_i, and obtains a bunch of such sets from other processes, and puts all the triples (S_j, v_j, j) obtained in a big set *bigJ*. At this point, the common set C of binding gather has been determined, by definition of binding gather, and with probability at least $(n-f)/n - 1/n \geq 1/3$ (assuming $f \geq 1$), C includes the process with largest rank and such a process is unique. Process p_i then unravels *bigJ* to extract a set P of processes and for each $j \in P$, extracts their values of S_j and v_j. In the third part, processes recover the random numbers and add them together to produce the rank of each process in P. A process may not be able to recover r_{kj} (i.e., return from *recover*^{kj}) since process p_k may have crashed before returning from *dealer-share*^{kj}. However, this does not happen for $j \in P$ and $k \in S_j$: for those values of j and k, p_j sent a *done* message and completed *share*$_{kj}$ by definition of P and S_j. Therefore, for all $j \in P$, p_i can retrieve the rank of p_j by adding together the appropriate r_{*j}'s mod n^4. This is stored in variable *rank*$_j$. Finally, in part 4, process p_i picks the process in P with largest rank and decides on the value of that process.

5.2 Fast Consensus

Our consensus algorithm is obtained by modifying Ben-Or's algorithm [3], which is a *binary* consensus algorithm in which the processes' initial values must be 0 or 1. The key idea of Ben-Or's protocol is that, if all processes start a round with the same estimate, then they all decide in that round. If there is no decision, at the end of the round some processes will set their estimate to a random bit in the hope that, if they are very lucky, processes will all end up with the same bit and therefore will decide in the next round. If all n processes pick a bit randomly, the probability that they will pick the same bit is exponentially small in n. As a result, the expected number of rounds until decision is exponentially large in n.

We modify Ben-Or's algorithm so that, at the end of each round, instead of using a random coin, processes use an instance of probabilistic agreement to set

Code for each process p_i with initial value v_i:

1. $k := 0$
2. while true do
3. $k := k + 1$
   (phase 1 *)*
4. send (REPORT, k, v_i) to all
5. wait to receive $(\text{REPORT}, k, *)$ from $n-f$ processes
   (phase 2 *)*
6. if all received (REPORT, k, w) are for the same w
7. then send $(\text{PROPOSAL}, k, w)$ to all
8. else send $(\text{PROPOSAL}, k, ?)$ to all
9. wait to receive $(\text{PROPOSAL}, k, *)$ from $n-f$ processes
10. if received some $(\text{PROPOSAL}, k, w)$ with $w \neq ?$ then $v_i := w$
11. if all received $(\text{PROPOSAL}, k, w)$ are for the same w then decide w
    (phase 3 *)*
12. $v_i :=$ *probabilistic-agreement*(k, v_i) (* run k-th instance of probabilistic agreement with input v_i *)

Fig. 3. Algorithm for consensus and $n \geq 2f+1$, which uses probabilistic agreement as a subroutine. By using a fast probabilistic agreement algorithm, we obtain a fast consensus algorithm.

their estimate (there is an instance of probabilistic agreement per round). The rationale is that probabilistic agreement has a high (constant) probability that processes will pick the same value and hence decide in the next round. As a result, the expected number of rounds until decision is constant. There is another more subtle difference between our algorithm and Ben-Or's. To ensure agreement, in Ben-Or's algorithm a process does *not* change its estimate to a random value at the end of a round if it believes another process may have decided. In our algorithm, all processes unconditionally change their estimate to the decision value of probabilistic agreement. Doing so does not jeopardize agreement because if a process decides v in a round, all processes will start probabilistic agreement with v and hence will decide v (in probabilistic agreement). Another difference between the algorithms is that our algorithm is not restricted to binary consensus: initial values can come from any domain.

Our complete algorithm is shown in Figure 3. Each process maintains an estimate of the decision in variable v_i, which is initially the process initial value. Processes proceed in rounds $k = 1, 2, \ldots$, where each round has three phases. In phase 1 of round k, a process sends a (REPORT, k, v_i) message with its estimate v_i to all, waits to receive $n-f$ reports of round k, and checks whether a majority of processes reported the same estimate w. If so, in phase 2, a process sends a $(\text{PROPOSE}, k, w)$ message to all, otherwise it sends a $(\text{PROPOSE}, k, ?)$, where "?" is a special value. There can be either 0 or 1 proposals different from ? in phase 2, because this proposal must have been reported in phase 1 by a majority of processes. Processes wait for $n-f$ proposals of round k. If all of them are for the same value $w \neq ?$ then the process decides on w. If one of them is for a value $w \neq ?$ then the process changes its estimate v_i to w. In phase 3, processes executes a new instance of probabilistic agreement using v_i as its initial value, and then changes v_i to the decision.

This algorithm can be easily extended to handle omission failures: we use the PKI to implement a private reliable send mechanism. Briefly, for p to send a message m to q privately and reliably, it performs the following. p encrypts and sends (m, q) to all processes and waits to receive acknowledgements from $f+1$ processes; a process that receives (m, q) from p sends an acknowledgement to p and forwards m to q.

6 Algorithms for Byzantine Failures Using Weak Sequence Broadcast

We now consider an asynchronous system with Byzantine failures, where $n \geq 2f+1$. In this setting, it is easy to show that consensus (with probability one termination) cannot be solved even if processes have access to a public-key infrastructure. We therefore consider solutions that use weak sequenced broadcast as a primitive, described in Section 6.1. We show that this primitive can be implemented with a minimal TPM. Because the minimal TPM can be implemented in a system with crash failures, it follows that deterministic consensus is still impossible, even if processes have weak sequenced broadcast (otherwise, processes could solve consensus in a system with crash failures). Thus, as before, we have to resort to randomized solutions that guarantee termination with probability one minus a negligible probability due to the use of cryptography. Using sequenced broadcast, we show how to implement probabilistic agreement with Byzantine failures, and then how to implement weak Byzantine agreement.

6.1 Sequenced Broadcast and Weak Sequenced Broadcast

Roughly speaking, sequenced broadcast is a type of broadcast that ensures, for a given sender p_i, that all processes deliver the messages of that sender in the same order. This provides an ordering *per sender* of messages, similar to FIFO broadcast [23].[2] This is useful because it prevents the problem of equivocation [12], in which a Byzantine process can send different values to different processes. In a system with $n \geq 3f+1$, equivocation can be avoided using Bracha's broadcast algorithm [5], but here we are concerned about systems with $n \geq 2f+1$.

We shall consider two versions of sequenced broadcast, where the stronger version ensures that all correct processes deliver the same set of messages, and the weaker version does not. More precisely, we define weak sequenced broadcast in terms of two primitives, *sbcast* and *sdeliver*. We are interested in the k-th message broadcast by a process, and the k-th message delivered from p by another process. To make this explicit, when a process p broadcasts m as its k-th message, we will say that p *sbcasts*(k, m). We note that k is determined by the order in which the process calls *sbcast*, and so it is not a real parameter; we make k explicit just to make it simpler to match a broadcast with its deliver in algorithms. When a process q delivers m as its k-th message from p, we will say that q *sdelivers*(k, m)

[2] FIFO broadcast is defined for systems with crash failures. Sequenced broadcast can be seen as an extension of FIFO broadcast to systems with Byzantine failures.

from p. We assume that messages broadcast by correct processes are unique, which can be ensured via sequence numbers. Weak sequenced broadcast satisfies the following properties:

- *Integrity.* If processes p and q are correct, and q *sdelivers* (k, m) from p then p previously *sbcasts*(k, m).
- *Validity.* If correct process p *sbcasts*(k, m) then eventually all correct processes *sdeliver*(k, m) from p.
- *FIFO Agreement:* If two correct processes *sdeliver* (k, m) and (k, m') from the same process q then $m = m'$.

Weak sequenced broadcast allows one correct process to deliver a message from a sender, and another correct process not to. This is not allowed in *sequenced broadcast*, which provides an additional Agreement property similar to reliable broadcast [23]:

- *Agreement.* If a correct process *sdelivers* (k, m) then eventually all correct processes *sdeliver* (k, m).

It is easy to implement sequenced broadcast using weak sequenced broadcast. We now explain how to implement weak sequenced broadcast using a minimal TPM. The TPM has a secret TPM signing key and provides each process p_i with a private set of registers PCR_k (Platform Configuration Registers) initialized to zero [31]. The TPM allows a process to modify a PCR register only by using the *tpm-extend*(k, v) function, which sets $PCR_k := hash(PCR_k \cdot v)$. Function *tpm-quote*$(k)$ allows p_i to sign (i, PCR_k) using the TPM secret key. There are many PCR registers, but we only use PCR_1.

To *sbcast*(m), process p_i calls *tpm-extend*$(1, m)$, and then calls *tpm-quote*(1) to obtain a TPM signature s on (i, PCR_1). Next, p_i sends m and s to all using FIFO-send (FIFO-send can be implemented using sequence numbers). Each process p_j keeps a vector $H_j[i]$ with the hash of the sequence of messages that p_j has seen from p_i for every i. When p_j receives (m, s) from p_i, it updates $H_j[i]$ to mimic the way the TPM of p_i updates PCR_1. Then p_j checks whether s is a valid signature on $(i, H_j[i])$. If so, p_j *sdelivers* m, otherwise it ignores m.

The key reason why the algorithm works is that a process cannot set its PCR_1 any way it wants, because *hash* is a one-way hash function. Once a process extends PCR_1 for k-th time using message m, it is restricted to send m as its k-th messages to all processes, otherwise m is rejected by correct processes.

6.2 Verifiable Secret Sharing

Our probabilistic agreement algorithm relies on VSS. We now give a simple VSS implementation for a system with Byzantine failures and $n \geq 2f+1$ using sequenced broadcast, and the cryptographic primitives of encryption, signature, and zero-knowledge proofs of knowledge. Roughly speaking, it works as follows. The dealer chooses a degree f polynomial $g(x)$ and sends via sequenced broadcast a message $\langle E_1(g(1)), \cdots, E_n(g(n))\rangle$ where $E_i(g(i))$ is an encryption of the share of player i using player i's public encryption scheme. The dealer then provides a zero-knowledge proof that the message sent is indeed an appropriate encryption

of a degree f polynomial. A player that is convinced by the proof sends an acknowledgement via sequenced broadcast and the share phase ends once $n-f$ acknowledgements are received. The algorithm is given in Figure 4.

```
Code for process p_i:
1.   procedure dealer-share(v_i)                                   { for the dealer process }
2.     choose a random polynomial g of degree f such that g(0) = v_i
3.     sbcast(1, ⟨E_1(g(1)), ··· , E_n(g(n))⟩)        E_j(·) encrypts using p_j's public key *)
4.     do an interactive constant-round zero-knowledge proof with each process on the statement that
         E_1(g(1)), ..., E_n(g(n)) is an encryption of a degree f polynomial
5.   procedure share()                                             { for all processes }
6.     wait to sdeliver(1, ⟨EG_1, ..., EG_n⟩) from dealer   (* if dealer is correct EG_j = E_j(g(j)) *)
7.     fork {
8.       participate in constant-round zero-knowledge proof with dealer that e_1, ..., e_n is an encryption of
           a degree f polynomial
9.       if process is convinced of zero-knowledge proof
10.      then sbcast(2, DONE)        (* if necessary sbcast an empty messages so we can sbcast(2, ...) *)
11.    }
12.    wait until sdeliver(2, DONE) from n−f processes
13.  procedure recover():                                          { for all processes }
14.    send (i, g(i)) to all
15.    wait to receive (j, g_j) from n−f processes where E_j(g_j) = EG_j   (* g_j must match encrypted values
                                                                          broadcast by dealer *)
16.    find degree polynomial g of degree f going through the n−f values (j, g_j) gotten in line 15
17.    return g(0)
```

Fig. 4. Algorithm for securely implementing verifiable secret sharing and $n \geq 2f+1$ using sequenced broadcast in a system with Byzantine failures.

6.3 Binding Gather with Certified Values

We now extend the definition of binding gather for a system with Byzantine failures. Clearly, we cannot require that Byzantine processes do anything, so we modify its properties as follows:

- *Validity.* Every value in every output set of a *correct* process is the input value of some process.
- *Binding Commonality.* The exists some value v such that v is in the output set of every *correct* process, and v can be determined by an external observer when the first *correct* process outputs its set.
- *Termination.* All correct processes eventually output some value.

(The italics indicate differences with respect to the definition in Section 4.) Even with these weaker requirements, there is still a problem with the Validity property: no implementation can provide this property, because if a Byzantine process acts correctly except that it changes its initial value to a bogus value, then this bogus value could appear in the output of correct processes.

To address this problem, we use the notion of *certified values*, which is similar to external validity [7]. Roughly speaking, a value v is certified if there is a legitimacy test that can be applied to v such that bogus values from Byzantine process cannot pass the test. For example, we may require a value to have signatures from $f+1$ processes, and the test verifies that the signatures are valid.

More precisely, we define a *certification scheme* as a set *Cert* of values and a set of Boolean procedures $check_i(v)$, one for each process p_i, such that a byzantine process cannot generate a value in *Cert* unless it is given that value (e.g., it receives the value from a correct process). Intuitively, values in *Cert* are the certified values and each $check_i(v)$ is a way for process p_i to check if $v \in Cert$. We require two properties: (a) if $v \notin Cert$ then $check_i(v)$ must return *false*; and (b) if $v \in Cert$ then there is a time after which $check_i(v)$ returns *true* (it may return *false* for a finite period until p_i sees evidence from other processes that $v \in Cert$). Since we rely on cryptographic primitives, we allow a negligible probability that properties (a) and (b) are violated.

By using certification schemes, we can provide a stronger validity condition. For example, with Byzantine agreement, we can require that initial values be certified, where the certification scheme is a parameter of the problem. Thus, each process has an unforgeable initial value and we can design algorithms that use the $check_i$ procedures to ignore bad values and ensure that correct processes decide on one of the initial values.

Similarly, we define *binding gather with certified values* via the three properties of binding gather with the requirement that initial values must be certified. Certification schemes are particularly useful in the composition of algorithms, when the input of an algorithm is an unforgeable output of another algorithm, as we demonstrate with probabilistic agreement and weak Byzantine agreement.

It is easy to implement binding gather with certified values, by using the same algorithm of Section 4 except that processes use sequenced broadcast to send values, and they ignore values that do not pass the certification check.

6.4 Fast Probabilistic Agreement with Certified Values

We now describe an algorithm for probabilistic agreement. To satisfy the Validity property of probabilistic agreement, we need to assume that initial values are certified according to some certification scheme, as in Section 6.3. We denote the certification test procedures by $checkInput_i$.

The algorithm for probabilistic agreement with Byzantine failures is similar to the one of Section 5.1, except that we replace send-to-all with the sequenced broadcast primitive (Section 6.1), we use the verifiable secret sharing algorithm for Byzantine failures (Section 6.2), and we use binding gather with certified values (Section 6.3). Furthermore, the algorithm accepts a message only if it is delivered by sequenced broadcast, and the value it carries has passed the secret sharing phase and the certification test of $checkInput_i$.

6.5 Fast Weak Byzantine Agreement

The algorithm for weak Byzantine agreement is similar to the algorithm for consensus and crash failures of Section 5.2. The differences are that we use the probabilistic agreement algorithm with certified values of Section 6.4, using a certification check that we will explain below. In addition, we replace send-to-all with sequenced broadcast. We must also check that messages received from

processes follow the protocol. To do so, when a process *sbcasts* a message m, it must attach a proof that m follows the algorithm. This proof consists of the messages that the process *sdelivered* that causes it to send m, where those messages themselves must carry proofs that they are legitimate. Thus, in the end, each message m will contain a history of the execution that justifies m. When a process waits to receive $n-f$ messages, it ignores messages with incorrect proofs. At the end of each round, a process calls probabilistic agreement with certified values using the input $\langle v_i, P_i \rangle$, where P_i is a proof that v_i is legitimate. The certification test $checkInput_i$ used in probabilistic agreement is the function that verifies that v_i is legitimate according to P_i.

Acknowledgments

We would like to thank Sergey Yekhanin for helpful discussions and insights on the binding gather protocol.

References

1. Abraham, I., Dolev, D., Halpern, J.Y.: An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In: ACM Symposium on Principles of Distributed Computing, pp. 405–414. ACM, New York (2008)
2. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd edn. John Wiley Interscience, Chichester (March 2004)
3. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: ACM Symposium on Principles of Distributed Computing, pp. 27–30. ACM, New York (1983)
4. Berman, P., Garay, J.A.: Randomized distributed agreement revisited. In: FTCS, pp. 412–419 (1993)
5. Bracha, G.: An asynchronous [(n - 1)/3]-resilient consensus protocol. In: ACM Symposium on Principles of Distributed Computing, pp. 154–162. ACM, New York (1984)
6. Cachin, C., Kursawe, K., Lysyanskaya, A., Strobl, R.: Asynchronous verifiable secret sharing and proactive cryptosystems. In: CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 88–97. ACM, New York (2002)
7. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 524–541. Springer, Heidelberg (2001)
8. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In: ACM Symposium on Principles of Distributed Computing, pp. 123–132. ACM, New York (2000)
9. Canetti, R., Rabin, T.: Fast asynchronous byzantine agreement with optimal resilience. In: ACM Symposium on Theory of Computing, pp. 42–51. ACM, New York (1993)
10. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)

11. Chor, B., Merritt, M., Shmoys, D.B.: Simple constant-time consensus protocols in realistic failure models. J. ACM 36(3), 591–614 (1989)
12. Chun, B.-G., Maniatis, P., Shenker, S., Kubiatowicz, J.: Attested append-only memory: making adversaries stick to their word. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 189–204. ACM, New York (2007)
13. Correia, M., Neves, N.F., Lung, L.C., Veríssimo, P.: Low complexity byzantine-resilient consensus. Distrib. Comput. 17(3), 237–249 (2005)
14. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM 34(1), 77–97 (1987)
15. Dolev, D., Strong, H.R.: Polynomial algorithms for multiple processor agreement. In: ACM Symposium on Theory of Computing, pp. 401–407. ACM, New York (1982)
16. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
17. Feldman, P.: Asynchronous byzantine agreement in constant expected time (copy availbale from M. Ben-Or) (1989) (unpublished)
18. Feldman, P., Micali, S.: Optimal algorithms for byzantine agreement. In: ACM Symposium on Theory of Computing, pp. 148–161 (1988)
19. Feldman, P., Micali, S.: An optimal probabilistic protocol for synchronous byzantine agreement. SIAM J. Comput. 26(4), 873–933 (1997)
20. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. Distributed Computing 1(1), 26–39 (1986)
21. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty processor. J. ACM 32(2), 374–382 (1985)
22. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. Journal of Cyptology 3(2), 99–111 (1991)
23. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Computer Science Department, Cornell University, Ithaca, New York (May 1994)
24. Katz, J., Koo, C.-Y.: On expected constant-round protocols for byzantine agreement. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 445–462. Springer, Heidelberg (2006)
25. Kotla, R., Roy, I.: Personal Communication (2010)
26. Lamport, L.: The weak byzantine generals problem. J. ACM 30(3), 668–676 (1983)
27. Levin, D., Douceur, J.R., Lorch, J.R., Moscibroda, T.: Trinc: small trusted hardware for large distributed systems. In: NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation, pp. 1–14. USENIX Association, Berkeley (2009)
28. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers, Inc., San Francisco (1996)
29. Patra, A., Choudhary, A., Rangan, C.P.: Simple and efficient asynchronous byzantine agreement with optimal resilience. In: PODC, pp. 92–101. ACM, New York (2009)
30. Stadler, M.: Publicly verifiable secret sharing. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 190–199. Springer, Heidelberg (1996)
31. http://www.trustedcomputinggroup.org/resources/tpm_main_specification (February 2010)

Transactions as the Foundation of a Memory Consistency Model*

Luke Dalessandro[1], Michael L. Scott[1], and Michael F. Spear[2]

[1] Department of Computer Science, University of Rochester
[2] Department of Computer Science and Engineering, Lehigh University

Abstract. We argue that traditional synchronization objects, such as locks, conditions, and atomic/volatile variables, should be defined in terms of transactions, rather than the other way around. A traditional critical section, in particular, is a region of code, bracketed by transactions, in which certain data have been *privatized*. We base our memory model on the notion of *strict serializability* (SS), and show that selective relaxation of the relationship between program order and transaction order can allow the implementation of transaction-based locks to be as efficient as conventional locks. We also show that condition synchronization can be accommodated without explicit mention of speculation, *opacity*, or aborted transactions. Finally, we compare SS to the notion of *strong isolation* (SI), arguing that SI is neither sufficient for *transactional sequential consistency* (TSC) nor necessary in programs that are *transactional data-race free* (TDRF).

1 Introduction

Transactional Memory (TM) attempts to simplify synchronization by raising the level of abstraction. Drawing inspiration from databases, it allows the programmer to specify that a block of code should execute atomically, without specifying how that atomicity should be achieved. (The typical implementation will be based on speculation and rollback.) In comparison to lock-based synchronization, TM avoids the possibility of deadlock, and—at least to a large extent—frees the programmer from an unhappy choice between the simplicity of coarse-grain locking and the higher potential concurrency of fine-grain locking.

Unfortunately, for a mechanism whose principal purpose is to simplify the programming model, TM has proven surprisingly resistant to formal definition. Difficult questions—all of which we address in this paper—include the following. ▷ Does the programmer need to be aware of speculation and rollback? What happens if a transaction attempts to perform an operation that cannot be rolled back? ▷ What happens when the same data are accessed both within and outside transactions? Does the answer depend on races between transactional and nontransactional code? ▷ Can transactions be added to a program already containing locks—that is, can the two be used together? ▷ How does one express

* This work was supported in part by NSF grants CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 20–34, 2010.

condition synchronization, given that activities of other threads are not supposed to be visible to an already-started transaction?

Answers to these questions require a *memory model*—a set of rules that govern the values that may be returned by reads in a multithreaded program. It is generally agreed that programmers in traditional shared-memory systems expect *sequential consistency*—the appearance of a global total order on memory accesses, consistent with program order in every thread, and with each read returning the value from the most recent write to the same location [17]. We posit that transactional programmers will expect *transactional sequential consistency* (TSC)—SC with the added restriction that accesses of a given transaction be contiguous in the total execution order. However, just as typical multiprocessors and parallel programming languages provide a memory model weaker than SC (due to its implementation cost), typical transactional systems can be expected to provide a model weaker than TSC. How should this model be defined?

Several possibilities have been suggested, including *strong isolation* (SI) (a.k.a. strong atomicity) [4,28], *single lock atomicity* (SLA) [14, 1st edn., p. 20] [23], and approaches based on ordering-based memory models [10], linearizability [11,26], and operational semantics [1,24]. Of these, SLA has received the most attention. It specifies that transactions behave as if they acquired a single global mutual exclusion lock.

Several factors, however, make SLA problematic. First, it requires an underlying memory model to explain the behavior of the equivalent lock-based program. Second, it leads to arguably inappropriate semantics for programs that have transactional-nontransactional data races, that mix transactions with fine-grain locks, or that contain infinite loops in transactions. Third—and perhaps most compelling—it defines transactions in terms of the mechanism whose complexity we were supposedly attempting to escape.

We have argued [30] that ordering-based memory models such as those of Java [20] and C++ [5] provide a more attractive foundation than locks for TM. Similar arguments have been made by others, including Grossman et al. [10], Moore and Grossman [24], Luchangco [18], Abadi et al. [1], and Harris [12]. Our model is based on the *strict serializability* (SS) of database transactions. We review it in Section 2.

In Section 3 we show how locks and other traditional synchronization mechanisms can be defined *in terms of* transactions, rather than the other way around. By making atomicity the fundamental unifying concept, SS provides easily understood (and, we believe, intuitively appealing) semantics for programs that use a mix of synchronization techniques. In Section 4 we note that *selective strict serializability* (SSS), also from our previous work, can eliminate the need for "unnecessary" fences in the implementation of lock and volatile operations, allowing those operations to have the same code—and the same cost—as in traditional systems, while still maintaining a global total order on transactions. In section 5 we show how to augment SS or SSS with condition synchronization (specifically, the retry primitive of Harris et al. [15]) without explicit mention of speculation or aborted transactions. Finally, in Section 6, we compare SS to the

notion of *strong isolation* (SI), arguing that SI is both insufficient to guarantee TSC for arbitrary programs, and unnecessary in programs that are TDRF. We conclude in Section 7.

2 The Basic Transactional Model

As is customary [9], we define a *program execution* to be a set of *thread histories*, each of which comprises a totally ordered sequence of *reads*, *writes*, and other *operations*—notably *external actions* like input and output. The history of a given thread is determined by the program text, the semantics (not specified here) of the language in which that text is written, the input provided at run time, and the values returned by reads (which may be set by other threads). An execution is said to be *sequentially consistent* (SC) if there exists a total order on reads and writes, across all threads, consistent with program order in each thread, such that each read returns the value written by the most recent preceding write to the same location.

An *implementation* maps source programs to sets of *target executions* consisting of instructions on some real or virtual machine. The implementation is correct only if, for every target execution, there exists an *equivalent* program execution—one that performs the same external actions, in the same order.

Given the cost of sequential consistency on target systems, *relaxed* consistency models differentiate between *synchronization* operations and *ordinary* memory accesses (reads and writes). Operations within a thread are totally ordered by *program order* $<_p$. Synchronization operations across threads are partially ordered by *synchronization order* $<_s$, which must be consistent with program order. The irreflexive transitive closure of $<_p$ and $<_s$, known as *happens-before order* ($<_{hb}$), provides a global partial order on operations across threads.

Two ordinary memory accesses, performed by different threads, are said to *conflict* if they access the same location and at least one of them is a write. An execution is said to have a *data race* if it contains a pair of conflicting accesses that are not ordered by $<_{hb}$. A program is said to be *data-race free* (DRF) with respect to $<_s$ if none of its sequentially consistent executions has a data race.

Relaxed consistency models differ in their choice of $<_s$ and in their handling of data races. In all models, a read is permitted to return the value written by the most recent write to the same location along some happens-before path. If the program is data-race free, any topological sort of $<_{hb}$ will constitute a sequentially consistent execution.

For programs with data races, arguably the simplest strategy is to make the behavior of the entire program undefined. Boehm et al. [5] argue that any attempt to define stronger semantics for C++ would impose unacceptable implementation costs. For managed languages, an at-least-superficially attractive approach is to allow a read to return either (1) the value written by the most recent write to the same location along some happens-before path or (2) the value written by a racing write to that location (one not ordered with the read under $<_{hb}$). As it turns out, this strategy is insufficiently strong to preclude circular

reasoning. To avoid "out of thin air" reads, and ensure the integrity of the virtual machine, the Java memory model imposes an additional *causality* requirement, under which reads must be incrementally explained by already-justified writes in shorter executions [20].

2.1 Transactional Sequential Consistency

Our transactional memory model builds on the suggestion, first advanced by Grossman et al. [10] and subsequently adopted by others [1,24,30], that $<_s$ be defined in terms of transactions. We extend thread histories to include begin_txn and end_txn operations, which we require to appear in properly nested pairs. We use the term "transaction" to refer to the contiguous sequence of operations in a thread history beginning with an outermost begin_txn and ending with the matching end_txn.

Given experience with conventional parallel programs, we expect that (1) races in transactional programs will generally constitute bugs, and (2) the authors of transactional programs will want executions of their (data-race-free) programs to appear sequentially consistent, with the added provision that transactions occur atomically. This suggests the following definition:

A program execution is *transactionally sequentially consistent* (TSC) iff there exists a global total order on operations, consistent with program order in each thread, that explains the execution's reads (in the sense that each read returns the value written by the most recent write to the same location), and in which the operations of any given transaction are contiguous. An *implementation* (system) is TSC iff for every realizable target execution there exists an equivalent TSC program execution.

Similar ideas have appeared in several previous studies. TSC is equivalent to the *strong semantics* of Abadi et al. [1], the *StrongBasic* semantics of Moore and Grossman [24], and the transactional memory with *store atomicity* described by Maessen and Arvind [19]. TSC is also equivalent to what Larus and Rajwar called *strong isolation* [14, 1st edn., p. 27], but stronger than the usual meaning of that term, which does not require a global order among nontransactional accesses [4,6].

2.2 Strict Serializability

In the database world, the standard ordering criterion is *serializability*, which requires that the result of executing a set of transactions be equivalent to 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* (SS) imposes the additional requirement that if transaction A completes before B starts (in the underlying implementation), then A must occur before B in the equivalent serial execution. The intent of this definition is that if external (non-database) operations allow one to tell that A precedes B, then A must serialize before B. We adopt strict serializability as the synchronization order for transactional memory, equating non-database operations with nontransactional

memory accesses, and insisting that such accesses occur between the transactions of their respective threads, in program order. In our formulation:

Program order, $<_p$, is a union of per-thread total orders, and is specified explicitly as part of the execution. In a legal execution, the operations performed by a given thread are precisely those specified by the sequential semantics of the language in which the source program is written, given the values returned by the execution's input operations and reads. Because (outermost) transactions are contiguous in program order, $<_p$ also orders transactions of a given thread with respect to one another and to the thread's nontransactional operations.

Transaction order, $<_t$, is a total order on transactions, across all threads. It is consistent with $<_p$, but is not explicitly specified. 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 induced by $<_p$ and $<_t$. Specifically, it is a superset of $<_t$ that also orders nontransactional accesses with respect to preceding and following transactions of the same thread. Formally, for all accesses a and c in a program execution, we say $a <_{ss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists$ a transaction A such that $(a \in A \ \wedge \ A <_p c)$; (3) $\exists$ a transaction C such that $(a <_p C \ \wedge \ c \in C)$; (4) $\exists$ an access b such that $a <_{ss} b <_{ss} c$.

We say a memory access b *intervenes* between a and c iff $a <_p b \vee a <_{ss} b$ and $b <_p c \vee b <_{ss} c$. Read r is then permitted to return the value written by write w if r and w access the same location l, $w <_p r \vee w <_{ss} r$, and there is no intervening write of l between w and r. Depending on the choice of programming language, r may also be permitted to return the value written by w if r and w are incomparable under both $<_p$ and $<_{ss}$. Specifically, in a Java-like language, a read should be permitted to see an incomparable but causally justifiable write.

An execution with program order $<_p$ is said to be *strictly serializable* (SS) if there exists a transaction order $<_t$ that together with $<_p$ induces a strict serial order $<_{ss}$ that (together with $<_p$) permits all the values returned by reads in the execution. A TM implementation is said to be SS iff for every realizable target execution there exists an equivalent SS program execution.

In a departure from nontransactional models, we do not include all of program order in the global $<_{ss}$ order. By adopting a more minimal connection between program order and transaction order, we gain the opportunity (in Section 4) to relax this connection as an alternative to relaxing the transaction order itself.

2.3 Transactional Data-Race Freedom

As in traditional models, two ordinary memory accesses are said to *conflict* if they are performed by different threads, they access the same location, and at least one of them is a write. A legal execution is said to have a *data race* if it contains, for every possible $<_t$, a pair of conflicting accesses that are not ordered by the resulting $<_{ss}$. A program is said to be *transactional data-race free* (TDRF) if none of its TSC executions has a data race. It is easy to show [7] that any execution of a TDRF program on an SS implementation will be TSC.

```
class lock
  Boolean held := false
  void acquire( )
    while true
      atomic
        if not held
          held := true
          return
  void release( )
    atomic
      held := false
```

```
class condition(lock L)
  class tok
    Boolean ready := false
  queue⟨tok⟩ waiting := [ ]
  void wait( )
    t := new tok
    waiting.enqueue(t)
    while true
      L.release( )
      L.acquire( )
      if t.ready return
```

```
void signal( )
  t := waiting.dequeue( )
  if t != null
    t.ready := true
void signal_all( )
  while true
    t := waiting.dequeue( )
    if t = null return
    t.ready := true
```

Fig. 1. Reference implementations for locks and condition variables. Lock L is assumed to be held when calling condition methods

3 Modeling Locks and Other Traditional Synchronization

One often sees attempts to define transactions in terms of locks. Given a memory consistency model based on transactions, however, we can easily define locks in terms of transactions. This avoids any objections to defining transactions in terms of the thing they're intended to supplant. It's also arguably simpler, since we need a memory model anyway to define the semantics of locks.

Our approach stems from two observations. First, any practical implementation of locks requires some underlying atomic primitive(s) (e.g., test-and-set or compare-and-swap). We can use transactions to model these, and then define locks in terms of a reference implementation. Second, a stream of recent TM papers has addressed the issue of *publication* [23] and *privatization* [14,21,34], in which a program uses transactions to transition data back and forth between logically shared and private states, and then uses nontransactional accesses for data that are private. We observe that privatization amounts to locking.

3.1 Reference Implementations

Fig. 1 shows reference implementations for locks and condition variables. Similar implementations can easily be written for volatile (atomic) variables, monitors, semaphores, conditional critical regions, etc. Note that this is *not* necessarily how synchronization mechanisms would be implemented by a high-quality language system. Presumably the compiler would recognize calls to acquire, release, etc., and generate semantically equivalent but faster target code.

By defining traditional synchronization in terms of transactions, we obtain easy answers to all the obvious questions about how the two interact. Suppose, for example, that a transaction attempts to acquire a lock (perhaps new transactional code calls a library containing locks). If there is an execution prefix in which the lock is free at the start of the transaction, then acquire will perform a single read and write, and (barring other difficulties) the transaction can occur. If there is no execution prefix in which the lock is free at the start of the

Initially v = w = 0

Thread 1	Thread 2
1: atomic	
2: v := 1	
	3: while (v ≠ 1) { }
	4: w := 1
5: while (w ≠ 1) { }	

Fig. 2. Reproduced from Figure 2 of Shpeisman et al. [27]. If transactions have the semantics of lock-based critical sections, then this program, though racy, should terminate successfully.

transaction, then (since transactions appear in executions in their entirety, or not at all) there is no complete (terminating) execution for the program. If there are some execution prefixes in which the lock is available and others in which it is not, then the one(s) in which it is available can be extended (barring other difficulties) to create a complete execution. (Note that executions enforce safety, not liveness—more on this in Section 5.) The reverse case—where a lock-based critical section contains a transaction—is even easier: since acquire and release are themselves separate transactions, no actual nesting occurs.

Note that in the absence of nesting, only acquire and release—not the bodies of critical sections themselves—are executed as transactions; critical sections protected by different locks can therefore run concurrently. Interaction between threads can occur within lock-based critical sections but not within transactions.

3.2 Advantages with Respect to Lock-Based Semantics

Several researchers, including Harris and Fraser [13] and Menon et al. [22,23], have suggested that lock operations (and similarly volatile variable accesses) be treated as tiny transactions. Their intent, however, was not to merge all synchronization mechanisms into a single formal framework, but simply to induce an ordering between legacy mechanisms and any larger transactions that access the same locks or volatiles. Harris and Fraser suggest that it should be possible (as future work) to develop a unified formal model reminiscent of the Java memory model. The recent draft TM proposal for C++ includes transactions in the language's synchronizes-with and happens-before orders, but as an otherwise separate mechanism; nesting of lock-based critical sections within transactions is explicitly prohibited [3].

Menon et al., by contrast, define transactions explicitly in terms of locks. Unfortunately, as noted in a later paper from the same research group (Shpeisman et al. [27]), this definition requires transactions to mimic certain unintuitive (and definitely non-atomic) behaviors of lock-based critical sections in programs with data races. One example appears in Fig. 2; others can be found in Luchangco's argument against lock-based semantics for TM [18]. By making transactions fundamental, we avoid any pressure to mimic the problems of locks. In Fig. 2, for example, we can be sure there is no terminating execution. If, however, we were to replace Thread 1's transaction with a lock-based critical section (L.acquire( ); v = 1; while (w != 1) { }; L.release( );), the program could terminate successfully.

3.3 Practical Concerns

Volos et al. [33] describe several "pathologies" in the potential interaction of transactions and locks. Their discussion is primarily concerned with implementation-level issues on a system with hardware TM, but some of these issues apply to STM systems as well. If traditional synchronization mechanisms are implemented literally *as transactions*, then our semantics will directly obtain, and locks will interact in a clear and well-defined manner with other transactions. If locks are implemented in some special, optimized fashion, then the implementation will need to ensure that all possible usage cases obey the memory model. Volos et al. describe an implementation that can be adapted for use with STM systems based on ownership records. In our NOrec system [8], minor modifications to the acquire operation would allow conventional locks to interact correctly with unmodified transactions.

4 Improving Performance with Selective Strictness

A program that accesses shared data only within transactions is clearly data-race free, and will experience TSC on any TM system that guarantees that reads see values consistent with some $<_t$. A program P that sometimes accesses shared data outside transactions, but that is nonetheless TDRF, will experience TSC on any TM system S that similarly enforces some $<_{ss}$. 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 *privatization* and *publication safe* with respect to SS.

Unfortunately, many existing TM implementations are not publication and privatization safe, and modifying them to be so imposes nontrivial costs [21]. In their paper on lock-based semantics for TM, Menon et al. note that these costs are particularly egregious under single lock atomicity (SLA), which forces every transaction to be ordered with respect to every other [23]. Their weaker models (DLA, ALA, ELA) aim to reduce the cost of ordering (and in particular publication safety) by neglecting to enforce it in questionable cases (e.g., for empty transactions, transactions with disjoint access sets, or transactions that share only an anti-dependence).

We can define each of these weaker models in our ordering-based framework, but the set of executions for a program becomes much more difficult to define, and program behavior becomes much more difficult to reason about. As noted by Harris [14, Chap. 3] and by Shpeisman et al. [27], orderings become dependent on the precise set of variables accessed by a transaction—a set that may depend not only on program input and control flow, but also on optimizations (e.g., dead code elimination) performed by the compiler.

Rather than abandon the global total order on transactions, we have proposed [30] an optional relaxation of the ordering between nontransactional accesses and transactions. Specifically, we allow a transaction to be labeled as *acquiring* (privatizing), *releasing* (publishing), both, or neither.

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, we say $a <_{sss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists$ an acquiring transaction A such that $(a \in A \ \wedge \ A <_p c)$; (3) $\exists$ a releasing transaction C such that $(a <_p C \ \wedge \ c \in C)$; (4) $\exists$ an access b such that $a <_{sss} b <_{sss} c$.

Note that for any given program $<_{sss}$ will be a subset of $<_{ss}$—typically a proper one—and so a program that is TDRF with respect to SS will not necessarily be TDRF with respect to SSS. This is analogous to the situation in traditional ordering-based memory models, where, for example, a program may be DRF1 but not DRF0 [9].

A transactional programming language will probably want to specify that transactions are both acquiring and releasing by default. A programmer who knows that a transaction does not publish or privatize data can then add an annotation that permits the implementation to avoid the cost of publication and privatization safety. Among other things, on hardware with a relaxed memory consistency model, identifying a transaction as (only) privatizing will allow the implementation to avoid an expensive *write-read fence*. The designers of the C++ memory model went to considerable lengths—in particular, changing the meaning of trylock operations—to avoid the need for such fences before acquiring a lock [5]. Given SSS consistency in Fig. 1, we would define the transaction in lock.acquire to be (only) acquiring, and the transaction in lock.release to be (only) releasing. Similarly, a get (read) operation on a volatile variable would be acquiring, and a put (write) operation would be releasing.

5 Condition Synchronization and Forward Progress

For programs that require not only atomicity, but also condition synchronization, traditional condition variables will not suffice: since transactions are atomic, they cannot be expected to see a condition change due to action in another thread. One could release atomicity, effectively splitting a transaction in half (as in the *punctuated* transactions of Smaragdakis et al. [29]), but this would break composability, and require the programmer to restore any global invariants before waiting on a condition. One could also limit conditions to the beginning of the transaction [13], but this does not compose.

Among various other alternatives, the most popular appears to be the retry primitive of Harris et al. [15]. The construct "if (! desired_condition) retry" instructs a speculation-based implementation of TM to roll the current thread back to the beginning of its current transaction, and then deschedule it until something in the transaction's read set has been written by another thread. While the name "retry" clearly has speculative connotations, it can also be interpreted (as Harris et al. do in their operational semantics) as controlling the conditions under which the surrounding transaction is able to perform its one

and only execution. We therefore define retry, for our ordering-based semantics, to be equivalent to while (true) { }.

At first glance, this definition might seem to allow undesirable executions. If T_1 says atomic {f := 1} and T_2 says atomic {if (f != 1) retry}, we would not want to admit an execution in which T_2 "goes first" and waits forever. But there is no such execution! Since transactions appear in executions in their entirety or not at all, T_2's transaction can appear only if T_1's transaction has already appeared. The programmer may think of retry in terms of prescience (execute this only when it can run to completion) or in terms of, well, re-trying; the semantics just determine whether a viable execution exists. It is possible, of course, that for some programs there will exist execution prefixes[1] such that some thread(s) are unable to make progress in any possible extension; these are precisely the programs that are subject to deadlock (and deadlock is undecidable).

Because our model is built on atomicity, rather than speculation, it does not need to address aborted transactions. An implementation based on speculation is simply required to ensure that such transactions have no visible effects. In particular, there is no need for the *opacity* of Guerraoui and Kapałka [11]; it is acceptable for the implementation of a transaction to see an inconsistent view of memory, so long as the compiler and run-time system "sandbox" its behavior.

5.1 Progress

Clearly an implementation must realize only target executions equivalent to some program execution. Equally clearly, it need not realize target executions equivalent to *every* program execution. Which do we want to require it to realize?

It seems reasonable to insist, for starters, that threads do not stop dead for no reason. Consider some realizable target execution prefix M and an equivalent program execution prefix E. If, for thread T, the next operation in program order following T's subhistory in E is nontransactional, we might insist that the implementation be able to extend M to M^+ in such a way that T makes progress—that is, that M^+ be equivalent to some extension E^+ of E in which T's subhistory is longer.

For transactions, which might contain retry or other loops, appropriate goals are less clear. Without getting into issues of fairness, we cannot insist that a thread T make progress in a given implementation just because there exists a program execution in which it makes progress. Suppose, for example, that flag f is initially 0, and that both T_1 and T_2 have reached a transaction reading if (f < 0) retry; f := 1. Absent other code accessing f, one thread will block indefinitely, and we may not wish to dictate which this should be.

Intuitively, we should like to preclude implementation-induced deadlock. As a possible strategy, consider a realizable target execution prefix M with corresponding program execution prefix E, in which each thread in some nonempty set $\{T_i\}$ has reached a transaction in its program order, but has not yet executed that transaction. If for every extension E^+ of E there exists an extension E^{++}

[1] We assume that even in such a prefix, transactions appear *in toto* or not at all.

of E^+ in which at least one of the T_i makes progress, then the implementation is not permitted to leave all of the T_i blocked indefinitely. That is, there must exist a realizable extension M^+ of M equivalent to some extension E' of E in which the subhistory of one of the T_i is longer.

5.2 Inevitability

If transactions are to run concurrently, even when their mutual independence cannot be statically proven, implementations must in general be based on speculation. This then raises the problem of irreversible operations such as interactive I/O. One option is simply to outlaw these within transactional contexts. This is not an unreasonable approach: locks and privatization can be used to make such operations safe.

If irreversible operations are permitted in transactions, we need a mechanism to designate transactions as *inevitable* (*irrevocable*) [32,35]. This can be a static declaration on the transaction as a whole, or perhaps an executable statement. Either way, irreversibility is simply a hint to the implementation; it has no impact on the memory model, since transactions are already atomic.

In our semantics, an inevitable transaction's execution history is indistinguishable from an execution history in which a thread (1) executes a privatizing transaction that privatizes the whole heap, (2) does all the work nontransactionally, and then (3) executes a publishing transaction. This description formalizes the oft-mentioned lack of composability between retry and inevitability.

5.3 orElse and abort

In the paper introducing retry [15], Harris et al. also proposed an orElse construct that can be used to pursue an alternative code path when a transaction encounters a retry. In effect, orElse allows a computation to notice—and explicitly respond to—the failure of a speculative computation.

Both basic transactions and the retry primitive can be described in terms of atomicity: "this code executes all at once, at a time when it can do so correctly." The orElse primitive, by contrast, "leaks" information—a failure indication—out of a transaction that "doesn't really happen," allowing the program to do something else instead. We have considered including failed transactions explicitly in program executions or, alternatively, imposing liveness-style constraints across sets of executions ("execution E is invalid because transaction T appears in some other, related execution"), but both of these alternatives strike us as distinctly unappealing. In the balance, our preference is to leave orElse out of TM. Its effect can always be achieved (albeit without composability or automatic roll-back) by constructs analogous to those of Section 3.

In a similar vein, consider the oft-proposed abort primitive, which abandons the current transaction (with no effect) and moves on to the next operation in program order. Shpeisman et al. observe that this primitive can lead to logical inconsistencies if its transaction does not contribute to the definition of data-race freedom [27]. In effect, abort, like orElse, can be used to leak information from

an aborted transaction. Shpeisman et al. conclude that aborted transactions must appear explicitly in program executions. We argue instead that aborts be omitted from the definition of TM. Put another way, orElse and abort are speculation primitives, *not* atomicity primitives. If they are to be included in the language, it should be by means of orthogonal syntax and semantics.

6 Strong Isolation

Blundell et al. [4] observed that hardware transactional memory (HTM) designs typically exhibit one of two possible behaviors when confronted with a race between transactional and non-transactional accesses. With *strong isolation* (SI) (a.k.a. *strong atomicity*), transactions are isolated both from other transactions and from concurrent nontransactional accesses; with *weak isolation* (WI), transactions are isolated only from other transactions.

Various papers have opined that STMs instrumented for SI result in more intuitive semantics than WI alternatives [25,28], but this argument has generally been made at the level of TM implementations, not user-level programming models. From the programmer's perspective, we believe that TSC is the reference point for intuitive semantics—and SS and SSS systems provide TSC behavior for programs that are correspondingly TDRF. At the same time, for a language that assigns meaning to racy programs, SS and SSS permit many of the results cited by proponents of SI as unintuitive. This raises the possibility that SI may improve the semantics of TM for racy programs.

It is straightforward to extend any ordering-based transactional memory model to one that provides SI. We do this for SS in the technical report version of this paper [7], and explore the resulting model (SI-SS) for example racy programs. We note that SI-SS is not equivalent to TSC; that is, there are racy programs that still yield non-TSC executions. One could imagine a memory model in which the racy programs that *do* yield TSC executions with SI are considered to be properly synchronized. Such a model would authorize programmers to write more than just TDRF code, but it would be a significantly more complicated model to reason about: one would need to understand which races are bugs and which aren't.

An additional complication of any programmer-centric model based on strong isolation is the need to explain exactly what is meant by a nontransactional access. Consider Fig. 3. Here x is an unsigned long long and is being assigned to nontransactionally. Is this a race under a memory model based on SI? The problem is that Thread 2's assignment to x may not be a single instruction. It is possible (and Java in fact permits) that two 32 bit stores will be used to move the 64 bit value. Furthermore, if the compiler is aware of this fact, it may arrange to execute the stores in arbitrary order. The memory model now must specify the granularity of protection for nontransactional accesses.

While SS and SSS do not require a strongly isolated TM implementation, they do not exclude one either. It may seem odd to consider using a stronger implementation than is strictly necessary, particularly given its cost, but there

Thread 1	Thread 2
1: atomic	
2: r := x	x := 3*ull*

Fig. 3. Is this a correct program under an SI-based memory model?

are reasons why this may make sense. First, SS with happens-before consistency for programs with data races is not trivially compatible with undo-log-based TM implementations [27]. These implementations require SI instrumentation to avoid out-of-thin-air reads due to aborted transactions. Indeed, two of the major undo-log STMs, McRT [2] and Bartok [16], are strongly isolated for just this reason. Second, as observed by Grossman et al. [10], strong isolation enables *sequential reasoning*. Given a strongly isolated TM implementation, all traditional single-thread optimizations are valid in a transactional context, even for a language with safety guarantees like Java. Third, SI hardware can make it trivial to implement volatile/atomic variables.

With these observations in mind, we would not discourage development of strongly isolated HTM. For STM, we note that a redo-log based TM implementation with a hash-table write set permits many of the same compiler optimizations that SI does, and, as shown by Spear et al. [31], can provide performance competitive with undo logs. Ultimately, we conclude that SI is insufficient to guarantee TSC for racy programs, and unnecessary to guarantee it for TDRF programs. It may be useful at the implementation level for certain STMs, and certainly attractive if provided by the hardware "for free," but it is probably not worth adding to an STM system if it adds significant cost.

7 Conclusions

While it is commonplace to speak of transactions as a near-replacement for locks, and to assume that they should have SLA semantics, we believe this perspective both muddies the meaning of locks and seriously undersells transactions. Atomicity is a fundamental concept, and it is *not* achieved by locks, as evidenced by examples like the one in Figure 2. By making atomic blocks the synchronization primitive of an ordering-based memory consistency model, we obtain clear semantics not only for transactions, but for locks and other traditional synchronization mechanisms as well.

In future work, we hope to develop a formal treatment of speculation that is orthogonal to—but compatible with—our semantics for TM. We also hope to unify this treatment with our prior work on implementation-level sequential semantics for TM [26].

It is presumably too late to adopt a transaction-based memory model for Java or C++, given that these languages already have detailed models in which other operations (monitor entry/exit, lock acquire/release, volatile/atomic read/write) serve as synchronization primitives. For other languages, however, we strongly suggest that transactions be seen as fundamental.

References

1. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of Transactional Memory and Automatic Mutual Exclusion. In: SIGPLAN Symp. on Principles of Programming Languages (January 2008)
2. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and Runtime Support for Efficient Software Transactional Memory. In: SIGPLAN Conf. on Programming Language Design and Implementation (June 2006)
3. Adl-Tabatabai, A.R., Shpeisman, T. (eds.): Draft Specification of Transaction Language Constructs for C++. Transactional Memory Specification Drafting Group, Intel, IBM, and Sun, 1.0 edn. (August 2009)
4. Blundell, C., Lewis, E.C., Martin, M.M.K.: Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters 5(2) (November 2006)
5. Boehm, H.J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: SIGPLAN Conf. on Programming Language Design and Implementation (June 2008)
6. Dalessandro, L., Scott, M.L.: Strong Isolation is a Weak Idea. In: 4th SIGPLAN Workshop on Transactional Computing (February 2009)
7. Dalessandro, L., Scott, M.L., Spear, M.F.: Transactions as the Foundation of a Memory Consistency Model. Tech. Rep. TR 959, Dept. of Computer Science, Univ. of Rochester (July 2010)
8. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: SIGPLAN Symp. on Principles and Practice of Parallel Programming (January 2010)
9. Gharachorloo, K., Adve, S.V., Gupta, A., Hennessy, J.L., Hill, M.D.: Programming for Different Memory Consistency Models. Journal of Parallel and Distributed Computing 15, 399–407 (1992)
10. Grossman, D., Manson, J., Pugh, W.: What Do High-Level Memory Models Mean for Transactions? In: SIGPLAN Workshop on Memory Systems Performance and Correctness (October 2006)
11. Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: SIGPLAN Symp. on Principles and Practice of Parallel Programming (February 2008)
12. Harris, T.: Language Constructs for Transactional Memory (invited keynote address). In: SIGPLAN Symp. on Principles of Programming Languages (January 2009)
13. Harris, T., Fraser, K.: Language Support for Lightweight Transactions. In: Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (October 2003)
14. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edn. Morgan & Claypool, San Francisco (2010) (first edition, by Larus and Rajwar only, 2007)
15. Harris, T., Marlow, S., Jones, S.P., Herlihy, M.: Composable Memory Transactions. In: SIGPLAN Symp. on Principles and Practice of Parallel Programming (June 2005)
16. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: SIGPLAN Conf. on Programming Language Design and Implementation (June 2006)
17. Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. on Computers C-28(9), 241–248 (1979)

18. Luchangco, V.: Against Lock-Based Semantics for Transactional Memory (brief announcement). In: ACM Symp. on Parallelism in Algorithms and Architectures (June 2008)
19. Maessen, J.W.: Arvind: Store Atomicity for Transactional Memory. Electronic Notes in Theoretical Computer Science 174(9), 117–137 (2007)
20. Manson, J., Pugh, W., Adve, S.: The Java Memory Model. In: SIGPLAN Symp. on Principles of Programming Languages (January 2005)
21. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: Intl. Conf. on Parallel Processing (September 2008)
22. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R., Saha, B., Welc, A.: Single Global Lock Semantics in a Weakly Atomic STM. In: 3rd SIGPLAN Workshop on Transactional Computing (February 2008)
23. 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: ACM Symp. on Parallelism in Algorithms and Architectures (June 2008)
24. Moore, K.F., Grossman, D.: High-Level Small-Step Operational Semantics for Transactions. In: SIGPLAN Symp. on Principles of Programming Languages (January 2008)
25. Schneider, F.T., Menon, V., Shpeisman, T., Adl-Tabatabai, A.R.: Dynamic Optimization for Efficient Strong Atomicity. In: Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (October 2008)
26. Scott, M.L.: Sequential Specification of Transactional Memory Semantics. In: 1st SIGPLAN Workshop on Transactional Computing (June 2006)
27. Shpeisman, T., Adl-Tabatabai, A.R., Geva, R., Ni, Y., Welc, A.: Towards Transactional Memory Semantics for C++. In: ACM Symp. on Parallelism in Algorithms and Architectures (August 2009)
28. 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 Conf. on Programming Language Design and Implementation (June 2007)
29. Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with Isolation and Cooperation. In: Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (October 2007)
30. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: Intl. Conf. on Principles of Distributed Systems (December 2008)
31. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A Comprehensive Contention Management Strategy for Software Transactional Memory. In: SIGPLAN Symp. on Principles and Practice of Parallel Programming (February 2009)
32. Spear, M.F., Silverman, M., Dalessandro, L., Michael, M.M., Scott, M.L.: Implementing and Exploiting Inevitability in Software Transactional Memory. In: 2008 Intl. Conf. on Parallel Processing (September 2008)
33. Volos, H., Goyal, N., Swift, M.: Pathological Interaction of Locks with Transactional Memory. In: 3rd SIGPLAN Workshop on Transactional Computing (February 2008)
34. 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 (March 2007)
35. Welc, A., Saha, B., Adl-Tabatabai, A.R.: Irrevocable Transactions and Their Applications. In: ACM Symp. on Parallelism in Algorithms and Architectures (June 2008)

The Cost of Privatization*

Hagit Attiya[1,2] and Eshcar Hillel[1]

[1] Department of Computer Science, Technion
[2] Ecole Polytechnique Federale de Lausanne (EPFL)

Abstract. *Software transactional memory* (STM) guarantees that a *transaction*, consisting of a sequence of operations on the memory, appears to be executed atomically. In practice, it is important to be able to run transactions together with *nontransactional legacy* code accessing the same memory locations, by supporting *privatization*. Privatization should be provided without sacrificing the parallelism offered by today's multicore systems and multiprocessors.

This paper proves an inherent cost for supporting privatization, which is linear in the number of privatized items. Specifically, we show that a transaction privatizing k items must have a data set of size at least k, in an STM with invisible reads, which is oblivious to different non-conflicting executions and guarantees progress in such executions. When reads are visible, it is shown that $\Omega(k)$ memory locations must be accessed by a privatizing transaction, where k is the minimum between the number of privatized items and the number of concurrent transactions guaranteed to make progresss, thus capturing the tradeoff between the cost of privatization and the parallelism offered by the STM.

1 Introduction

Software transactional memory (STM) is an attractive paradigm for programming parallel applications for multicore systems. STM aims to simplify the design of parallel systems, as well as improve their performance with respect to sequential code by exploiting the scalability opportunities offered by multicore systems. An STM supports *transactions*, each encapsulating a sequence of operations applied on a set of *data items*; an STM guarantees that if any operation takes place, they all do, and that if they do, they appear to do so atomically, as one indivisible operation.

In practice, some operations cannot, or simply are preferred not to be executed within the context of a transaction. For example, an application may be required to invoke irrevocable operations, e.g., I/O operations, or use library functions that cannot be instrumented to execute within a transaction. *Strong atomicity* [19,22,28] guarantees isolation and consistent ordering of transactions in the presence of non-transactional memory accesses. Supporting strong atomicity is crucial both for interoperability with legacy code and in order to improve performance.

A simple solution is to make each nontransactional operation a (degenerate) transaction, but this means that nontransactional operations incur the overhead associated with a transaction. Although compiler optimizations can reduce this cost in some

* This research is supported in part by the *Israel Science Foundation* (grant number 953/06). The full version of this paper [4] contains additional results, proofs and illustrations.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 35–49, 2010.

situations [27, 3], they do not alleviate it completely. Thus, STMs seek to improve performance by supporting *uninstrumented* nontransactional operations [30, 14], which are executed as is, typically as a single access to the shared memory.

Many recent STMs [24, 31, 21, 23, 20, 11, 8, 13, 9] provide strong atomicity by supporting *privatization* [30, 28], thereby allowing to "isolate" some items making them private to a process; the process can thereafter access them nontransactionally, without interference by other processes. It is commonly assumed that privatizing a set of items simply involves disabling all shared references to those items [20, 31, 9], e.g., by nullifying these references. However, it has been claimed that privatization is a major source of overhead for transactional memories [33], and that supporting uninstrumented nontransactional operations can seriously limit their parallelism [7].

Consider, for example, the linked-list depicted in Figure 1, in which every node points to a root item. Every root item points to some disjoint subgraph, such as a a tree, and is the only path to items in the subgraph. Throughout the paper we consider a *workload* in which one transaction privatizes all root items and their subgraphs, and other transactions read all the nodes of the linked list but write only to one root item that is pointed from the list. In this workload, the hope is that a constant amount of work, e.g., nullifying the head of the linked-list, will suffice for privatizing the whole linked list; afterwards, all items in the rooted trees can be accessed by the privatizing process with simple (uninstrumented) reads and writes to the shared memory.

This paper proves that in many important workloads, including the linked list presented above, the hope to combine efficient privatizing transactions with uninstrumented nontransactional reads, cannot be realized, unless parallelism is compromised. Specifically, the privatizing transaction must incur an inherent cost, linear in the number of data items that are privatized and later accessed with uninstrumented reads.

Our lower bounds do not apply to overly sequential STMs, which achieve efficient privatization by using a single global lock and allowing only one transaction to make progress at each time [24, 8], thereby significantly reducing the throughput. We make a fairly weak progress assumption (Property 1), requiring the STM to allow concurrent progress of nonconflicting transactions: a transaction can abort or block only due to a conflicting pending transaction.

A key factor in many efficient STMs is not having to track the data sets of other transactions, especially if they are not conflicting. We capture this feature by assuming that the STM is *oblivious*, namely, a transaction does not distinguish between nonconflicting transactions (Property 2). A simple example is provided by STMs using a global

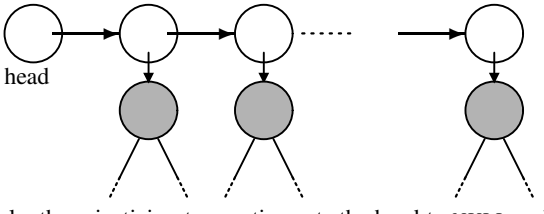


Fig. 1. In this example, the privatizing transaction sets the head to NULL and privatizes the dark items and their subgraphs. Other transactions write to the dark items; each transaction writes to a different one.

clock or counter [26,25,10], or a decentralized clock [5], in which a transaction cannot tell whether a process p executes a transaction that writes to item x or a transaction that writes to item y, unless it accesses either x or y; it can observe that the clock or a counter has increased, but this happens in both cases. A less-immediate example is the behavior of TLRW [11] for so-called *slotted* threads. Several other STMs [23,8,31,13,9] are also oblivious. (A detailed discussion appears in Section 5.)

Our first main result further assumes that reads do not write to the memory (*invisible* reads) and shows that a transaction privatizing k items must have a data set of size $\Omega(k)$ (Theorem 1 in Section 3). In an oblivious STM with invisible reads, transactions are unaware of, and hence, unaffected by, read-read conflicts. In the linked-list example this means that, for every process, the execution of other transactions appears only to write to a single item (either the head of the list or an item pointed by the links).

Our second main result removes the assumption of invisible reads, and shows an $\Omega(k)$ lower bound on the number of shared memory accesses performed by a privatizing transaction, where k is the minimum between the number of privatized items and the level of parallelism, i.e., the number of transactions guaranteed to make progress concurrently (Theorem 2 in Section 4). The proof is more involved and relies on the assumption that the STM provides a significant level of parallelism. This lower bound explains why the *quiescence* mechanism [9, 30, 23], for example, must compromise parallelism in order to support efficient privatization.

Obliviousness generalizes *disjoint-access parallelism* [18], and our lower bounds hold also for disjoint-access parallel STMs (see [4]).

Our proofs only assume a weak safety property that requires a nontransactional read of a data item, where no nontransactional write precedes the read, to return the value written by an earlier committed transaction, or the initial value, if no such transaction commits. This property follows from *parameterized opacity* [14], regardless of the memory model imposed on nontransactional reads and writes.

2 Preliminaries

A *transaction* is a sequence of operations executed by a single process on a set of *data items* shared with other transactions; each item is initialized to some initial value. The collection of data items accessed by a transaction is the transaction's *data set*; in particular, the items written by the transaction are its *write set*, and the items read by the transaction are its *read set*.

A *software implementation* of *transactional memory* (*STM*) provides data representation for transactions and data items using *base objects*, and algorithms, specified as *primitive operations* (abbreviated *primitives*) on the base objects, which *asynchronous* processes have to follow in order to execute the operations of transactions. In addition to ordinary read and write primitives, we allow *arbitrary* read-modify-write primitives, including `CAS`. A primitive is *nontrivial* if it may change the value of the object, e.g., a write or `CAS`; otherwise, it is *trivial*, e.g., a read. An *access* to base object o is the application of a primitive to o.

An *event* is a computation *step* by a process consisting of local computation and the application of a primitive to base objects, followed by a change to the process's state,

according to the results of the primitive. A *configuration* is a complete description of the system at some point in time, i.e., the state of each process and the state of each shared base object. In the unique *initial* configuration, every process is in its initial state and every base object contains its initial value.

Two executions α_1 and α_2 are *indistinguishable* to a process p, if p goes through the same sequence of state changes in α_1 and in α_2; in particular, this implies that p goes through the same sequence of events.

2.1 STM Properties

The consistency condition assumed by all our results is that if a transaction writing to an item t a value other than the initial value, commits, then a later nontransactional read of t returns a value that is different from the initial value; vice versa, if no transaction writing to t commits and no nontransactional write changes t then a nontransactional read of t returns the initial value. This condition is satisfied by *parameterized opacity* [14] and hence our lower bounds hold for parameterized opacity as well.

A transaction *blocks* if it takes an infinite number of steps without committing or aborting. Our progress condition requires a transaction to commit if it has no nontrivial conflict[1] with any pending transaction; that is, a transaction can abort or block only due to a nontrivial conflict with such a transaction. A transaction T is *logically committed* in a configuration C if T does not abort in any infinite extension from C.

Property 1 (l-progressive STM). An STM is *l-progressive*, $l \geq 0$, if a transaction T aborts or blocks in a solo execution (of all the transaction or a suffix of it) after an execution α that contains l or less incomplete transactions, only due to a nontrivial conflict with an incomplete logically committed transaction.

Note that a transaction that must commit according to this definition becomes logically committed at some point, e.g., right before it commits. Property 1 means that, in the absence of conflicts, the STM must ensure parallelism. This property (for any $k \geq 1$) is satisfied by *weakly progressive* STMs [16], in which a transaction must commit if it does not encounter conflicts, and by *obstruction-free* STMs [17], in which a transaction commits when it runs by itself for long enough, implying that it must not abort or block if it runs solo after an execution without nontrivial conflicts.

An *ℓ-independent* execution contains $\ell \geq 0$ transactions, each executed by a different process, $p_{i_1}, \ldots, p_{i_\ell}$, running solo until it is logically committed, on data sets without nontrivial conflicts. An STM is *oblivious* if a transaction running solo after an ℓ-independent execution, without nontrivial conflicts with the pending transactions, behaves in a manner that is independent of the data sets of the pending transactions.

Property 2 (Oblivious STM). An STM is oblivious if for any pair of ℓ-independent executions α_1 and α_2, each containing ℓ transactions executed by the same processes $p_{i_1}, \ldots, p_{i_\ell}$, in the same order, if some transaction T executed by a process p does not have nontrivial conflicts with the transactions in α_1 and α_2, then $\alpha_1 T$ and $\alpha_2 T$ are indistinguishable to p.

[1] A *conflict* occurs when two operations access the same data item; the conflict is *nontrivial* if one of the operations is a write.

An STM has *invisible reads*, if an execution of any transaction is indistinguishable from the execution of a transaction writing the same values to the same items, while omitting all read operations. More formally, consider an execution α that includes a transaction T of process p with write set W and read set R, and consider a transaction T' of process p writing the same values to W in the same order as in T, but with an empty read set. In STMs with invisible reads there is an execution α' that includes T' instead of T, such that α' is indistinguishable to all other processes from α.

2.2 Privatization

An STM may contain transactions that privatize a set of data items. Rather than getting into the details of what privatization means, we only state a property that is naturally expected out of any notion of privatization, as it guarantees isolation when accessing the shared memory with nontransactional operations.

We assume *uninstrumented* nontransactional read operations, which simply read a fixed base object. The base object might depend on the process and the item, e.g., a private (local) copy of the item, but the process applies no manipulation on the value and simply returns the value written in the base object as the value of the item.

Process p_j *privatizes* item t_i when p_j commits a transaction privatizing t_i. The private base object that process p_j associates with a data item t_i, after privatizing the item, is denoted m_i^j. Uninstrumented nontransactional write operations can be defined analogously (although they are not used in our proofs).

Property 3 (Privatization-safe STM). An STM with uninstrumented nontransactional operations is *privatization safe* if after process p_j privatizes item t_i, no process $p_h \neq p_j$ applies a nontrivial primitive to the base object m_i^j.

Previous work [20] informally assumed that a transaction privatizing a region must conflict with other transaction accessing this region.

Indeed, it can be easily shown that a weakly progressive STM cannot support privatization *if the read set of every writing transaction is empty*, unless the privatizing transaction accesses all the items it privatizes. If there is an item the privatizing transaction does not access, then a transaction writing to this item executed after the privatizing transaction completes, is unaware of the privatization, and may access private locations.

We first show that in a privatization-safe STM, a transaction applies a nontrivial primitive (e.g., writes) to a base object associated with a privatized item, only after it is already logically committed.

An STM is *eager* if there exists a configuration C such that a transaction T is the only pending transaction in C, T is not logically committed, and a process p executing T applies a nontrivial primitive to a base object associated with an item that another process privatizes. It is simple to show that a 1-progressive eager STM is not privatization-safe (see [4]). Note that assuming uninstrumented nontransactional operations implies that the mapping of each privatized item to the corresponding base object is static. In the sequel, we assume that a privatization-safe STM is not eager.

3 Privatization with Invisible Reads

The next theorem shows that in an oblivious STMs supporting privatization, the data set of a privatizing transaction must contain all privatized items. The proof proceeds by creating a scenario in which a privatizing transaction misses the up-to-date value of a privatized item; some care is needed in order to argue about each item separately.

Theorem 1. *For any privatization-safe STM that is* 1*-progressive, oblivious and with invisible reads, there is a privatization workload in which transactions have nonempty read sets, for which there is an execution where the size of the data set of a transaction privatizing* k *items is* $\Omega(k)$.

Proof. Consider two processes p_0 and p_1: p_0 executes a transaction T_0 that privatizes the items $t_1, \ldots t_k$. For p_1, consider a transaction T_1' with an arbitrary read set, writing to an item u that is never accessed by T_0.

Consider the execution $\alpha' = I_1^{'} T_0$, such that in $I_1^{'}$, p_1 executes a prefix of the transaction T_1' until it is logically committed (see Figure 2(a)). $I_1^{'}$ is indistinguishable to p_0 from an execution in which p_1 executes a transaction that only writes to u until it is logically committed. After $I_1^{'}$, there is no incomplete transaction that has a conflict with T_0, and since the STM is 1-progressive, T_0 commits when executed after $I_1^{'}$.

Assume, by way of contradiction, that the data set of T_0 does not include some item t_i that it privatizes when executed after $I_1^{'}$. Consider the execution $\alpha = I_1 T_0$, such that in I_1, p_1 executes a prefix of a transaction T_1 with the same read set as T_1' and writing to the item t_i a value different than its initial value, and T_1 is logically committed after I_1 (see Figure 2(b)). It can be shown (see [4]) that t_i is not in the data set of T_0 also when executed after I_1.

Since the reads are invisible I_1 is indistinguishable to p_0 from an execution with no nontrivial conflicts, and since the STM is oblivious, T_0 commits also when executed after I_1 in α. Let $m_1, \ldots, m_k$ be the base objects that are private to p_0 after α (we omit the superscript 0). Since the STM is 1-progressive T_1 commits when completed after α. Since T_1 is logically committed after I_1, it writes to t_i. Consider the execution $I_1 T_0 J_1$, such that J_1 is the suffix of the execution of T_1 until it commits (see Figure 2(c)).

Lemma 1. p_1 *modifies the state of* m_i *in* J_1.

Proof. We first show that p_1 does not modify m_i in the first part of its transaction in α. Assume that p_1 applies a nontrivial primitive to m_i in some step when executing I_1 in α, and let τ be the first such step. Let $\widehat{I}_1$ be the prefix of I_1 preceding τ. I_1 is the shortest prefix of T_1 after which T_1 is logically committed. Hence, T_1 is not logically committed after $\widehat{I}_1$, and it is the only transaction that is pending after $\widehat{I}_1$. In a solo execution of T_0 after $\widehat{I}_1$, T_0 is committed, making m_i private to p_0. Since the STM is not eager, p_1 does not apply τ after $\widehat{I}_1$.

A similar argument shows that p_1 does not apply nontrivial primitive to m_i in α'.

Next, we argue that p_0 does not modify the state of m_i in α. Otherwise, a nontransactional, uninstrumented read operation, to t_i of p_0 after α' returns a value that is not the initial value of m_i, whereas no committed transaction writes to t_i in α', contradicting our correctness condition. The executions of T_0 after I_1 and $I_1^{'}$ are indistinguishable,

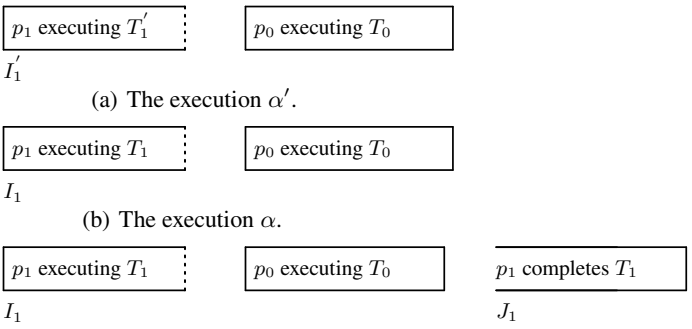


Fig. 2. The executions used in the proof of Theorem 1. A dotted line indicates that the transaction is logically committed.

since the STM is oblivious and since T_0 does not access neither u nor t_i. We have shown that p_1 accesses m_i neither in α nor in α', and hence, p_0 does not successfully apply a nontrivial primitive to m_i also in α.

If p_1 does not modify the state of m_i also in J_1, then a nontransactional read by p_0 to t_i after $I_1 T_0 J_1$ returns the initial value of m_i, since reads are uninstrumented. This contradicts our correctness condition since T_1, which writes to t_i a value that is different from the initial value of m_i, commits before the nontransactional read of p_0. ■

Therefore, p_1 applies a nontrivial primitive to m_i in J_1 after the execution of T_0, in contradiction to privatization safety. ■

The proof allows T_1 and T_1' to have non-empty read sets. Since the reads are invisible, this looks to p_0 as if they have empty read sets. Note however, that p_1 does read from the memory and distinguishes its execution of T_1 and T_1' from executing a transaction with an empty read set. Thus, the result does not follow from the trivial lower bound for transactions with empty read sets.

4 Privatization with Visible Reads

A similar lower bound holds also for STMs with visible reads, assuming they ensure some degree of parallelism. The cost is stated in terms of low-level accesses by the privatizing transaction, rather than in terms of the high-level aspects of the transaction. Some key ideas in the proof are similar to the proof of Theorem 1; however, the technical details are more involved, in order to handle visible reads.

Theorem 2. *For any privatization-safe STM that is l-progressive and oblivious there is a privatization workload in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing m items accesses $\Omega(k)$ base objects, where $k = \min\{l, m\}$.*

The proof of this theorem is quite involved technically, so we outline it for the linked-list workload of Figure 1; a detailed proof sketch appears in Appendix A.

We have k *updating* transactions traverse the nodes of a linked list, while each transaction writes to a different item pointed by the list that is not read by other transactions; so these transactions have only trivial conflicts. Later, a transaction by another process, privatizing all items pointed by the linked list, is shown to miss the up-to-date value of the privatized items, unless it accesses many base objects.

Since reads are visible, however, it is difficult to hide the updating transaction from the privatizing transaction. The challenge is to create an execution in which an updating transaction runs long enough to guarantee that it will eventually commit—even after the privatizing transaction commits, and even if the privatizing transaction writes to an item it reads—but not long enough to become visible to the privatizing transaction.

The privatizing transaction may write to an item in the read set of an updating transaction (e.g., the head of the list), thus invalidating its read set. Hence, to guarantee that an updating transaction eventually commits in the execution constructed, the updating transaction runs until it is logically committed, before the privatizing transaction starts.

It may seem that, at this point, the privatizing transaction does not need to access many objects to observe a conflict with the updating transactions, and it can abort or at least block until the conflicts are resolved. However, the obliviousness and non-eagerness of the STM can be used to "hide" the updating transactions from the privatizing transaction, by swapping the updating transactions with other *confusing* transactions, accessing a completely disjoint linked list; the confusing transactions also have only trivial conflicts among them. Due to obliviousness, these confusing transactions are indistinguishable from the original updating transactions.

We start with an execution in which the confusing transactions run one after the other; this execution is k-independent. Then, we swap confusing transactions with updating transactions. Swapping is done inductively: Each inductive step swaps one confusing transaction with an updating transaction by the same process; that is, at each step one additional process executes the updating transaction instead of the confusing transaction, and *incurs an access to at least one additional base object* by the privatizing transaction. This yields an execution in which the privatizing transaction accesses many objects, implying the lower bound.

Progressiveness is used to ensure that if at some point the privatizing transaction observes a conflict, the updating transaction causing the conflict may run to completion. This also ensures that the privatizing transaction runs to completion.

A technical challenge in the proof is in deciding which transaction to swap next, so as not to lose the accesses by the privatizing transaction that appear in the execution we have created so far. Specifically, we need to pick a transaction T such that swapping it is invisible to the privatizing transaction in its execution prefix, at least during the memory accesses incurred due to previous swaps. This is done by letting T be the last transaction to modify the next location seen by the privatizing transaction, so that future swaps will not overwrite locations T writes to and that are accessed by the privatizing transaction in its execution prefix. (This is the purpose of Item 5 maintained in the inductive construction; see Appendix A.)

Reducing the cost of privatization by restricting parallelism: The lower bound, stated as the minimum between the number of privatized items and the level of parallelism, indicates that one way to reduce the cost associated with privatization, is to limit the

parallelism offered by the STM. We next show how this tradeoff can be exploited, by sketching a "counter-example" STM, which is a variant of RingSTM [31]. The variant, called *VisibleRingSTM*, reduces the cost of privatization while limiting parallelism.

RingSTM is oblivious and privatization-safe, but not progressive; privatizing k items accesses $O(c)$ base objects, where c is the number of concurrent transactions. RingSTM represents transactions' read and write sets as Bloom filters [6]. Transactions commit by enqueuing a Bloom filter onto a global ring; the Bloom filter representing the read set of a transaction is only for its internal use. During validation, a transaction T checks for intersections between the read set of T and the write sets of other logically committed transactions in the ring, and aborts in case of a conflict. In the commit phase, T ensures that a write-after-write ordering is preserved. This is done by checking for intersections between the write set of T and the write sets of other logically committed transactions in the ring. RingSTM is not l-progressive, for any l, since a transaction blocks until all concurrent logically committed transactions are completed.

In VisibleRingSTM, the read set Bloom filter is visible to other transactions, like the filter of the write set. In the commit phase, T ensures that a write-after-read ordering is preserved, in addition to the write-after-write ordering, as in RingSTM. This is done by checking for intersections between the write set of T and the (visible) read sets of other logically committed transactions in the ring (in addition to checking for intersections with the write sets of these transactions). Intersection between the read set of T and the write set of another transaction is checked by validation. There is no need to check for intersection between read sets, as these are trivial conflicts that should not interfere. Finally, waiting for all logically committed transactions to complete (at the end of the commit phase) is removed in VisibleRingSTM, as the write-after-read and write-after-write ordering ensure that all the concurrent conflicting transactions have completed.

In VisibleRingSTM, a transaction aborts only due to read-after-write conflicts with other logically committed transactions, and blocks after it is logically committed only due to write-after-write or write-after-read conflicts with other logically committed transactions. A privatizing transaction accesses the c ring entries of concurrent logically committed transactions, the items in its data set and the global ring index.

The cost of a privatizing transaction can be bounded by $O(c_0)$, for any $c_0 > 1$, by using a ring of size c_0; thus, a privatizing transaction needs to access at most c_0 ring entries. In order to commit, a transaction scans the ring for an empty entry. When there are at most c_0 concurrent transactions, it will find an empty entry, become logically committed, and continue as in VisibleRingSTM. This STM is $(c_0 - 1)$-progressive, but a transaction blocks in executions with more than c_0 concurrent transactions (even if they are not conflicting). Thus, the cost of privatization is reduced by limiting the progress of concurrent transactions.

5 Related Work

Many STMs supporting privatization are oblivious [8, 31, 23, 11, 13, 9] because they avoid the cost of tracking the read sets of other transactions, especially if they are not conflicting. The visibility of reads is not induced by the obliviousness of the STM: Some oblivious STMs use invisible reads [8, 31, 23], making their read set nonexistent

for other transactions. Other STMs, e.g., [9], use *partially visible* reads [21], implying that other transactions cannot determine which transaction exactly is reading the item. Some oblivious STMs even use visible reads, e.g., [13], however, their execution is unaffected by trivial, read-read conflicts. Our lower bounds hold for all these STMs.

TLRW [11] uses read locks, making reads visible. The lock contains a byte per each *slotted* reader, and a reader-count that is modified by other, *unslotted* readers. Slotted readers only write to their slot when reading, so they are unaware of other reads, while unslotted processes read and write to a common counter, and their execution is affected by other reads to the same item (read-read conflict). Therefore, TLRW is oblivious when restricted to slotted readers, and, as predicted by our lower bound, the number of locations accessed by a privatizing transaction is linear in the number of slotted readers.

Our lower bounds indicate that providing efficient privatization requires to compromise parallelism. Inspecting many STMs supporting privatization, e.g., [13,9,24,8,21,31,23], reveals that they limit parallelism, in one way or another.

In *explicit privatization* [29], the application explicitly annotates privatizing transactions, and the STM implementation can be optimized to handle such transactions efficiently; this approach is error-prone and places additional burden on the programmer, which STM tries to avoid in the first place [20]. In *implicit privatization* [21], the STM implementation is required to handle all transactions as if they are potentially privatizing items; this incurs excessive overhead for all transactions.

Some experiments [33,12,29] tested techniques used to support implicit privatization in implementations with invisible reads. The results show a significant impact on the scalability and performance relative to STMs supporting explicit privatization; in some cases, the performance degrades to be worse than in sequential code.

Parameterized opacity [14] is a framework for describing the interaction between transactions and nontransactional operations, extending *opacity* [15], and parameterized by a memory model for the semantics of nontransactional operations. Guerraoui et al. [14, Theorem 1] prove that parameterized opacity cannot be achieved for memory models that restrict the order of some pair of read or write operations to different variables. They also show that if operations on different variables can be reordered, then parameterized opacity with uninstrumented operations requires RMW primitives when writing inside a transaction. Their results assume that items are accessed nontransactionally, without a preceding privatization transaction. These results indicate the *implications of not privatizing*, while our results show *the cost of privatization*.

They also present an uninstrumented STM that guarantees parameterized opacity with respect to memory models that do not restrict the order of any pair of read or write operations; it uses a global lock, and is not weakly progressive. In the full version of this paper [4], we show that a 1-progressive, oblivious uninstrumented STM, cannot achieve opacity parameterized with respect to *any* memory model. This indicates that a privatizing transaction must precede nontransactional accesses to data items, unless parallelism is compromised.

Private transactions [9] attempt to combine the ease of use of implicit privatization with the efficiency benefits of explicit privatization. A private transaction inserts a *quiescing barrier* that waits till all active transactions have completed; thus other, non-privatizing transactions avoid the overhead of privatization. The barrier accesses

an array whose size is proportional to the maximal parallelism, demonstrating again the tradeoff between parallelism and privatization cost, in oblivious STMs.

Static separation [2] is a discipline in which each data item is accessed either only transactionally or only nontransactionally. In order to privatize items, the transaction copies them to a private buffer, trivially demonstrating our lower bound. *Dynamic separation* [1] allows data to change access modes without being copied, simply by setting a protection mode in the item. Dynamic separation requires the programmer to access all items to become unprotected, i.e., privatized, as is indicated by our lower bound.

6 Discussion

This paper studies the theoretical complexity of privatization that allows uninstrumented nontransactional reads, and shows an inherent cost, linear in the number of privatized items. Privatizing transactions in STMs with invisible reads must have a data set of size k, where k is the number of privatized items. A more involved proof shows that even with visible reads, the privatizing transaction must access $\Omega(k)$ memory locations, where k is the minimum between the number of privatized items and the number of concurrent transactions that make progress. Both results assume that the STM is oblivious to different non-conflicting executions and guarantees progress in such executions. The specific assumptions needed to prove the bounds indicate that limiting the parallelism or tracking the data sets of other transactions are the price to pay for efficient privatization.

The privatization problem is informally characterized by two subproblems: The *delayed cleanup* problem [19], in which transactional writes interfere with nontransactional operations, and the *doomed transaction* problem [32], in which transactional reads of private data lead to inconsistent state. Our definition of privatization safety (Property 3) formalizes the first problem; our results show that this problem by itself is an impediment to the efforts to provide efficient privatization.

As discussed in Section 5, some STMs maintain visible reads, yet they are oblivious [9, 13]. SkySTM [20] has visible reads, and it avoids the cost of the privatizing transaction by not being oblivious; it makes transactions with trivial read-read conflicts visible to each other. Since SkySTM is not oblivious, our lower bounds do not hold for it. SkySTM, however, demonstrates the alternative cost of not being oblivious, since any writing transaction—not only privatizing transactions—writes to a number of base objects that is linear in the size of its data set, not just the write set. It remains an interesting open question whether this is an inherent tradeoff, or whether there is an STM such that a privatizing transaction accesses $O(1)$ base objects, and any writing transaction writes to a number of base objects that is linear in the size of its write set.

Strong privatization safety [20] further guarantees that no primitive (including a read) is applied to a private location of a process that completed a privatizing transaction. It formalizes the other problem with privatization, of doomed transactions, and it would be interesting to investigate the cost of supporting it.

Acknowledgements. We thank Keren Censor Hillel, Panagiota Fatourou, Petr Kuznetsov, Alessia Milani, and Michael Spear for helpful comments.

References

1. Abadi, M., Birrell, A., Harris, T., Hsieh, J., Isard, M.: Implementation and use of transactional memory with dynamic separation. In: CC '09, pp. 63–77 (2009)
2. Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: POPL '08, pp. 63–74 (2008)
3. Abadi, M., Harris, T., Mehrara, M.: Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In: PPoPP'09, pp. 185–196 (2009)
4. Attiya, H., Hillel, E.: The cost of privatization. Technical Report CS-2010-11, Department of Computer Science, Technion. (2010)
5. 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. 131–140. Springer, Heidelberg (2008)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. ACM Commun. 13(7), 422–426 (1970)
7. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: why is it only a research toy? ACM Commun. 51(11), 40–46 (2008)
8. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by abolishing ownership records. In: PPoPP '10 (2010)
9. Dice, D., Matveev, A., Shavit, N.: Implicit privatization using private transactions. In: TRANSACT '10 (2010)
10. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
11. Dice, D., Shavit, N.: TLRW: Return of the read-write lock. In: SPAA '10 (2010)
12. Dragojevic, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a Research Toy. Technical Report LPD-REPORT-2009-003, EPFL (2009)
13. Gottschlich, J.E., Vachharajani, M., Jeremy, S.G.: An efficient software transactional memory using commit-time invalidation. In: CGO'10 (2010)
14. Guerraoui, R., Henzinger, T., Kapalka, M., Singh, V.: Transactions in the jungle. In: SPAA '10, pp. 275–284 (2010)
15. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP '08, pp. 175–184 (2008)
16. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. In: POPL '09, pp. 404–415 (2009)
17. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS '03, p. 522 (2003)
18. Israeli, A., Rappoport, L.: Disjoint-access parallel implementations of strong shared memory primitives. In: PODC '94, pp. 151–160 (1994)
19. Larus, J.R., Rajwar, R.: Transactional Memory. Morgan & Claypool, San Francisco (2006)
20. Lev, Y., Luchangco, V., Marathe, V.J., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: TRANSACT '09 (2009)
21. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable techniques for transparent privatization in software transactional memory. In: ICPP '08, pp. 67–74 (2008)
22. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. 5(2), 17 (2006)
23. 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: SPAA '08, pp. 314–325 (2008)

24. Olszewski, M., Cutler, J., Steffan, J.G.: JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In: PACT '07, pp. 365–375 (2007)
25. 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)
26. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: SPAA '07, pp. 221–228 (2007)
27. Schneider, F.T., Menon, V., Shpeisman, T., Adl-Tabatabai, A.-R.: Dynamic optimization for efficient strong atomicity. SIGPLAN Not. 43(10), 181–194 (2008)
28. 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. SIGPLAN Not. 42(6), 78–88 (2007)
29. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-based semantics for software transactional memory. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 275–294. Springer, Heidelberg (2008)
30. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. Technical Report Tr 915, Dept. of Computer Science, Univ. of Rochester (2007)
31. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: scalable transactions with a single atomic instruction. In: SPAA '08, pp. 275–284 (2008)
32. 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: CGO '07, pp. 34–48 (2007)
33. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., Lee, H.-H.S.: Kicking the tires of software transactional memory: why the going gets tough. In: SPAA '08, pp. 265–274 (2008)

A More Details for the Proof of Theorem 2

We sketch the proof for a workload similar to the linked list of Figure 1; after the proof, we discuss how it can be extended to other scenarios.

Theorem 2. *For any privatization-safe STM that is l-progressive and oblivious there is a privatization workload in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing m items accesses $\Omega(k)$ base objects, where $k = \min\{l, m\}$.*

Sketch of proof. Consider the scenario described in Figure 1: Let $r_0, r_1, \ldots, r_k$ be the nodes of the linked list (with r_0 being the head node); each node r_i, $1 \leq i \leq k$ points to an item t_i. We consider $k+1$, $k \geq 1$, processes $p_0, \ldots, p_k$. Process p_0 executes a transaction T_0 that privatizes the linked list, including the items $t_1, \ldots t_k$. For every process p_i, $1 \leq i \leq k$, we consider a transaction T_i that traverses the nodes of the list and then writes to t_i *a value different than its initial value*; namely, its read set is $\{r_0, r_1, \ldots, r_i\}$, and its write set is $\{t_i\}$.

For the proof, we take another linked list with the same structure that is not connected to the first list in any way. This list contains $k+1$ nodes $r'_0, r'_1, \ldots, r'_k$ and k items $t'_1, \ldots, t'_k$. T_0 does not access this list at all; however, for every process p_i, $1 \leq i \leq k$, we consider another transaction, $T_i{}'$, that traverses the nodes of the second list and then writes to t'_i; namely, its read set is $\{r'_0, r'_1, \ldots, r'_i\}$, and its write set is $\{t'_i\}$.

A process p *reads from* a process q *via* a base object o in an execution α if p accesses o, and o was last modified by q. Process p reads from a *set* of processes P in an execution α if for every process $q \in P$, there is a base object o such that p reads from q via o in α.

Consider the following execution $\alpha_0\beta_0\gamma_0$: α_0 is $I_1' \ldots I_k'$, such that p_i executes in I_i', a prefix of the transaction T_i' and T_i' is logically committed after I_i'; β_0 is the empty execution interval; and γ_0 is a solo execution of T_0 by p_0 to completion.

For every ℓ, $0 < \ell \leq k$, we show how to perturb $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ to obtain an execution $\alpha_\ell\beta_\ell\gamma_\ell$, such that

1. p_0 executes T_0 to successful completion in $\beta_\ell\gamma_\ell$.
2. p_0 reads from all processes in P_ℓ, a subset of $\{p_1, \ldots, p_k\}$ of size (at least) ℓ, in $\alpha_\ell\beta_\ell$.
3. There is a subset Q_ℓ of P_ℓ, where every process $p_j \in Q_\ell$ executes I_j, a prefix of the transaction T_j, in α_ℓ, such that T_j is logically committed after I_j, and p_j completes T_j in β_ℓ.
4. Every process p_j, $\{p_1, \ldots, p_k\} \setminus Q_\ell$, executes I_j' in α_ℓ.
5. For every process p_j from which p_0 does not read in $\alpha_\ell\beta_\ell\gamma_\ell$, α_ℓ^j is a k-independent execution, in which p_j executes I_j instead of I_j', and all other processes take the same steps as in α_ℓ; in α_ℓ^j, p_j does not modify any base object o, such that, in $\alpha_\ell\beta_\ell$ p_0 reads o from a process p_h, $h < j$.

For $\ell = k$, we get an execution $\alpha_\ell\beta_\ell\gamma_\ell$, such that p_0 reads from k different processes in P_k (Condition 2). The theorem follows since p_0 accesses k different base objects,

The proof is by induction on ℓ. The base case is straightforward (see [4]). For the induction step, assume an execution $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ satisfies the above conditions. Consider the subset $V_{\ell-1}$ of $\{p_1, \ldots, p_k\} \setminus P_{\ell-1}$ from which p_0 does not read in $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$. If $V_{\ell-1}$ is empty then p_0 reads from all the processes and the theorem holds. Otherwise, one of the processes in $V_{\ell-1}$ is used to construct the next step.

Pick an arbitrary process $p_j \in V_{\ell-1}$ and consider the execution $\alpha_{\ell-1}^j$, in which p_j executes I_j instead of I_j', and other processes take the same steps as in α_ℓ.

The execution $\alpha_{\ell-1}^j$ is k-independent, since all the transactions are nonconflicting. Since the STM is oblivious, $\alpha_{\ell-1}^j$ is a valid execution, and since the STM is k-progressive T_j is logically committed in $\alpha_{\ell-1}^j$. Since the STM is oblivious, $\alpha_{\ell-1}^j$ is indistinguishable to every process in $\{p_1, \ldots, p_k\} \setminus \{p_j\}$ from $\alpha_{\ell-1}$. Furthermore, the inductive assumption implies that p_j does not modify in $\alpha_{\ell-1}^j$ any base object o, if in $\alpha_{\ell-1}\beta_{\ell-1}$ p_0 reads o from a process p_h, $h < j$. Thus, p_0 reads the same values as in the execution of $\beta_{\ell-1}$, and there is an execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ such that $\beta_{\ell-1}^j$ and $\beta_{\ell-1}$ are indistinguishable, and p_0 runs solo in $\gamma_{\ell-1}^j$.

Assume that p_0 does not read from p_j also in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Then p_0 takes the same steps in $\gamma_{\ell-1}^j$ and $\gamma_{\ell-1}$ and T_0 is committed in $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Let $m_1, \ldots, m_k$ be the base objects that are private to p_0 after $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$.

The pending transactions in the execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ are not conflicting. Since the STM is k-progressive and T_j is logically committed after I_j, it commits (writing to t_j) when executed solo after $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$. Consider the execution $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j J_j$,

such that J_j is the execution of T_j until it commits. In a manner similar to Lemma 1, we show that p_j must modify the state of m_j in J_j (see [4]). Therefore, p_j applies a nontrivial primitive to m_j in some step during J_j, which is pending after the execution of T_0, in contradiction to privatization safety. Thus, p_0 must read from p_j in $\alpha^j_{\ell-1}\beta^j_{\ell-1}\gamma^j_{\ell-1}$.

Let s_j be the number of steps until p_0 reads from p_j for the first time in $\gamma^j_{\ell-1}$. Pick a process p_{j_ℓ} such that s_{j_ℓ} is the smallest, and if $s_{j_\ell} = s_{h_\ell}$ then $j_\ell > h_\ell$.

Let the execution interval α_ℓ be $\alpha^{j_\ell}_{\ell-1}$. The execution interval β_ℓ is $\beta_{\ell-1}$ extended with the first $s_{j_\ell} - 1$ steps of p_0 in $\gamma^{j_\ell}_{\ell-1}$, then a solo execution of p_{j_ℓ} completing T_{j_ℓ}, and finally, the s_{j_ℓ} step of p_0 from $\gamma^{j_\ell}_{\ell-1}$, which reads from p_{j_ℓ}. Since T_{j_ℓ} is logically committed in α_ℓ, and the STM is k-progressive, T_{j_ℓ} commits in β_ℓ. The execution interval γ_ℓ is defined as a solo execution of p_0 completing T_0.

It remains to verify the conditions hold for $\alpha_\ell\beta_\ell\gamma_\ell$.

1. T_0 completes successfully as there is no incomplete conflicting transaction after $\alpha_\ell\beta_\ell$, and the STM is k-progressive.
2. By the induction assumption, p_0 reads from at least $\ell - 1$ processes, $P_{\ell-1}$, in $\alpha_{\ell-1}\beta_{\ell-1}$, not including p_{j_ℓ} that was chosen in the last iteration. The executions $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes p_h, for $h < j_\ell$. Furthermore, since the STM is oblivious, $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes p_h, for $h > j_\ell$. Hence, p_0 reads from at least the same $\ell - 1$ processes in $\alpha_\ell\beta_{\ell-1}$. In addition, p_0 reads from p_{j_ℓ} in $\alpha_\ell\beta_\ell$. Thus, $P_\ell \supseteq P_{\ell-1} \dot\cup \{p_{j_\ell}\}$, and $|P_\ell| \geq |P_{\ell-1}| + 1 \geq \ell$.
3. By the induction assumption, $Q_{\ell-1}$ is a subset of $P_{\ell-1}$, such that every process $p_h \in Q_{\ell-1}$ executes I_h in $\alpha_{\ell-1}$, and completes T_h in $\beta_{\ell-1}$. Only p_{j_ℓ} is in $Q_\ell \setminus Q_{\ell-1}$, and it executes I_{j_ℓ} in α_ℓ and completes T_{j_ℓ} in β_ℓ. Since $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes in $\{p_1, \ldots, p_k\} \setminus \{p_{j_\ell}\}$, and since only p_{j_ℓ} switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ, all the processes $p_h \in Q_\ell \setminus \{p_{j_\ell}\}$ execute I_h in α_ℓ and complete T_h in $\beta_{\ell-1}$, which is the prefix of β_ℓ.
4. By the induction assumption, every process $p_h \in \{p_1, \ldots, p_k\} \setminus Q_{\ell-1}$, executes I'_h in $\alpha_{\ell-1}$. Since only $p_{j_\ell} \in \{p_1, \ldots, p_k\} \setminus Q_{\ell-1}$ switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ, and since $p_{j_\ell} \notin \{p_1, \ldots, p_k\} \setminus Q_\ell$, every process $p_h \in \{p_1, \ldots, p_k\} \setminus Q_\ell$ executes I'_h in α_ℓ.
5. Assume, by way of contradiction, that for some j, p_j modifies in α^j_ℓ the base object o, which in $\alpha_\ell\beta_\ell$ p_0 reads from p_h, $h < j$. Denote by σ the step during $\alpha_\ell\beta_\ell$, in which p_0 reads from p_h. There is a step, σ', which is either σ or follows σ during $\alpha_\ell\beta_\ell$, in which p_0 reads from $p_{h_{\ell'}} \in Q_\ell$, since p_0 always reads from a process in Q_ℓ in the last step of $\alpha_\ell\beta_\ell$. Consider the iteration, ℓ', in which $p_{h_{\ell'}}$ was chosen to switch from $T_{h_{\ell'}}{}'$ to $T_{h_{\ell'}}$. Since the STM is oblivious, the executions $\alpha_{\ell'-1}$ and $\alpha_{\ell'}$ are indistinguishable to the processes in $\{p_1, \ldots, p_k\} \setminus \{p_{h_{\ell'}}\}$. Process p_j should have been chosen in the iteration ℓ', since if σ' follows σ, then $s_j < s_{h_{\ell'}}$, otherwise, σ' is σ, i.e., $h = h_{\ell'}$ and $j > h_{\ell'}$. ■

The proof holds for every workload in which the updating transaction T_i does not read any item from $\{t_1, \ldots t_k\} \setminus \{t_i\}$.

A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware

Francois Carouge and Michael Spear

Lehigh University
{frc309,spear}@cse.lehigh.edu

Abstract. The imminent arrival of best-effort transactional hardware has spurred new interest in the construction of nonblocking data structures, such as those that require atomic updates to k words of memory (for some small value of k). Since transactional memory itself (TM) was originally proposed as a *universal* construction for crafting scalable lock-free data structures, we explore the possibility of using this emerging transactional hardware to implement a scalable, unbounded transactional memory that is simultaneously nonblocking and compatible with strong language-level semantics. Our results show that it is possible to use this new hardware to build nonblocking TM systems that perform as well as their blocking counterparts. We also find that while the construction of a lock-free TM is possible, correctness arguments are complicated by the many caveats and corner cases that are built into current transactional hardware proposals.

1 Introduction

Transactional Memory (TM) [12] was initially proposed as a hardware mechanism to support lock-free programming. The subsequent development of Software Transactional Memory (STM) [21] continued this focus on nonblocking progress. Indeed, it was two nonblocking algorithms, the obstruction-free DSTM [11] and lock-free FSTM [10] that reinvigorated STM research in the past decade. However, as processors hit the heat wall and vendors embraced parallelism across the entire spectrum of products, the urgent need for low-latency STM algorithms led to a flurry of research into lock-based designs. Roughly speaking, lock-based STM algorithms are easier to design and implement, and with careful design (and OS support) are resilient to preemption [4] and priority inversion. Some lock-based STM algorithms are even livelock-free [25, 3, 20]. These results have not stopped exploration into nonblocking STM, and two recent obstruction-free STM designs show how a blocking system can be made nonblocking [16, 27] without much overhead.

The prospect of integrating STM into high-level languages presents a further complication. If data is to be used both transactionally and nontransactionally, then an STM implementation may introduce races between active or committing transactions and concurrent nontransactional code [24, 28, 18]. These races are

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 50–63, 2010.

implementation artifacts; they affect race-free code because the STM implementation does not provide strong correctness guarantees. While some STM designs are privatization-safe [25,3,17,6,15], solutions to the "privatization" problem appear to require blocking, either at transaction boundaries [24,17,18,19], or during the execution of transactional [15,6], or even nontransactional [22] code.

Two recent innovations provide hope that these obstacles can be overcome. First, "single-CAS" STM (SCSTM) algorithms [20,25,3] show that it is possible to construct scalable lock-based systems that are privatization-safe, livelock-free, and whose blocking behavior is both localized and easy to characterize. Secondly, interest in bounded or "best-effort" hardware support for transactional memory (BETM) promises to extend common instruction-set architectures with features to facilitate lock-free programming. It has been shown that a simple hardware extension called alert-on-update (AOU) [26] can make a traditional (weak semantics) STM implementation obstruction-free. However, the AOU proposal ignored implementation details, whereas BETM proposals do not. Given the weaker guarantees of BETM, we explore the creation of *lock-free* STM implementations that are scalable and provide strong semantics.

Our results are mixed. While using BETM is relatively easy, arguing about the correctness of a BETM algorithm is more difficult, as it requires knowledge of the underlying hardware, careful programming to avoid corner cases, and a mix of transactional and nontransactional accesses to metadata and program data. Assuming that we have not missed any causes of spurious aborts in the BETM systems we considered, we have made three SCSTM algorithms obstruction-free or lock-free without sacrificing performance or safety. The result is strong: using very little of what BETM offers, we can build algorithms that are lock-free, provide strong semantics, and scale well.

This paper is organized as follows. In Sections 2 and 3 we describe the hardware and software foundations of our new lock-free construction. We then present our lock-free construction in 4. In Section 5, we evaluate our nonblocking algorithms through simulation. We conclude in Section 6.

2 Background: Best Effort Transactional Hardware

Recent BETM systems from Sun Microsystems (the "Rock" processor [5]) and AMD (the "advanced synchronization facility" proposal [7], or ASF) embody a significant compromise: whereas previous HTM systems attempt to transparently support long-running transactions with large memory footprints, these best-effort systems provide limited guarantees. In exchange, BETMs are implementable in modern instruction set architectures. We briefly summarize the key attributes below:

- **Capacity:** BETMs limit the amount of data that can be accessed within a transaction. In ASF, transactions that speculatively access four locations or fewer will not fail due to capacity (the upper bound is not specified); in Rock, the L1 cache manages speculative loads, and a 32-entry store queue bounds the number of speculative writes. A software fallback is required if these resources are insufficient.

- **Mixed-Mode Access:** A transaction can execute "nonspeculative" loads and stores to lines that have not been accessed speculatively by the same transaction. These accesses can cause concurrent transactions in other processors to abort, but do not result in conflicts that cause the caller to abort. In ASF, nonspeculative stores occur immediately, which slightly alters the processor memory model. In Rock, nonspeculative stores are buffered in the 32-entry store queue, and do not occur until the transaction commits or aborts.
- **Progress:** BETM detects conflicts at the granularity of cache lines, using the "requester wins" model. Since "requester wins" resolves conflicts in favor of the most recent access to a location, it is prone to livelock [1]: from the moment transaction T speculatively accesses line X until it commits, any concurrent incompatible access from another processor will cause T to abort.
- **Forbidden Instructions:** BETM forbids certain instructions within a transaction. System calls are never allowed, and Rock also forbids floating point divide and certain control flow patterns.
- **Forbidden Events:** BETM transactions cannot survive a context switch. In addition, hardware exceptions and asynchronous interrupts cause the active transaction to abort. Particularly problematic are TLB misses and page faults. In Rock, some TLB misses can cause a hardware transaction to abort, and the TLB miss may need to be triggered outside of the transactional region before retrying. Page faults also cause hardware transactions to abort, with Rock requiring the fault to be re-created outside of the transaction before retrying.

To summarize, BETM transactions are limited in terms of the number of locations they can access speculatively, the number of locations they can access nonspeculatively, and the instructions they can execute. Interrupts and exceptions cause transactions to abort, and it is likely that there are undocumented reasons why a transaction in isolation may spuriously abort. Transactions that perform speculative stores are prone to livelock, and in general some form of contention management or software fallback is recommended. Nonetheless, we will show that BETM can be used to create a scalable, nonblocking, privatization-safe, unbounded STM.

3 Background: Single-CAS STM

The software foundation for our lock-free construction is the concept of Single-CAS STM (SCSTM). Example implementations of SCSTM include JudoSTM [20], RingSTM [25], and TMLLazy/NOrec [3]. These blocking algorithms are characterized by (1) livelock-freedom, (2) strong semantics, and (3) a simple linearizability condition [13].

3.1 SCSTM Behavior

Figure 1 presents a template for SCSTM algorithms. Specific SCSTM algorithms can be thought of as instantiations of this code using different metadata for

conflict detection. The general behavior of SCSTM algorithms is very simple: concurrent updates are mediated via a single monotonically increasing counter. This counter serves both as a lock that protects shared memory when a transaction is finalizing its writes, and as a flag that is polled by in-progress transactions to determine when it is necessary to perform intermediate correctness checks ("validation").

```
TMBegin:
 1  create_checkpoint()
 2  // wait if global lock is held
 3  while ((my_ts = global_ts) & 1)
 4    spin()

TMReadWord(addr):
 1  // check for buffered write
 2  if (writes.contains(addr))
 3    return writes.lookup(addr)
 4  // read from shared memory
 5  tmp = *addr
 6  // check for new commits...
 7  // if found, validate, then
 8  // re-read location
 9  while (my_ts != global_ts)
10    TMValidate()
11    tmp = *addr
12  // log address for future
13  // validation
14  readset.add(addr, tmp)
15  return tmp

TMWriteWord(addr, val):
 1  // record store in private buffer
 2  writes.insert(addr, val)
 3  // log address so it's easy to
 4  // announce set of modifications
 5  // at commit time
 6  writeset.add(addr)

TMAbort():
 1  reset_per_thread_metadata()
 2  restore_checkpoint()
```

```
TMEnd:
 1  // read-only fastpath
 2  if (writes.empty())
 3    reset_per_thread_metadata();
 4    return
 5  // acquire global lock before
 6  // committing.  Only attempt
 7  // to get lock when tx is
 8  // known to be in a valid state
 9  while (!CAS(global_ts, my_ts, my_ts+1)
10    TMValidate()
11  // announce set of changes
12  publish_writeset()
13  // update main memory
14  for <addr, val> in writes
15    *addr = val
16  // release lock
17  global_ts++
18  reset_per_thread_metadata();

TMValidate():
 1  while (true)
 2    // wait if writeback is in progress
 3    tmp = global_ts
 4    if (tmp & 1)
 5      spin()
 6      continue
 7    // ensure reads still valid
 8    if (!all_reads_still_valid())
 9      TMAbort()
10    // must re-validate if another
11    // transaction committed
12    if global_ts == tmp
13      my_ts = tmp
14      return
```

Fig. 1. Generic pseudocode for Single-CAS STM algorithms

Transactions buffer all updates to shared memory, and then commit by incrementing the counter to odd, optionally publishing some metadata describing the changes they are making to memory, committing their writes to memory, and then incrementing the counter again to even. Every read operation first checks for a pending local write, and then if no such write exists, performs a read from main memory. After reads, the global counter is polled to check if any new transactions have committed, possibly invalidating this transaction. If any are found, the transaction validates its entire read set and aborts if any reads have been invalidated by another transaction's commit.

A transaction only blocks when another transaction is in the process of committing. When writing transaction T is ready to commit (line 9 of `TMEnd`), its `my_ts` field stores the even value of the global counter (`global_ts`) at the last

time that T performed a validation. Clearly, at that time T's reads and writes were all valid, or else T would have aborted. If T can atomically increment `global_ts` from `my_ts` to one greater, then it is sure no other writer transaction R committed in the interim. For R to have committed first, it would have had to increment `global_ts`, which would have ensured that T's increment of `global_ts` would fail. If T cannot successfully increment `global_ts`, then it re-validates (and thus updates its local `my_ts`) before trying again.

The code in Figure 1 is general enough to describe NOrec, RingSTM, and TMLLazy: In TMLLazy, no `readset` or `writeset` is maintained, and calls to `TMValidate` are be replaced with calls to `TMAbort`. In RingSTM, the `writeset` is a signature that is published by appending it to a global list, and `TMValidate` intersects the per-thread `readset` (also a signature) with new entries on the list. In NOrec, no `writeset` is maintained, so `TMValidate` iterates through the {address, value} pairs in the `readset`, making sure that each address still holds the expected value.

3.2 SCSTM Properties

In SCSTM, an in-flight transaction only blocks when another transaction is performing writeback (lines 11–15 of `TMEnd`). During writeback, reads and commits cannot be guaranteed an atomic view of metadata or shared memory, and thus threads wait until the lock is released (`TMBegin` lines 3–4, and `TMValidate` lines 3–6, called from `TMReadWord` and `TMEnd`).

By virtue of its single lock and commit-time validation protocol, SCSTM is livelock-free. The argument for livelock freedom in SCSTM is simple: a transaction only aborts if it fails a validation, and it only validates when the global counter is updated. Since the global counter is only updated when a transaction commits, the chain of events leading to an abort always begins with a transaction committing [25].

The single lock also serves to completely serialize the commit phase of writing transactions. On architectures with nonfaulting loads, the combination of polling on every read, validating whenever *any* transaction commits, and completely serializing writeback leads to privatization safety [24]. When nonfaulting loads are unavailable, privatization safety requires some additional measures to handle OS reclamation of memory in the middle of a concurrent `TMReadWord` operation [14,8]. Thus SCSTM provides at least "ELA" semantics in the taxonomy of Menon et al. [18].

4 The Lock-Free Transformation

We next make SCSTM nonblocking using BETM. Our approach resembles that proposed by Spear et al. [26], where Alert-On-Update (AOU) was used to interrupt a thread while it held the lock protecting an idempotent critical section. The main differences in our technique center around the use of BETM. Specifically, previous work assumed an idealized TM in which the only cause of transactional

aborts is transaction conflicts. We consider all published causes of BETM aborts, which makes our construction considerably more complex, but also realistic.

4.1 Making Stealing Safe

Our goal is to allow multiple threads to perform the same writeback operation simultaneously, and interrupt any partial writebacks when one thread completes the full writeback. This means that after a transaction has made `global_ts` odd on `TMEnd` line 9, any thread can perform the rest of the commit operation on that transaction's behalf.

To make writeback stealable, we must first make the list of pending `writes` visible. There are two means to provide this; either we can superimpose a pointer over the `global_ts` variable, such that an odd least significant bit indicates that the remaining bits are a pointer to the `writeset` (and any other needed metadata, such as "`next_ts`", the value to which `global_ts` must be set after writeback), or else we can use a hardware transaction to perform a multiword store. We use the former option for NOrec and TMLLazy, since `TMEnd` line 12 is a no-op, and the latter for RingSTM, since line 12 of `TMEnd` must be atomic with line 9.

Secondly, we must ensure that the result of multiple threads performing overlapping portions of the writeback simultaneously remains correct. In Spear's AOU work, writeback was done on a per-object basis, with at most one thread performing writes to an object at any time [26]. Since in SCSTM the entire write set is protected by a single counter, such an approach does not make sense for large write sets. Instead, we use a hash-based write log [20] to ensure that every address in the log is unique. We also require that no two addresses in the log overlap. This property is provided differently for type-safe and non-type-safe languages.

With these requirements in place, the writeback operation can be done in any order, and any number of {address, value} pairs in the write log can be written back multiple times, so long as (a) every pair is written back at least once and (b) as soon as one thread increments `global_ts` to even (`TMEnd` line 17), all helping immediately stops. If all writeback were performed using BETM speculative stores, this second property would be simple. However, with "requester wins" conflict detection, using BETM for writeback would rapidly lead to livelock. Instead, we use BETM to monitor to the `global_ts` and immediately interrupt a writeback, and then perform the writeback using nonspeculative stores.

Figure 2 presents the lock-free writeback code, assuming that when odd, `global_ts` represents a pointer to all needed metadata. To make SCSTM nonblocking, we replace the `spin()` codes (`TMBegin` line 4; line 5 of `TMValidate`) with calls to `TMStealWriteBack`, and replace `TMEnd` lines 14–15 with a call to `TMStealWriteBack`. The `HW_LIMIT` field addresses capacity constraints, and is discussed in the next section.

The use of `TMStealWriteBack` preserves privatization safety, livelock-freedom, and correctness. The argument is straightforward: in SCSTM, these properties are all preserved by the following pair of invariants: *an in-flight transaction never*

```
TMStealWriteBack:
 1  // start BETM transaction
 2  BETM_BEGIN
 3  // if global_ts changes, this
 4  // thread returns to line 2
 5  BETM_LOAD(x, global_ts)
 6  // writeback may be finished
 7  if ((x & 1) == 0)
 8    BETM_COMMIT
 9    return
10  // global_ts is a pointer
11  metadata = x & ~1
12  // metadata->done_already estimates
13  // how much writeback is complete
14  c = metadata->done_already
15  // do writeback, starting at c
16  for <a, v> in metadata->writeset[c..end]
17    *a = v
18    if ((++c % HW_LIMIT) == 0)
19      metadata->done_already = c
20  // compute new global_ts value
21  n = metadata->next_ts
22  BETM_COMMIT
23  // nontransactionally update global_ts
24  // with CAS.  must use counted pointer
25  CAS(global_ts, x, n + 2)
```

Fig. 2. Lock-free writeback

performs a read of shared data while write-back is in progress and *the commit of writing transactions is serialized*. In SCSTM, these invariants are preserved via blocking. In our constructions, the invariants still hold, but the blocking is replaced by assisting with write-back.

4.2 Ensuring Progress via HW_LIMIT

In Figure 2, the entire writeback operation is performed using nonspeculative stores issued from within a transaction. By using nonspeculative stores, performing writeback simultaneously in multiple threads will not prevent progress. However, BETM transactions have limits that must be managed. Specifically, they can access a limited amount of memory and they cannot survive exceptions or interrupts. Our approach is to break the writeback into small pieces, and then use the `done_already` field to track how much of the writeback is complete. The `HW_LIMIT` field dictates how many writes are performed before updating `done_already`. We discuss constraints on its size below:

First, since BETM transactions cannot execute for longer than a quantum, we must not attempt more writes than can be reasonably finished in that time. Of greater practical concern, in virtualized environments with high I/O interrupt frequencies, `HW_LIMIT` must be small enough that a bundle of writes can be completed between interrupts.

The next danger occurs when the speculative region performs an instruction that causes an abort; if there is no mechanism to prevent the offending instruction from re-occurring when the region is restarted, then an infinite loop can occur. In a BETM such as Rock, where page faults and some TLB misses can cause aborts that must be handled by the user code, an abort handler will be required to update the TLB or simulate a page fault in nontransactional code. In ASF, where a page-fault will be handled automatically before the region restarts, this is not a concern. Additionally, in both systems the speculative region must not perform prohibited instructions, such as division, far calls, system calls, and some other function calls. This condition clearly holds in our code, which only performs loads, stores, and read-only traversal of a data structure.

Lastly, BETM may limit nonspeculative writes within a transaction. On Rock, stores may be issued to at most 32 locations from within a hardware transaction. Thus if the entries in the `writeset` map to more than 31 locations (we reserve one for `done_already`), the speculative region is guaranteed to abort. In addition to the store buffer, the TLB introduces limitations. Suppose that a processor has T entries in its TLB. If each variable in the `writeset` is located on a different page of memory, then if the TLB capacity is fewer than 31 entries, a tighter bound is required if TLB misses can cause aborts. In the worst case, if the instruction and data caches share a TLB, and every interesting memory element with a size greater than a single word happens to span a page boundary, the following restrictions would apply (note that we assume the page size is larger than the word size):

- Two TLB entries must be reserved for the code pages holding the `TMStealWriteBack` function.
- Two TLB entries must be reserved for the data pages holding the stack.
- The data pages holding `writeset` portion being written-back require two TLB entries.
- One TLB entry must be reserved for the data page holding the metadata object.

Thus if we attempt to write to more than $T-7$ pages, the region may abort due to a TLB miss, and repeated attempts to execute the region will continue to abort due to TLB misses (caused by writes to different pages). If we generalize the size of the store buffer to a constant `MAX_STORES`, then the total number of stores that can be guaranteed to succeed is `HW_LIMIT = min(T-7, MAX_STORES)`. In practice, however, if multiple updates are performed to the same page and/or cache line, this limit may be exceeded without aborting the region.[1]

To overcome this limitation without being overly conservative, we use a lazy counter, `done_already`. When a transaction logically commits (the CAS on line 9 of `TMEnd` succeeds), it installs a pointer that indicates both the `writeset` and a count of the number of `writeset` entries that have already been written back. `TMStealWriteBack` updates this count with its local progress after every `HW_LIMIT` stores. If a thread is executing `TMStealWriteBack` in isolation and its writeback aborts due to a hardware limitation, when the region restarts it resumes writeback at the last value of `done_already`. This ensures progress and avoids infinite loops, since at least `HW_LIMIT` stores were performed before the abort, and will not be performed again.

If `TMStealWriteBack` is executed by multiple threads, it is possible for the value of `done_already` to decrease. However, we observe that once all threads observe the value v in `done_already`, then the first v elements will not be written again, and that `done_already` will only hold values greater than v from that point until one thread completes writeback. Thus the writeback will ultimately succeed, though the worst-case bound completion time depends both on the

[1] These assumptions are conservative. Modern machines have large pages and separate instruction/data TLBs. Explicit alignment of data structures can decrease the number of reserved TLB entries from 7 to as low as 2.

number of threads (T) and the size of the `writeset` (W). This bound could be improved by making the value of c persist from one writeback attempt to the next, and updating it to `max(c, metadata->done_already)` within each attempt. Doing so would require additional common-case code to ensure that the value of x has not changed; for example, if thread 1 performs K writes, then delays and another thread both completes the writeback and installs a new `writeset` to be written back, then thread 1 cannot resume at the Kth element.

There are two other properties that our approach provides. First, existing limits on `HW_LIMIT` ensure that context switches during writeback will not force the entire writeback to restart; this is particularly important if writeback of an extremely large dataset could take longer than a quantum to complete. Second, the writeback could be parallelized by partitioning the `writeset` and protecting each partition with a different lock [9]. As long as a thread's call to `TMStealWriteBack` does not return until *all* partitions of the writeback have been performed, the behavior is indistinguishable from that of a single-lock writeback, but with the added benefit of parallel writeback. In the limit case, we could set the partition size to `HW_LIMIT`, and simplify the code significantly. We do not evaluate such an implementation in this paper.

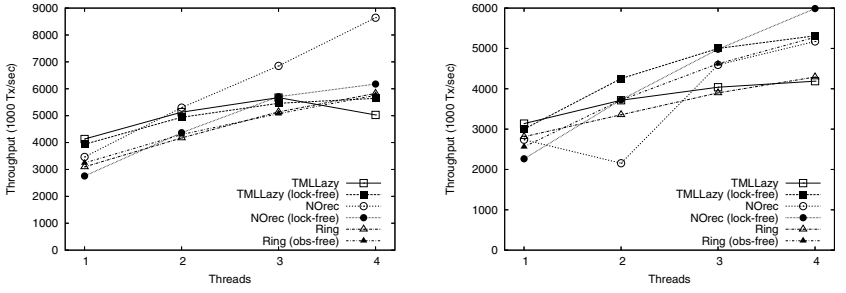
4.3 ABA Safety

The code in Figure 2 can admit an "ABA" problem on line 25. Specifically, when `global_ts` equals some value V and two threads T_1 and T_2 both execute `TMStealWriteBack`, it is possible that T_1 pauses between lines 22 and 25. It has done *all* of the writeback, but it has not halted writeback in any helper threads (such as T_2). If T_2 then also performs the writeback, executes line 25, and then commits a new transaction that re-uses the same location as its "metadata" object, then when T_1 awakes and issues its CAS, it can "clean up" from an in-flight writeback that it has mistaken for the writeback it previously performed. We avoid this problem by attaching a counter to the `global_ts` pointer and using a wide CAS. Techniques using garbage collection are equally applicable. However, the use of a hardware transaction does not would only provide obstruction-freedom.

Since RingSTM requires a multiword atomic update in order to commit, we are already limited to producing an obstruction-free version since a BETM-based nonblocking multiword update is only obstruction-free. That being the case, when making RingSTM nonblocking we make the writeback obstruction-free, by removing lines 23–25 and inserting the statement `BETM_STORE(global_ts, n+2)` before line 22.

5 Evaluation

We evaluate our algorithms using the ASF extensions to PTLSim [29]. PTLSim faithfully models the AMD64 instruction set architecture, with extensions to support ASF. Previous studies have found PTLSim to be sufficiently accurate when compared to real hardware [2].



(a) RBTree (no blocking) (b) RBTree (with blocking)

Fig. 3. RBTree workloads with 8-bit keys and an equal insert/lookup/remove ratio

For this work, we made TMLLazy and NOrec lock-free. Since we use a writing hardware transaction to achieve atomic multiword updates in RingSTM, it is only obstruction-free. Our codes do not use the parallel writeback optimization discussed in Section 4.2, and our algorithms use a hard-coded value of 16 for `HW_LIMIT`. Our RingSTM, NOrec, and TMLLazy codes are based on their published implementations, which are more optimized than the pseudocode in Figure 1.

We first evaluate the cost of our nonblocking algorithms in comparison to their nonblocking counterparts. Figure 3(a) depicts a Red-Black Tree experiment, using 8-bit keys. In each trial, we perform 100K transactions per thread, using an equal mix of inserts, lookups, and deletes. The maximum number of locations written by a single transaction is 33. Our first finding, which we do not show, is that it is extremely easy to livelock hardware transactions that perform writes. In Ring (obs-free), we used a writing hardware transaction to commit transactions. Initially we had each thread execute a fixed no-op loop whenever its hardware transaction aborted. At three threads this lead to two orders of magnitude slowdown; the hardware transaction to effect a commit would abort 98K times per 67 transactions. To avoid this behavior, we added exponential backoff. Note that backoff was never needed for stealable writeback.

Regarding performance, we see that the cost of lock freedom in TMLLazy is negligible, as is the cost of obstruction freedom in RingSTM. For NOrec, the cost of lock freedom seems higher. We attribute this to two causes: first, we note that NOrec's scalability on the simulator is much better than previously reported for this workload [3, 23]. Secondly, we observed that NOrec steals writeback much more frequently than the other systems. In Ring, transactions validate before helping, and in TMLLazy they abort and roll back before helping; in NOrec they must help with writeback before validating, since validation uses the actual values instead of the value of `global_ts` (TMLLazy) or signature (RingSTM). We conjecture that making stealing less aggressive, by introducing a small delay before attempting to steal writeback, would mitigate this effect.

Secondly, we consider the impact of preemption. For these tests, we introduced a delay between lines 12 and 13 of `TMEnd`. The delay occurs on randomly selected

writing transactions (roughly once per 32 commits), and consists of a loop executing 8K no-ops. This simulates the effect of preemption during the write-back phase. In our blocking implementations, this delay prevents any progress in any other thread, since the committing thread holds the lock; we expect dampened scalability. For nonblocking implementations, a concurrent transaction should not block when the committer experiences a delay; instead, the concurrent transaction should steal the writeback and then continue.

Figure 3(b) depicts the behavior for our algorithms with and without the nonblocking transformation. The fact of delays in the commit protocol immediately results in lower single-thread throughput, and the artificial blocking also substantially limits the scalability of the lock-based STM implementations. By stealing writeback, our nonblocking algorithms overcome fact that some transactions block while holding locks. This, in turn, leads to higher throughput at all thread levels for the nonblocking code, when compared to its lock-based counterpart.

6 Conclusions and Future Work

In this paper we considered the construction of lock-free STM, and showed how to make single-CAS STM obstruction-free (and in some cases lock-free) through the use of simple best-effort transactional hardware (BETM). Our results indicate that the cost of nonblocking progress guarantees is negligible, and the nonblocking systems are immune to pathologies such as preemption and livelock. Our algorithms provide the first scalable, lock-free TM implementations with strong semantics.

We hope that this research will encourage further consideration of how much must be guaranteed to make BETM useful. In our algorithm, transactions executing only one speculative load, no speculative stores, and 16 nonspeculative stores, were sufficient to build a privatization-safe lock-free STM. Looking forward, questions include:

1. In virtualized or I/O-intensive environments, will parameters like `HW_LIMIT` need to be dynamically adjusted?
2. Do aborted BETM transactions know immediately that they have aborted, or can they continue to execute nonspeculative stores for any duration after their abort? What impact would such "zombie" transactions or delayed stores have on our algorithms?
3. What unspecified sources of aborts can be overcome in practice? Are there any undocumented sources of aborts (we imagine branch misprediction is a possibility) that would break our progress guarantees?

Additionally, we hope that successive refinements of BETM will improve ease-of-use and performance. Clearly support for both speculative and nontransactional accesses is critical to progress. In our opinion, a platform in which very few locations can be accessed, but transactions are guaranteed to succeed if they are retried enough times in succession, is very appealing.

Acknowledgments. We thank our reviewers for many helpful suggestions, and Victor Luchangco for his advice during the preparation of our final manuscript. Stephan Diestelhorst, Dave Dice, and Hagit Attiya gave feedback on early drafts of this paper. We are also grateful to Martin Pohlack and Stephan Diestelhorst for their continued assistance with the ASF toolchain, and Dave Dice for his help understanding the Rock processor. We also thank Michael Scott, Luke Dalessandro, and Torvald Riegel.

References

1. Bobba, J., Moore, K., Volos, H., Yen, L., Hill, M., Swift, M., Wood, D.: Performance Pathologies in Hardware Transactional Memory. In: Proc. of the 34th Intl. Symp. on Computer Architecture, San Diego, CA (June 2007)
2. Christie, D., Chung, J.W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Riviere, E.: Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In: Proc. of the EuroSys2010 Conf., Paris, France (April 2010)
3. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Bangalore, India (January 2010)
4. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proc. of the 20th Intl. Symp. on Distributed Computing, Stockholm, Sweden (September 2006)
5. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early Experience with a Commercial Hardware Transactional Memory Implementation. In: Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, Washington, DC (March 2009)
6. Dice, D., Shavit, N.: TLRW: Return of the Read-Write Lock. In: Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing, Raleigh, NC (February 2009)
7. Diestelhorst, S., Hohmuth, M.: Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In: Proc. of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods, Boston, MA (April 2008)
8. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT (February 2008)
9. Fernandes, S., Cachopo, J.: A Scalable and Efficient Commit Algorithm for the JVSTM. In: Proc. of the 5th ACM SIGPLAN Workshop on Transactional Computing, Paris, France (April 2010)
10. Fraser, K., Harris, T.: Concurrent Programming Without Locks. ACM Trans. on Computer Systems 25(2) (2007)
11. Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N.: Software Transactional Memory for Dynamic-sized Data Structures. In: Proc. of the 22nd ACM Symp. on Principles of Distributed Computing, Boston, MA (July 2003)
12. Herlihy, M.P., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Proc. of the 20th Intl. Symp. on Computer Architecture, San Diego, CA (May 1993)

13. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Trans. on Prog. Languages and Systems 12(3), 463–492 (1990)
14. Hudson, R.L., Saha, B., Adl-Tabatabai, A.R., Hertzberg, B.: A Scalable Transactional Memory Allocator. In: Proc. of the 2006 Intl. Symp. on Memory Management, Ottawa, ON, Canada (June 2006)
15. Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a Scalable Software Transactional Memory. In: Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing, Raleigh, NC (February 2009)
16. Marathe, V., Moir, M.: Toward High Performance Nonblocking Software Transactional Memory. In: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT (February 2008)
17. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: Proc. of the 37th Intl. Conf. on Parallel Processing, Portland, OR (September 2008)
18. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (June 2008)
19. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and Implementation of Transactional Constructs for C/C++. In: Proc. of the 23rd ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications, Nashville, TN, USA (October 2008)
20. Olszewski, M., Cutler, J., Steffan, J.G.: JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In: Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques, Brasov, Romania (September 2007)
21. Shavit, N., Touitou, D.: Software Transactional Memory. In: Proc. of the 14th ACM Symp. on Principles of Distributed Computing, Ottawa, ON, Canada (August 1995)
22. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K., Saha, B.: Enforcing Isolation and Ordering in STM. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation, San Diego, CA (June 2007)
23. Spear, M.: Lightweight, Robust Adaptivity for Software Transactional Memory. In: Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures, Santorini, Greece (June 2010)
24. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: Proc. of the 12th Intl. Conf. On Principles Of DIstributed Systems, Luxor, Egypt (December 2008)
25. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable Transactions with a Single Atomic Instruction. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (June 2008)
26. Spear, M.F., Shriraman, A., Dalessandro, L., Dwarkadas, S., Scott, M.L.: Nonblocking Transactions Without Indirection Using Alert-on-Update. In: Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures, San Diego, CA (June 2007)

27. Tabba, F., Wang, C., Goodman, J.R., Moir, M.: NZTM: Nonblocking Zero-Indirection Transactional Memory. In: Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR (August 2007)
28. 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: Proc. of the 2007 Intl. Symp. on Code Generation and Optimization, San Jose, CA (March 2007)
29. Yourst, M.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: Proc. of the 2007 IEEE Intl. Symp. on Performance Analysis of Systems and Software, San Jose, CA (April 2007)

Window-Based Greedy Contention Management for Transactional Memory

Gokarna Sharma[1], Brett Estrade[2], and Costas Busch[1]

[1] Department of Computer Science, Louisiana State University,
Baton Rouge, LA 70803, USA
{gokarna,busch}@csc.lsu.edu
[2] Department of Computer Science, University of Houston,
501 Philip G. Hoffman Hall, Houston, TX 77204, USA
estrabd@cs.uh.edu

Abstract. We consider greedy contention managers for transactional memory for $M \times N$ *execution windows of transactions* with M threads and N transactions per thread. Assuming that each transaction has duration τ and conflicts with at most C other transactions inside the window, a trivial greedy contention manager can schedule them within τCN time. In this paper, we explore the theoretical performance boundaries of this approach from the worst-case perspective. Particularly, we present and analyze two new randomized greedy contention management algorithms. The first algorithm Offline-Greedy produces a schedule of length $O(\tau \cdot (C + N \log(MN)))$ with high probability, and gives competitive ratio $O(\log(MN))$ for $C \leq N \log(MN)$. The offline algorithm depends on knowing the conflict graph which evolves while the execution of the transactions progresses. The second algorithm Online-Greedy produces a schedule of length $O(\tau \cdot (C \log(MN) + N \log^2(MN)))$, with high probability, which is only a $O(\log(NM))$ factor worse, but does not require knowledge of the conflict graph. Both of the algorithms exhibit competitive ratio very close to $O(s)$, where s is the number of shared resources. Our algorithms provide new tradeoffs for greedy transaction scheduling that parameterize window sizes and transaction conflicts within the window.

1 Introduction

Multi-core architectures present both an opportunity and challenge for multi-threaded software. The opportunity is that threads will be available to an unprecedented degree, and the challenge is that more programmers will be exposed to concurrency related synchronization problems that until now were of concern only to a selected few. Writing concurrent programs is difficult because of the complexity of ensuring proper synchronization. Conventional lock based synchronization suffers from well known limitations, so researchers considered non-blocking transactions as an alternative. Software Transactional Memory (STM) [22,13,14] systems use lightweight and composable in-memory software transactions to address concurrency in multi-threaded systems ensuring safety all the time [10,11].

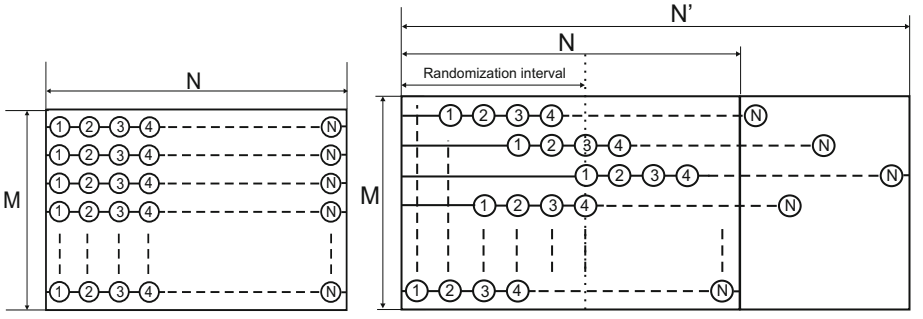
N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 64–78, 2010.

In transactional memory (TM) systems, a *contention management* strategy is responsible for the system as a whole to make progress [13,18]. If transaction T discovers that it conflicts with another transaction T' (because they share a resource), it has two choices, it can give T' a chance to finish by aborting itself, or it can proceed by forcing T' to abort; the aborted transaction then retries again until it eventually commits. To solve this scheduling problem efficiently, T will consult the *contention manager* module for which choice to make. Of particular interest are *greedy contention managers* where a transaction starts again immediately after every abort. Several (greedy) contention managers have been proposed in the literature [3,9,7,5,21]. However, most contention managers have been assessed only experimentally by specific benchmarks [18]. There is a small amount of work in the literature which analyzes formally the performance of contention managers [3,9,7,21].

The competitive ratio results are not encouraging. For example in [3] the authors give an $O(s)$ competitive ratio bound, where s is the number of resources. When the number of resources s increases, the performance degrades linearly. A difficulty in obtaining tight bounds is that the algorithms studied in [3,9,7,5,21] apply to the *one-shot scheduling problem*, where each thread issues a single transaction. One-shot problems are directly related with vertex coloring, where the problem of determining the chromatic number of a graph is reduced to finding an optimal time schedule for the one-shot problem. Since it is known that computing an optimal coloring given complete knowledge of the graph is a very hard problem to approximate, the one-shot problem is very hard to approximate too [15].

In order to obtain alternative and improved formal bounds, we propose to investigate the performance of program executions in *windows of transactions* (see Fig. 1a), which has the potential to overcome some of the limitations of the coloring reduction in certain circumstances. A $M \times N$ *window* W consists of M threads with an execution sequence of N different transactions per thread. The execution window W can be viewed as a collection of N one-shot transaction sets with M concurrent transactions in each set. Let C denote the maximum number of conflicting transactions for any transaction in the window. If we assume that all transactions have the same duration τ, then a straightforward upper bound for the makespan of the window is $\tau \cdot \min(CN, MN)$, since τCN follows from the observation that each transaction in a thread may be delayed at most C times by its conflicting transactions, and τMN follows from the serialization of the transactions. The competitive ratio of the makespan using the one-shot analysis results is bounded by $O(sN)$. Using the one-shot Algorithm RandomizedRounds provided in [21] N times, the completion time is in the worst case $O(\tau CN \log M)$.

In this paper, we show that we can obtain better results by utilizing the window representation. We present a family of randomized greedy algorithms where transactions are assigned priorities values, such that for some random initial interval in the beginning of the window W each transaction is in low priority mode and then after the random period expires the transactions switch to high priority mode. In high priority mode the transaction can only be aborted by other



(a) Before execution **(b)** After execution

Fig. 1. Execution window model for transactional memory

high priority transactions. The random initial delays have the property that the conflicting transactions are shifted inside their window and their execution times may not coincide (see Fig. 1b). The benefit is that conflicting transactions can execute at different time slots and potentially many conflicts are avoided. The benefits become more apparent in scenarios where the conflicts are more frequent inside the same column transactions and less frequent between different column transactions.

Contributions: We propose the contention measure C within the window to allow more precise statements about the worst-case complexity bound of any contention management algorithm. We give two window-based randomized greedy algorithms for the contention management in any execution window W. For simplicity, we assume that each transaction has the same duration τ (this assumption can be removed). Our first algorithm Offline-Greedy gives a schedule of length $O(\tau \cdot (C + N \log(MN)))$ with high probability, and improves on one-shot contention managers from a worst-case perspective. The algorithm is offline in the sense that it uses explicitly the dynamic conflict graph of the transactions at each time step of execution to resolve the conflicts. Our second algorithm Online-Greedy produces a schedule of length $O(\tau \cdot (C \log(MN) + N \log^2(MN)))$ with high probability, which is only a factor of $O(\log(MN))$ worse in comparison to Offline-Greedy. The benefit of the online algorithm is that does not need to know the conflict graph of the transactions to resolve the conflicts. The online algorithm uses as a subroutine a variation of algorithm RandomizedRounds [21].

We also give a third algorithm Adaptive-Greedy which is the adaptive version of the previous algorithms which achieves similar worst-case performance and adaptively guesses the value of the contention measure C. The technique we use for the analysis of these algorithms is similar to the one used by Leighton *et al.* [16] to analyze an online packet scheduling problem.

An advantage of our algorithms is that if the conflicts in the window are bounded by $C \leq N \log(MN)$ then the upper bounds we have obtained are within poly-logarithmic factors from optimal, since N is a trivial lower bound

in execution time. This is an improvement over the trivial approach of using N one-shot executions. We also show that the offline algorithm is $O(s+\log(MN))$-competitive and the online algorithm is $O(s\cdot\log(MN)+\log^2(MN))$-competitive (for any choice of C).

Outline of Paper: The rest of the paper is organized as follows: In Section 2, we discuss the related work. We present the transactional memory model in Section 3. We present and formally analyze an offline randomized greedy algorithm in Section 4. The online version of the randomized greedy algorithm is given in Section 5. In Section 6, we describe the adaptive version of the aforementioned algorithms. Section 7 concludes the paper.

2 Related Work

Transactional Memory (TM) has been proposed in the early nineties as an alternative implementation of mutual exclusion that avoids many of the drawbacks of locks e.g., deadlock, reliance on the programmer to associate shared data with locks, priority inversion, and failures of threads while holding locks [14]. In 2003, Dynamic STM (DSTM) [13] was proposed for dynamic-sized data structures which uses a contention manager module as an independent module to resolve conflicts between two transactions and ensure progress. DSTM is a practical obstruction-free[1] STM system that seeks advice from the contention manager module to either wait or abort an transaction at the time of conflict.

As TM has been gaining attention, several contention managers are available in the literature [3,9,7,5,21]. Most of them have been assessed by specific benchmarks only and not analytically. A comparison of contention managers based on different benchmarks can be found in [18,19,17,20,7,8]. They observed that the choice of the best contention manager varies with the complexity of the considered benchmark. The more detailed analysis of the performance of different contention managers in complex benchmarks has recently been studied by Ansari *et al.* [1]. From all the aforementioned references, one can notice that the coordination cost and the overhead involved in contention management is very high.

The first formal analysis of the performance of a contention manager was given by Guerraoui *et al.* [9] by presenting the Greedy contention manager which decides in favor of old transactions using timestamps and proving that it achieves $O(s^2)$ competitive ratio in comparison to the optimal off-line schedulers for M concurrent transactions that share s objects. They argue that the bound holds for any algorithm which ensure that at any point in time at least one transaction is running uninterrupted until it commits, which is called the *pending commit* property. Later, Guerraoui *et al.* [7] studied the impact of transaction failures on contention management. They presented the algorithm FTGreedy and proved the $O(ks^2)$ competitive ratio when some running transaction may fail at most k

[1] A synchronization mechanism is obstruction-free if any thread runs itself for a long time makes progress [12].

times and then eventually commits. Attiya *et al.* [3] improved the result of [9] to $O(s)$, and the result of [7] to $O(ks)$, which are significant improvements over the competitive ratio of Greedy. They also proved the matching lower bound of $\Omega(s)$ for the competitive ratio for deterministic *work-conserving* algorithms which schedule as many transactions as possible (by choosing a maximal independent set of transactions).

Recently, Schneider and Wattenhofer [21] presented a deterministic algorithm CommitBounds with competitive ratio $\Theta(s)$ for M concurrent transactions using s shared resources and a randomized algorithm RandomizedRounds with makespan $O(C \log M)$, for the one-shot problem of a set of M transactions in separate threads with C conflicts (assuming unit delays for transactions). While previous studies showed that contention managers Polka [18] and SizeMatters [17] exhibit good overall performance for variety of benchmarks, this work showed that they may perform exponentially worse than RandomizedRounds from a worst-case perspective. Another recent proposal for the contention management is Serializer [5], which resolves a conflict by removing a conflicting transaction T from the processor core where it was running, and scheduling it on the processor core of the other transaction to which it conflicted with. It is $O(M)$-competitive and in fact, it ensures that two transactions never conflict more than once.

On the other side, *TM schedulers* [6,23,2,4] offer an alternative approach to boost the TM performance. TM scheduler is, basically, a software component which decides when a particular transaction executes. One proposal in this approach is Adaptive Transaction Scheduling (ATS)[2] [23] scheduler. Restart [6] scheduler is another recent proposal whose worst case performance is very sensitive to the accuracy of the prediction of the future conflicts. In [6] they also propose a scheduler called Shrink which predicts the future accesses of a based on the past accesses, and dynamically serializes transactions based on the prediction to prevent conflicts. The ATS, Restart and Shrink schedulers are $O(M)$-competitive in the worst-case. Steal-On-Abort [2] is the yet another proposal where the aborted transaction is given to the opponent transaction and queued behind it, preventing the two transactions from conflicting again. Recently, Attiya *et al.* [4] proposed the BIMODAL scheduler which alternates between *writing epochs* where it gives priority to writing transactions and *reading epochs* where it gives priority to transactions that have issued only reads so far. It achieves $O(s)$ competitive ratio on bimodal[3] workloads with equi-length transactions.

3 Execution Window Model

Consider M threads $\mathcal{P} = \{P_1, \cdots, P_M\}$. We consider a model that is based on a $M \times N$ execution window W consisting of a set of transactions $\mathcal{T}(W) = \{(T_{11}, \cdots, T_{1N}),\ (T_{21}, \cdots, T_{2N}),\ \ldots, (T_{M1}, \cdots, T_{MN})\}$, where each thread P_i

[2] ATS measures adaptively the contention intensity of a thread, when the contention intensity increases beyond a threshold, it serializes the transactions.

[3] A workload containing only early-write and read-only transactions (see [4] for details).

issues N transactions $T_{i1}, \cdots, T_{iN}$ in sequence, so that T_{ij} is issued as soon as $T_{i(j-1)}$ has committed. If $N = 1$ then this is similar to the one-shot TM model, that uses one transaction per thread.

Transactions share a set of s shared resources $\mathcal{R} = \{R_1, \ldots, R_s\}$. Each transaction is a sequence of actions that is either a read or write to some shared resource R. A transaction after it is issued it either commits or aborts. A transaction that has been issued but not committed is said to be pending. Concurrent write-write actions or read-write actions to shared objects by two or more transactions cause conflicts between transactions. If a transactions conflicts then it either aborts, or it may commit and force to abort all other conflicting transactions. In a *greedy schedule*, if a transactions aborts it then immediately restarts and attempts to commit again.

The *makespan* of a schedule for the transactions is defined as the duration from the start of the schedule, i.e., the time when the first transaction is issued, until all transactions have committed. The makespan of the transaction scheduling algorithm for the sequences of transactions can be compared to the makespan of an optimal off-line scheduling algorithm, denoted $makespan_{\text{opt}}$ to provide a competitive ratio. Each transaction T_{ij} has execution time duration τ_{ij} which is greater than 0. When we describe our algorithms we assume that all transactions have the same duration $\tau = \tau_{ij}$. We also assume that the execution time advances synchronously for all threads, where each time step corresponds to a period of duration τ. We also assume that and all transactions inside the execution window are correct, i.e., there are no faulty transactions[4]. Our results can be extended by relaxing these assumptions. In Section 7, we describe the impact that variable time durations for the transactions have to the performance of our algorithms.

3.1 Conflict Graph

Consider a set of k transactions $\mathcal{T} = \{T_1, \ldots, T_k\}$. Let $\mathcal{R}(T_i)$ denote the set of resources used by transaction T_i. We can write $\mathcal{R}(T_i) = \mathcal{R}_w(T_i) \cup \mathcal{R}_r(T_i)$, where $\mathcal{R}_w(T_i)$ are the resources which are to be written by T_i and $\mathcal{R}_r(T_i)$ are the resources to be read by T_i.

Definition 1 (Transaction Conflict). *Two transactions T_i and T_j* conflict *if at least one of them writes on a common resource, that is, there is a resource R such that $R \in (\mathcal{R}_w(T_i) \cap \mathcal{R}(T_j)) \cup (\mathcal{R}(T_i) \cap \mathcal{R}_w(T_j))$ (we also say that R causes the conflict).*

Definition 2 (Conflict Graph). *The* conflict graph $G = (\mathcal{T}, E)$ *has nodes the transactions, and $(T_i, T_j) \in E$ for any two transactions T_i, T_j that conflict.*

Let $\delta(T_i)$ denote the degree of node T_i in G. We denote $C = \max_i \delta(T_i)$. Let $\gamma(R_j)$ denote the number of transactions that write R_j. Let $\gamma_{\max} = \max_j \gamma(R_j)$.

[4] A transaction is called faulty when it encounters an illegal instruction producing a segmentation fault or experiences a page fault resulting to wait for a long time for the page to be available [7].

Let $\lambda(T_i) = |\{R : R \in \mathcal{R}(T_i) \wedge (\gamma(R) \geq 1)\}|$, denote the number of resources that can be the cause of conflicts to transaction T_i. Let $\lambda_{\max} = \max_i \lambda(T_i)$. Note that $C \leq \lambda_{\max} \cdot \gamma_{\max}$, and, $C \geq \gamma_{\max}$.

Assuming that there is one transaction per thread, the conflict graph can be used to obtain a simple greedy schedule of the transactions as follows. Compute a $C+1$ vertex coloring of the conflict graph. All transactions of same color can commit simultaneously. The transactions can be scheduled in a greedy manner by giving a different priority to each transaction color. This produces a greedy schedule of length $Makespan = \tau \cdot (C+1)$. Since $C \leq \lambda_{\max} \cdot \gamma_{\max}$, we have that $makespan \leq \tau \cdot (\lambda_{\max} \cdot \gamma_{\max} + 1)$. Further, since $C \geq \gamma_{\max}$, $makespan_{\text{opt}} \geq \tau \cdot \gamma_{\max}$. Since $\lambda_{\max} \leq s$, the competitive ratio of the schedule is $\lambda_{\max} + 1 = O(s)$.

4 Offline Algorithm

We present the Algorithm Offline-Greedy (Algorithm 1), which is an offline greedy contention resolution algorithm that it uses the conflict graph explicitly to resolve conflicts of transactions. In addition to M and N, we assume that each thread P_i knows C_i, which denotes the maximum number of transactions that any transaction in P_i conflicts with; namely, using the conflict graph $G(\mathcal{T}(W))$, $C_i = \max_j \delta(T_{ij})$. Note that $C = \max_i C_i$.

Time is measured in discrete time steps, where each time step represents the duration τ of the transactions. We divide time into *frames*, which are time periods of duration $\Theta(\tau \cdot \ln(MN))$ (namely, each frame consists of $\Phi = \Theta(\ln(MN))$ time steps). Then, each thread P_i is assigned an initial random time period consisting of q_i frames, where q_i is chosen randomly, independently and uniformly, from the range $[0, \alpha_i - 1]$, where $\alpha_i = C_i / \ln(MN)$. Each transaction has two priorities: *low* or *high*. Transaction T_{ij} is initially in low priority. Transaction T_{ij} switches to high priority in the first time step of frame $F_{ij} = q_i + (j-1)$ (this is the assigned frame for T_{ij}) and remains in high priority thereafter until it commits. In the analysis, we show that with high probability each transaction commits in its assigned frame.

The priorities are used to resolve conflicts. A high priority transaction may only be aborted by another high priority transaction. A low priority transaction is always aborted if it conflicts with a high priority transaction. Let G_t denote the conflict graph of transactions at time step t which evolves while the execution of the transactions progresses. Note that the maximum degree of G_t is bounded by C, but the effective degree between high priority transactions is lower. At each time step t we select to commit a maximal independent set of transactions in G_t. We first select a maximal independent set I_H of high priority transactions, then remove this set and its neighbors from G_t, and then select a maximal independent set I_L of low priority transactions from the remaining conflict graph. The transactions that commit are $I_H \cup I_L$.

The intuition behind the algorithm is as follows. Consider a thread i and its first transaction in the window T_{i1}. According to the algorithm, T_{i1} becomes high priority in the beginning of frame F_{i1}. Because q_i is chosen at random

Algorithm 1. Offline-Greedy

Input: A $M \times N$ window W of transactions with M threads each with N transactions; Each thread P_i knows C_i, the maximum number of transactions in W that any transaction in P_i conflicts with; Each transaction has same duration τ;

Divide time into frames consisting of $\Phi = 1 + (e^2 + 2)\ln(MN)$ time steps;
Each thread P_i chooses a random number $q_i \in [0, \alpha_i - 1]$ for $\alpha_i = C_i/\ln(MN)$;
Each transaction T_{ij} is assigned to frame $F_{ij} = q_i + (j-1)$;
foreach *time step* $t = 0,\ 1\tau,\ 2\tau,\ 3\tau, \ldots$ **do**
  Phase 1: Priority Assignment
  foreach *transaction* T_{ij} **do**
    if $t < F_{ij} \cdot \tau\Phi$ **then** $Priority(T_{ij}) \leftarrow 1$ (*low*); **else**
    $Priority(T_{ij}) \leftarrow 0$ (*high*);
  Phase 2: Conflict Resolution
  begin
    Let G_t be the conflict graph at time t;
    Compute G_t^H and G_t^L, the subgraphs of G_t induced by high and low priority nodes, respectively;
    Compute $I_H \leftarrow I(G_t^H)$, maximal independent set of nodes in graph G_t^H;
    $Q \leftarrow$ low priority nodes adjacent to nodes in I_H;
    Compute $I_L = I(G_t^L \setminus Q)$, maximal independent set of nodes in graph G_t^L after removing Q nodes;
    Commit $I_H \cup I_L$;

among $C_i/\ln(MN)$ positions it is expected that T_{i1} will conflict with at most $O(\ln(MN))$ transactions in its assigned frame F_{i1} which become simultaneously high priority in F_{i1}. Since a time frame contains $\Phi = \Theta(\ln(MN))$ time steps, transaction T_{i1} and all its high priority conflicting transactions will be able to commit by the end of time frame F_{i1}, using the conflict resolution graph. The initial randomization period of $q_i \cdot \Phi$ frames will have the same effect to the remaining transactions of the thread i, which will also commit within their assigned frames.

4.1 Analysis of Offline Algorithm

We study the makespan of Algorithm Offline-Greedy. According to the algorithm, when a transaction T_{ij} is issued, it will be in low priority until the respective frame F_{ij} starts. As soon as F_{ij} starts, the transaction T_{ij} will begin executing in high priority (if it didn't commit already). Let A denote the set of conflicting transactions with T_{ij} in the conflict graph $G(\mathcal{T}(W))$. Let $A' \subseteq A$ denote the subset of conflicting transactions with T_{ij} which become high priority during frame F_{ij} (simultaneously with T_{ij}).

Lemma 1. *If* $|A'| \leq \Phi - 1$ *then transaction* T_{ij} *will commit in frame* F_{ij}.

Proof. Due to the use of the high priority independent sets in the conflict graph G_t, if in time t during frame F_{ij} transaction T_{ij} does not commit, then some conflicting transaction in A' must commit. Since there are at most $\Phi - 1$ high priority conflicting transactions, and the length of the frame F_{ij} is at most Φ, T_{ij} will commit by the end of frame F_{ij}.

We show next that it is unlikely that $|A'| > \Phi - 1$. We use the following Chernoff bound:

Lemma 2 (Chernoff Bound 1). *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}(X_i = 1) = pr_i$, where $0 < pr_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} pr_i$, and any $\delta > e^2$, $\mathbf{Pr}(X > \delta\mu) < e^{-\delta\mu}$.*

Lemma 3. *$|A'| > \Phi - 1$ with probability at most $(1/MN)^2$.*

Proof. Let $A_k \subseteq A$, where $1 \leq k \leq M$, denote the set of transactions of thread P_k that conflict with transaction T_{ij}. We partition the threads $P_1, \ldots, P_M$ into 3 classes Q_0, Q_1, and Q_2, such that:

- Q_0 contains every thread P_k which either $|A_k| = 0$, or $|A_k| > 0$ but the positions of the transactions in A_k are such that it is impossible to overlap with F_{ij} for any random intervals q_i and q_k.
- Q_1 contains every thread P_k with $0 < |A_k| < \alpha_i$, and at least one of the transactions in A_k is positioned so that it is possible to overlap with frame F_{ij} for some choices of random intervals q_i and q_k.
- Q_2 contains every thread P_k with $\alpha_i \leq |A_k|$. Note that $|Q_2| \leq C_i/\alpha_i = \ln(MN)$.

Let Y_k be a random binary variable, such that $Y_k = 1$ if in thread P_k any of the transactions in A_k becomes high priority in F_{ij} (same frame with T_{ij}), and $Y_k = 0$ otherwise. Let $Y = \sum_{k=1}^{M} Y_k$. Note that $|A'| = Y$. Denote $pr_k = \mathbf{Pr}(Y_k = 1)$. We can write $Y = Z_0 + Z_1 + Z_2$, where $Z_\ell = \sum_{P_k \in Q_\ell} Y_k$, for $0 \leq \ell \leq 2$. Clearly, $Z_0 = 0$. and $Z_2 \leq |Q_2| \leq \ln(MN)$.

Recall that for each thread P_k there is a random initial interval with q_k frames, where q_k is chosen uniformly at random in $[0, \alpha_k - 1]$. Given the random choice of P_k, $0 < pr_k \leq |A_k|/\alpha_i < 1$, since there are $|A_k| < \alpha_i$ conflicting transactions in A_i and there are at least α_i random choices for the relative position of transaction T_{ij}. Consequently,

$$\mu = \mathbf{E}[Z_1] = \sum_{P_k \in Z_1} pr_k \leq \sum_{P_k \in Z_1} \frac{|A_k|}{\alpha_i} = \frac{1}{\alpha_i} \cdot \sum_{P_k \in Z_1} |A_k| \leq \frac{C_i}{\alpha_i} \leq \ln(MN).$$

By applying a Chernoff bound we obtain that $\mathbf{Pr}(Z_1 > (e^2+1)\mu) < e^{-(e^2+1)\mu} < e^{-2\ln(MN)} = (MN)^{-2}$. Since $Y = Z_0 + Z_1 + Z_2$, and $Z_2 \leq \ln(MN)$, we obtain $\mathbf{Pr}((|A'| = Y) > ((e^2+2)\mu = \Phi - 1)) < (MN)^{-2}$, as needed.

Lemma 4. *All transactions commit by the end of their assigned frames with probability at least $1 - (MN)^{-1}$.*

Proof. From Lemmas 1 and 3, Φ time steps do not suffice to commit transaction T_{ij} within its assigned frame F_{ij} with probability at most $(NM)^{-2}$ (we call this a *bad event*). Considering all the MN transactions in the window a bad event for any of them occurs with probability at most $MN \cdot (MN)^{-2} = (MN)^{-1}$. Thus, with probability at least $1 - (MN)^{-1}$, all transactions will commit within their assigned frames.

Since $C = \max_i C_i$, the makespan bound of the algorithm follows immediately from Lemma 4.

Theorem 1 (Makespan of Offline-Greedy). *Algorithm* Offline-Greedy *produces a schedule of length* $O(\tau \cdot (C + N \log(MN)))$ *with probability at least* $1 - (MN)^{-1}$.

Since in the conflict graph $G(\mathcal{T}(W))$, $C \leq \lambda_{\max} \cdot \gamma_{\max}$, we have that $makespan = O(\tau \cdot (\lambda_{\max} \cdot \gamma_{\max} + N \log(MN)))$. Further, since $C \geq \gamma_{\max}$, and $\tau \cdot N$ is a lower bound on the schedule length, $makespan_{\text{opt}} \geq \tau \cdot \max(\gamma_{\max}, N)$. Therefore, the competitive ratio of the schedule is $O(\lambda_{\max} + \log(MN)) = O(s + \log(MN))$.

Corollary 1 (Competitive Ratio of Offline-Greedy). *The makespan of the schedule produced by Algorithm* Offline-Greedy *has competitive ratio* $O(s + \log(MN))$ *with probability at least* $1 - (MN)^{-1}$.

5 Online Algorithm

We present Algorithm Online-Greedy algorithm (Algorithm 2), which is online in the sense that it does not depend on knowing the dependency graph to resolve conflicts. In addition to M and N, we assume that each thread P_i knows C_i. This algorithm is similar to Algorithm 1 with the difference that in the conflict resolution phase we use as a subroutine a variation of Algorithm RandomizedRounds proposed by Schneider and Wattenhofer [21]. The makespan of the online algorithm is slightly worse than the offline algorithm, since the duration of the frame is now $\Phi' = O(\tau \cdot \log^2(MN))$.

There are two different priorities associated with each transaction under this algorithm. The pair of priorities for a transaction is given as a vector $\langle \pi^{(1)}, \pi^{(2)} \rangle$, where $\pi^{(1)}$ represents the Boolean priority value *low* or *high* (with respective values 1 and 0) as described in Algorithm 1, and $\pi^{(2)} \in [1, M]$ represents the random priorities used in Algorithm RandomizedRounds. The conflicts are resolved in lexicographic order based on the priority vectors, so that vectors with lower lexicographic order have higher priority.

When a transaction T is issued, it starts to execute immediately in low priority ($\pi^{(1)} = 1$) until the respective randomly chosen time frame F starts where it switches to high priority ($\pi^{(1)} = 0$). Once in high priority, the field $\pi^{(2)}$ will be used to resolve conflicts with other high priority transactions. A transaction chooses a discrete number $\pi^{(2)}$ uniformly at random in the interval $[1, M]$ on start of the frame F_{ij}, and after every abort. In case of a conflict of T with another high priority transaction K which has higher $\pi^{(2)}$ value than T, then T proceeds and K aborts. The procedure $abort(T, K)$ aborts transaction K.

Algorithm 2. Online-Greedy

Input: A $M \times N$ window W of transactions with M threads each with N transactions; Each thread P_i knows C_i, the maximum number of transactions in W that any transaction in P_i conflicts with; Each transaction has same duration τ;

Output: A greedy execution schedule for the window of transactions W;

Divide time into frames of $\Phi' = 16e \cdot \Phi \ln(MN)$ time steps, where $\Phi = 1 + (e^2 + 2)\ln(MN)$;
Each thread P_i chooses a random number $q_i \in [0, \alpha_i - 1]$ for $\alpha_i = C_i / \ln(NM)$;
Each transaction T_{ij} is assigned to frame $F_{ij} = q_i + (j-1)$;
Associate pair of priorities $\langle \pi_{ij}^{(1)}, \pi_{ij}^{(2)} \rangle$ to each transaction T_{ij};
foreach *time step* $t = 0,\ 1\tau,\ 2\tau,\ 3\tau, \ldots$ **do**
  Phase 1: Priority Assignment
  foreach *transaction* T_{ij} **do**
    if $t < F_{ij} \cdot \tau\Phi'$ **then** Priority $\pi_{ij}^{(1)} \leftarrow 1$ (low); **else** Priority $\pi_{ij}^{(1)} \leftarrow 0$ $(high)$;
  Phase 2: Conflict Resolution
  if $\pi_{ij}^{(1)} == 0$ *(T_{ij} has high priority)* **then**
    On (re)start of transaction T_{ij};
    $\pi_{ij}^{(2)} \leftarrow$ random integer in $[1, M]$;
    On conflict of transaction T_{ij} with high priority transaction T_{kl};
    if $\pi_{ij}^{(2)} < \pi_{kl}^{(2)}$ **then** $abort(T_{ij}, T_{kl})$; **else** $abort(T_{kl}, T_{ij})$;

5.1 Analysis of Online Algorithm

In the analysis given below, we study the makespan and the response time of Algorithm Online-Greedy. The analysis is based on the following adaptation of the response time analysis of a one-shot transaction problem with algorithm RandomizedRounds [21]. It uses the following Chernoff bound:

Lemma 5 (Chernoff Bound 2). *Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \le i \le n$, $\mathbf{Pr}(X_i = 1) = pr_i$, where $0 < pr_i < 1$. Then, for $X = \sum_{i=1}^{n} X_i$, $\mu = \mathbf{E}[X] = \sum_{i=1}^{n} pr_i$, and any $0 < \delta \le 1$, $\mathbf{Pr}(X < (1-\delta)\mu) < e^{-\delta^2\mu/2}$.*

Lemma 6. ***(Adaptation from Schneider and Wattenhofer [21])*** *Given a one-shot transaction scheduling problem with M transactions, the time span a transaction T needs from the moment it is issued until commit is $16e(d_T+1)\log n$ with probability at least $1 - \frac{1}{n^2}$, where d_T is the number of transactions conflicting with T.*

Proof. Consider the respective conflict graph G of the one-shot problem. Let N_T denote the set of conflicting transactions for T (these are the neighbors of T in G). Let $d_T = |N_T| \le M$. Let y_T denote the random priority number choice of T in

range $[1, M]$. The probability that for transaction T no transaction $K \in N_T$ has the same random number is:

$$\mathbf{Pr}(\nexists K \in N_T | y_T = y_K) = \left(1 - \frac{1}{M}\right)^{d_T} \geq \left(1 - \frac{1}{M}\right)^{M} \geq \frac{1}{e}.$$

The probability that y_T is at least as small as y_K for any transaction $K \in N_T$ is $\frac{1}{d_T+1}$. Thus, the chance that y_T is smallest and different among all its neighbors in N_T is at least $\frac{1}{e(d_T+1)}$. If we conduct $16e(d_T+1)\ln n$ trials, each having success probability $\frac{1}{e(d_T+1)}$, then the probability that the number of successes Z is less than $8\ln n$ becomes: $\mathbf{Pr}(Z < 8 \cdot \ln n) < e^{-2 \cdot \ln n} = \frac{1}{n^2}$, using the Chernoff bound of Lemma 5.

Lemma 7. *In Algorithm* Online-Greedy *all transactions commit by the end of their assigned frames with probability at least* $1 - 2(MN)^{-1}$.

Proof. According to the algorithm, a transaction T_{ij} becomes high priority ($\pi_{ij}^{(1)} = 0$) in frame F_{ij}. When this occurs the transaction will start to compete with other transactions which became high priority during the same frame. Lemma 3 from the analysis of Algorithm 1, implies that the effective degree of T_{ij} with respect to high priority transactions is $d_T > \Phi - 1$ with probability at most $(MN)^{-2}$ (we call this *bad event-1*). From Lemma 6, if $d_T \leq \Phi - 1$, the transaction will not commit within $16e(d_T + 1)\log n \leq \Phi'$ time slots with probability at most $(MN)^{-2}$ (we call this *bad event-2*). Therefore, T_{ij} does not commit in F_{ij} when either bad event-1 or bad event-2 occurs, which happens with probability at most $(MN)^{-2} + (MN)^{-2} = 2(MN)^{-2}$. Considering now all the MN transactions, the probability of failure is at most $2(MN)^{-1}$. Thus, with probability at least $1 - 2(MN)^{-1}$, every transaction T_{ij} commits during the F_{ij} frame.

The makespan and competitive ratios follow immediately from Lemma 7.

Theorem 2 (Makespan of Online-Greedy**).** *Algorithm* Online-Greedy *produces a schedule of length* $O(\tau \cdot (C \log(MN) + N \log^2(MN)))$ *with probability at least* $1 - 2(MN)^{-1}$.

Corollary 2 (Competitive Ratio of Online-Greedy**).** *The makespan of the schedule produced by Algorithm* Online-Greedy *has competitive ratio* $O(s \cdot \log(MN) + \log^2(MN))$ *with probability at least* $1 - 2(MN)^{-1}$.

6 Adaptive Algorithm

A limitation of Algorithms 1 and 2 is that the values C_i need to be known in advance for each window W. We present the Algorithm Adaptive-Greedy (Algorithm 3) in which each thread can guess the individual values of C_i. The algorithm works based on the exponential back-off strategy used by many contention managers developed in the literature such as Polka.

Algorithm 3. Adaptive-Greedy

Input: An $M \times N$ execution window W with M threads each with N transactions, where C is unknown;
Output: A greedy execution schedule for the window of transactions;

Code for thread P_i;
begin
  Initial contention estimate $C_i \leftarrow 1$;
  repeat
    Online-Greedy(C_i, W);
    if *bad event* **then**
      $C_i \leftarrow 2 \cdot C_i$;
  until *all transactions are committed*;

Each thread P_i starts with assuming $C_i = 1$. Based on the current estimate C_i, the thread attempts to execute Algorithm 2, for each of its transactions assuming the window size $M \times N$. Now, if the choice of C_i is correct then each transaction of the thread in the window W of the thread P_i should commit by the end of the respective assigned frame that it becomes high priority. Thus, all transactions of thread P_i should commit within the time estimate of Algorithm 2 which is $L_i = O(\tau \cdot (C_i \log(MN) + N \log^2(MN)))$. However, if during L_i thread P_i is unable to commit one of its transactions within its assigned frame (we call this a *bad event*), then thread P_i will assume that the choice of C_i is incorrect, and will start over again with the remaining transactions assuming $C_i' = 2C_i$. Eventually thread P_i will guess the right value of C_i for the window W, and all its transactions will commit within their respective time frames. It is easy to that the correct choice of C_i will be reached by a thread P_i within $\log C_i$ iterations. The total makespan is asymptotically the same as with Algorithm 2.

7 Conclusions

We considered greedy contention managers for transactional memory for $M \times N$ windows of transactions with M threads and N transactions per thread and present three new algorithms for contention management in transactional memory from a worst-case perspective. These algorithms are efficient, adaptive, and improve on the worst-case performance of previous results. These are the first such results for the execution of sequences of transactions instead of the one-shot problem considered in the literature. Our algorithms present new trade-offs in the analysis of greedy contention managers for transactional memory.

When we consider variable time durations for the transactions, in the makespan bounds expressions in Theorems 1 and 2 of our algorithms we can replace the parameter τ with $\tau^{\max}$, which is the maximum duration of any transaction in the window. The impact is that in the competitive ratio in Corollaries 1 and 2 there will appear an additional factor $\tau^{\max}/\tau^{\min}$, where $\tau^{\min}$ is the minimum duration of any transaction in the window. In the algorithms, the basic

time step duration is changed from τ to $\tau^{\max}$. Note that with variable time delays the transactions are not perfectly aligned when they enter a frame. In Offline-Greedy, this doesn't cause a problem when we compute the independent sets. On the other hand, we need to modify Online-Greedy so that when a high-priority transaction aborts, it always gives the right of way to the transaction that aborted it.

With this work, we are left with some issues for future work. The other aspect is to explore alternative algorithms where the randomization does not occur at the beginning of each window but rather during the execution of the algorithm by inserting random periods of low priority between the subsequent transactions in each thread. One may also consider the dynamic expansion and contraction of the execution window to preserve the contention measure C. This will result to more practical algorithms with good performance guarantees.

References

1. Ansari, M., Kotselidis, C., Lujan, M., Kirkham, C., Watson, I.: On the performance of contention managers for complex transactional memory benchmarks. In: ISPDC '09: Proceedings of the 8th International Symposium on Parallel and Distributed Computing (2009)
2. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: Seznec, A., Emer, J., O'Boyle, M., Martonosi, M., Ungerer, T. (eds.) HiPEAC 2009. LNCS, vol. 5409, pp. 4–18. Springer, Heidelberg (2009)
3. Attiya, H., Epstein, L., Shachnai, H., Tamir, T.: Transactional contention management as a non-clairvoyant scheduling problem. Algorithmica 57(1), 44–61 (2010)
4. Attiya, H., Milani, A.: Transactional scheduling for read-dominated workloads. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 3–17. Springer, Heidelberg (2009)
5. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In: PODC '08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 125–134. ACM, New York (2008)
6. Dragojević, A., Guerraoui, R., Singh, A.V., Singh, V.: Preventing versus curing: avoiding conflicts in transactional memories. In: PODC '09: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 7–16. ACM, New York (2009)
7. Guerraoui, R., Herlihy, M., Kapalka, M., Pochon, B.: Robust Contention Management in Software Transactional Memory. In: SCOOL '05: Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (2005)
8. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 303–323. Springer, Heidelberg (2005)
9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC '01: Proceedings of the Twenty-Fourth Annual Symposium on Principles of Distributed Computing (2005)

10. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA '03: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 388–402 (2003)
11. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60 (2005)
12. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, pp. 522–529 (2003)
13. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM, New York (2003)
14. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300 (1993)
15. Khot, S.: Improved inapproximability results for maxclique, chromatic number and approximate graph coloring. In: FOCS '01: Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science, pp. 600–609 (2001)
16. Leighton, F.T., Maggs, B.M., Rao, S.B.: Packet routing and job-shop scheduling in $O(congestion + dilation)$ steps. Combinatorica 14, 167–186 (1994)
17. Ramadan, H.E., Rossbach, C.J., Porter, D.E., Hofmann, O.S., Bhandari, A., Witchel, E.: Metatm/txlinux: Transactional memory for an operating system. IEEE Micro. 28(1), 42–51 (2008)
18. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC '05: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248 (2005)
19. Scherer III, W.N., Scott, M.L.: Contention management in dynamic software transactional memory. In: CSJP '04: Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs, St. John's, NL, Canada (2004)
20. Scherer III, W.N., Scott, M.L.: Randomization in STM contention management (POSTER). In: PODC '05: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, Las Vegas, NV (2005)
21. Schneider, J., Wattenhofer, R.: Bounds on contention management algorithms. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 441–451. Springer, Heidelberg (2009)
22. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213. ACM, New York (1995)
23. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA '08: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 169–178 (2008)

Scalable Flat-Combining Based Synchronous Queues

Danny Hendler[1], Itai Incze[2], Nir Shavit[2,3], and Moran Tzafrir[2]

[1] Ben-Gurion University
[2] Tel-Aviv University
[3] Sun Labs at Oracle

Abstract. In a *synchronous queue*, producers and consumers handshake to exchange data. Recently, new scalable unfair synchronous queues were added to the Java JDK 6.0 to support high performance thread pools.

This paper applies flat-combining to the problem of designing a synchronous queue algorithm. We first use the original flat-combining algorithm, a single "combiner" thread acquires a global lock and services the other threads' combined requests with very low synchronization overheads. As we show, this single combiner approach delivers superior performance up to a certain level of concurrency, but unfortunately does not continue to scale beyond that point.

In order to continue to deliver scalable performance as concurrency increases, we introduce a new *parallel flat-combining* algorithm. The new algorithm dynamically adds additional concurrently executing flat-combiners that coordinate their work. It enjoys the low coordination overheads of sequential flat combining, with the added scalability that comes with parallelism.

Our novel unfair synchronous queue using parallel flat combining exhibits scalability far and beyond that of the JDK 6.0 algorithm: it matches it in the case of a single producer and consumer, and is superior throughout the concurrency range, delivering up to 11 (eleven) times the throughput at high concurrency.

1 Introduction

This paper presents a new highly scalable design of an *unfair synchronous queue*, a fundamental concurrent data structure used in a variety of concurrent programming settings.

In many applications, one or more *producer* threads produce items to be consumed by one or more *consumer* threads. These items may be jobs to perform, keystrokes to interpret, purchase orders to execute, or packets to decode. As noted in [7], many applications require "poll" and "offer" operations which take an item only if a producer is already present, or put an item only if a consumer is already waiting (otherwise, these operations return an error). The *synchronous queue* provides such a "pairing up" of items, without buffering; it is entirely symmetric: Producers and consumers wait for one another, rendezvous, and leave

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 79–93, 2010.

in pairs. The term "unfair" refers to the fact that the queue is actually a pool [5]: it does not impose an order on the servicing of requests, and permits starvation. Previous synchronous queue algorithms were presented by Hanson [3], by Scherer, Lea and Scott [8,7,9] and by Afek et al. [1]. A survey of past work on synchronous queues can be found in [7].

New scalable implementations of synchronous queues were recently introduced by Scherer, Lea, and Scott [7] into the Java 6.0 JDK, available on more than 10 million desktops. They showed that the unfair version of a synchronous queue delivers scalable performance, both in general and when used to implement the JDK's thread pools.

In a recent paper [4], we introduced a new synchronization paradigm called *flat combining* (FC). At the core of flat combining is a low cost way to allow a single "combiner" thread at a time to acquire a global lock on the data structure, learn about all concurrent access requests by other threads, and then perform their combined requests on the data structure. This technique has the dual benefit of reducing the synchronization overhead on "hot" shared locations, and at the same time reducing the overall cache invalidation traffic on the structure. The effect of these reductions is so dramatic, that in a kind of "anti-Amdahl's law" effect, they outweigh the loss of parallelism caused by allowing only one combiner thread at a time to manipulate the structure.

This paper applies the flat-combining technique to the synchronous queue implementation problem. We begin by presenting a scalable flat-combining implementation of an *unfair synchronous queue* using the technique suggested in [4]. As we show, this implementation outperforms the new Java 6.0 JDK implementation at all concurrency levels (by a factor of up to 3 on a Sun Niagara 64 way multicore). However, it does not continue to scale beyond some point, because in the end it is based on a single sequentially executing combiner thread that executes the operations of all others.

Our next implementation, and the core result in this paper, is a synchronous queue based on *parallel flat-combining*, a new flat combining algorithm in which multiple instances of flat combining are executed in parallel in a coordinated fashion. The parallel flat-combining algorithm spawns new instances of flat-combining dynamically as concurrency increases, and folds them as it decreases. The key problem one faces in such a scheme is how to deal with imbalances: in a synchronous queue one must "pair up" requests, without buffering the imbalances that might occur. Our solution is a dynamic two level exchange mechanism that manages to take care of imbalances with very little overhead, a crucial property for making the algorithm work at both high and low load, and at both even and uneven distributions. We note that a synchronous queue, in particular a parallel one, requires a higher level of coordination from a combiner than that of queues, stacks, or priority queues implemented in our previous paper [4], which introduced flat combining. This is because the lack of buffering means there is no "slack": the combiner must actually match threads up before releasing them.

As we show, our parallel flat-combining implementation of an unfair synchronous queue outperforms the single combiner, continuing to improve with

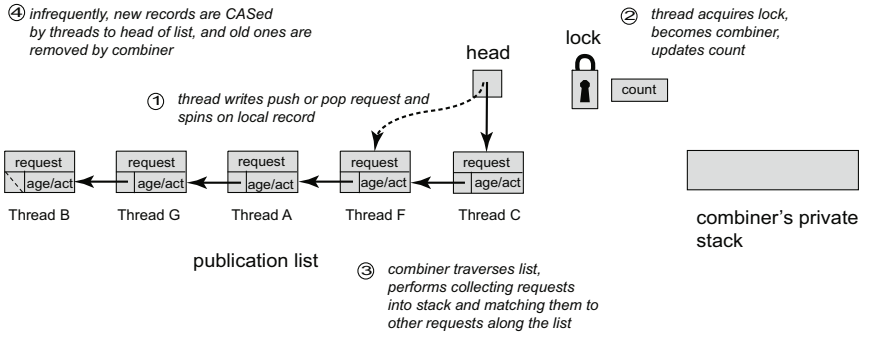


Fig. 1. A synchronized-queue using a *single combiner* flat-combining structure. Each record in the publication list is local to a given thread. The thread writes and spins on the request field in its record. Records not recently used are once in a while removed by a combiner, forcing such threads to re-insert a record into the publication list when they next access the structure.

the level of concurrency. On a Sun Niagara 64 way multicore, it reaches up to 11 times the throughput of the JDK implementation at 64 threads.

The rest of this paper is organized as follows. We outline the basic sequential flat-combining algorithm and describe how it can be used to implement a synchronous queue in Section 2. In Section 3, we describe our parallel FC synchronous queue implementation. Section 4 reports on our experimental evaluation. We conclude the paper in Section 5 with a short discussion of our results.

2 A Synchronous Queue Using Single-Combiner Flat-Combining

In a previous paper [4], we showed how given a sequential data structure D, one can design a (single-combiner) *flat combining* (henceforth FC) concurrent implementation of the structure. For presentation completeness, we outline this basic sequential flat combining algorithm in this section and describe how we use it to implement a synchronous queue. Then, in the next section, we present our parallel FC algorithm that is based on running multiple dynamically maintained instances of this basic algorithm in a two level hierarchy.

As depicted in Figure 1, to implement a single instance of FC, a few structures are added to a sequential structure D: a *global lock*, a *count* of the number of combining passes, and a pointer to the *head* of a *publication list*. The publication list is a list of thread-local records of a size proportional to the number of threads that are concurrently accessing the shared object. Though one could implement the list in an array, the dynamic publication list using thread local pointers is necessary for a practical solution: because the number of potential threads is unknown and typically much greater than the array size, using an array one would have had to solve a renaming problem [2] among the threads accessing it.

This would imply a CAS per location, which would give us little advantage over existing techniques.

Each thread t accessing the structure to perform an invocation of some method m on the shared object executes the following sequence of steps. We describe only the ones important for coordination so as to keep the presentation as simple as possible. The following then is the *single combiner algorithm* for a given thread executing a method m:

1. Write the invocation opcode and parameters (if any) of the method m to be applied sequentially to the shared object in the *request* field of your thread local publication record (no need to use a load-store memory barrier). The *request* field will later be used to receive the response. If your thread local publication record is marked as active continue to step 2, otherwise continue to step 5.
2. Check if the global lock is taken. If so (another thread is an active combiner), spin on the *request* field waiting for a response to the invocation (one can add a yield at this point to allow other threads on the same core to run). Once in a while, while spinning, check if the lock is still taken and that your record is active. If your record is inactive proceed to step 5. Once the response is available, reset the request field to null and return the response.
3. If the lock is not taken, attempt to acquire it and become a combiner. If you fail, return to spinning in step 2.
4. Otherwise, you hold the lock and are a combiner.
   - Increment the combining pass *count* by one.
   - Traverse the publication list (our algorithm guarantees that this is done in a wait-free manner) from the publication list head, combining all non-null method call invocations, setting the *age* of each of these records to the current *count*, applying the combined method calls to the structure D, and returning responses to all the invocations.
   - If the *count* is such that a cleanup needs to be performed, traverse the publication list from the *head*. Starting from the second item (as we explain below, we always leave the item pointed to by the *head* in the list), remove from the publication list all records whose *age* is much smaller than the current *count*. This is done by removing the record and marking it as inactive.
   - Release the *lock*.
5. If you have no thread local publication record allocate one, marked as active. If you already have one marked as inactive, mark it as active. Execute a store-load memory barrier. Proceed to insert the record into the list by repeatedly attempting to perform a successful CAS to the *head*. If and when you succeed, proceed to step 1.

Records are added using a CAS only to the head of the list, and so a simple wait free traversal by the combiner is trivial to implement [5]. Thus, removal will not require any synchronization as long as it is not performed on the record pointed to from the head: the continuation of the list past this first record is only ever changed by the thread holding the global lock. Note that the first item is not an

anchor or dummy record, we are simply not removing it. Once a new record is inserted, if it is unused it will be removed, and even if no new records are added, leaving it in the list will not affect performance.

The common case for a thread is that its record is active and some other thread is the combiner, so it completes in step 2 after having only performed a store and a sequence of loads ending with a single cache miss. This is supported by the empirical data presented later.

Our implementation of the FC mechanism allows us to provide the same clean concurrent object-oriented interface as used in the Java concurrency package [6] and similar C++ libraries [13], and the same consistency guarantees. We note that the Java concurrency package supports a time-out capability that allows operations awaiting a response to give up after a certain elapsed time. It is straightforward to modify the push and pop operations we support in our implementation into dual operations and to add a time-out capability. However, for the sake of brevity, we do not describe the implementation of these features in this extended abstract.

To access the synchronous queue, a thread t posts the respective pair <PUSH,v> or <POP,0> to its publication record and follows the FC algorithm. As seen in Figure 1, to implement the synchronous queue, the combiner keeps a *private stack* in which it records push and pop requests (and for each also the publication record of the thread that requested them). As the combiner traverses the publication list, it compares each requested operation to the top operation in the private stack. If the operations are complementary, the combiner provides the requestor and the thread with the complementary operation with their appropriate responses, and releases them both. It then pops the operation from the top of the stack, and continues to the next record in the publication list. The stack can thus alternately hold a sequence of pushes or a sequence of pops, but never a mix of both.

In short, during a single pass over the publication list, the combiner matches up as best as possible all the push and pop pairs it encountered. The operations remaining in the stack upon completion of the traversal are in a sense the "overflow" requests of a certain type, that were not serviced during the current combining round and will remain for the next.

In Section 4 a single instance of the single combiner FC algorithm is shown to provide a synchronous queue with superior performance to the one in JDK6.0, but it does not continue to scale beyond a certain number of threads. To overcome this limitation, we now describe how to implement a highly scalable generalization of the FC paradigm using multiple concurrent instances of the single combiner algorithm.

3 Parallel Flat Combining

In this section we provide a description of the parallel flat combining algorithm. We extend the single combiner algorithm to multiple parallel combiners in the following way. We use two types of flat combining coordination structures, depicted in Figure 2. The first is a *dynamic* FC structure that has the ability to

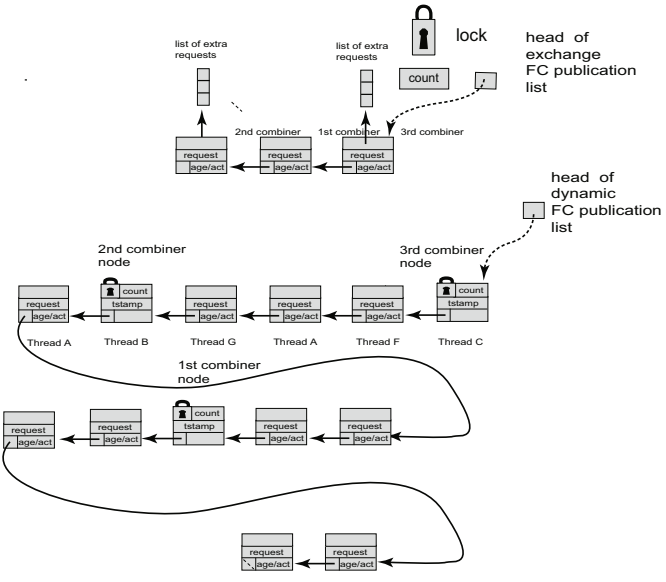


Fig. 2. A synchronized-queue based on *parallel flat combining*. There are two main interconnected elements: the *dynamic* FC structure and the *exchange* FC structure. As can be seen, in this example there are three combiner sublists with approximately 4 records per sublist. Each of the combiners also has a record in the exchanger FC structure.

split the publication list into shorter sublists when its length passes a certain threshold, and collapse publication sublists if their lengths go below a certain threshold. The second is an *exchange* single combiner FC structure that implements a synchronous queue in which each request can involve a collection of several push or pop requests.

Each of the multiple combiners that operate on sublists of the dynamic FC structure may fail to match all its requests and be left with an "overflow" of operations of the same type. The exchange FC structure is used for trying to match these overflows. The key technical challenge of the new algorithm is to allow coordination between the two levels of structures: multiple parallel dynamically-created single combiner sublists, and the exchange structure that deals with their overflows. Any significant overhead in this mechanism will result in a performance deterioration that will make the scheme as a whole work poorly.

Each record in the dynamic flat combining publication list is augmented with a pointer (not shown in Figure 2) to a special *combiner node* that contains the lock and other accounting data associated with the sublist currently associated with the record; the request itself is now also a separate *request structure* pointed to by the publication record (this structural change is not depicted in Figure 2). Initially there is a single combiner node and the initial publication records are added to the list starting with this node and point to it.

Each thread t performing an invocation of a push or a pop starts by accessing the head of the *dynamic FC publication list* and executes the following sequence of steps:

1. Write the invocation opcode of the operation and its parameters (if any) to a newly created request structure. If your thread local publication record is marked as active, continue to step 3., otherwise mark it as active and continue to step 2.
2. Publication record is not in list: count the number of records in the sublist pointed to by the currently first combining node in the dynamic FC structure. If less than the threshold (in our case 8, chosen statically based on empirical testing), set the combiner pointer of your publication record to this combining node, and try to CAS yourself into this sublist. Otherwise:
   - Create a new combiner node, pointing to the currently first combiner node.
   - Try to CAS the head pointer to point to the new combiner node.

   Repeat the above steps until your record is in the list and then proceed to step 3.
3. Check if the lock associated with the combiner node pointed at by your publication record is taken. If so, proceed similarly to step 2. of the single FC algorithm: spin on your request structure waiting for a response and, once in while, check if the lock is still taken and that your publication record is active. If your response is available, reset the request field to null and return the response. If your record is inactive, mark it as active and proceed to step 2; if the lock is not taken, try to capture it and, if you succeed, proceed to step 4.
4. You are now a combiner in your sublist: run the combiner algorithm using a local stack, matching pairs of requests in your sublist. If, after a few rounds of combining, there are leftover requests in the stack that cannot be matched, access the exchanger FC structure, creating a record pointing to a list of excess request structures and add it to the exchanger's publication list using the single FC algorithm. The excess request structures are no longer pointed at from the corresponding records of the dynamic FC list.
5. If you become a combiner in the exchanger FC structure, traverse the exchanger publication list using the single combiner algorithm. However, in this single combiner algorithm, each request record points to a list of overflow requests placed by a combiner of some dynamic list, and so you must either match (in case of having previously pushed counter-requests) or push (in other cases) all items in each list before signaling that the request is complete. This task is simplified by the fact that the requests will always be all pushes or all pops (since otherwise they would have been matched in the dynamic list and never posted to the exchange).

In our implementation we chose to split lists so that they contain approximately 8 threads each (in Figure 2 the threshold is 4). Making the lengths of the sublists and thresholds vary dynamically in a reactive manner is a subject for further research. For lack of space, detailed pseudo-codes of our algorithms are not presented in this extended abstract and will appear in the full paper.

3.1 Correctness

Though there is no obvious way to specify the "rendezvous" property of synchronous queues using a sequential specification, it is straightforward to see that our implementation is linearizable to the next closest thing, an object whose histories consist of a sequence of pairs consisting of a push followed by a pop of the matching value (i.e. push, pop, push, pop...). This follows immediately because each thread only leaves the structure after the flat combiner has matched it to a complementary concurrent operation, and we can linearize the operations at the point of the release of the first of them by the flat combiner.

In terms of robustness, our flat combining implementation is as robust as any global lock based data structure: in both cases a thread holding the lock could be preempted, causing all others to wait. [1]

4 Performance Evaluation

For our experiments we used two machines. The first is an Oracle 64-way Niagara II multicore machine with 8 SPARC cores that multiplex 8 hardware threads each, and share an on chip L2 cache. The second is an Intel Nehalem 8-way machine, with 4 cores that each multiplex 2 hardware threads. We ran benchmarks in Java using the Java 6.0 JDK. In the figures we will refer to these two architectures respectively as SPARC and INTEL.

Our empirical evaluation is based on comparing the relative performance of our new flat combining implementations to the most efficient known synchronous queue implementations: the current Java 6.0 JDK `java.util.concurrent` implementation of the unfair synchronous queue, and the recently introduced *Elimination-Diffraction* trees of Afek et al. [1].

The JDK algorithm, due to Scherer, Lea, and Scott [7], was recently added to Java 6.0 and was reported to provide a three fold improvement over the Java 5.0 unfair synchronous queue implementation. The JDK implementation uses a lock-free linked list based stack in the style of Treiber [12] in order to queue either producer requests or consumer requests but never both at the same time. Whenever a request appears, the queue is examined - if it is empty or has nodes which have the same type of requested operation, the request is enqueued using a CAS to the top of the list. Otherwise, the requested operation at the top of the stack is popped using a CAS operation on the head end of the list.

This JDK version must provide the option to *park* a thread (i.e. context switch it out) while it waits for its chance to rendezvous in the queue. A park operation is costly and involves a system call. This, as our graphs show, hurts the JDK algorithm's performance. To make it clear that the performance improvement we obtain relative to the JDK synchronous queue is not a result of the park calls, we implemented a version of the JDK algorithm with the parks neutralized, and use it in our comparison.

[1] On modern operating systems such as Solaris^{TM}, one can use mechanisms such as the *schetdl* command to control the quantum allocated to a combining thread, significantly reducing the chances of it being preempted.

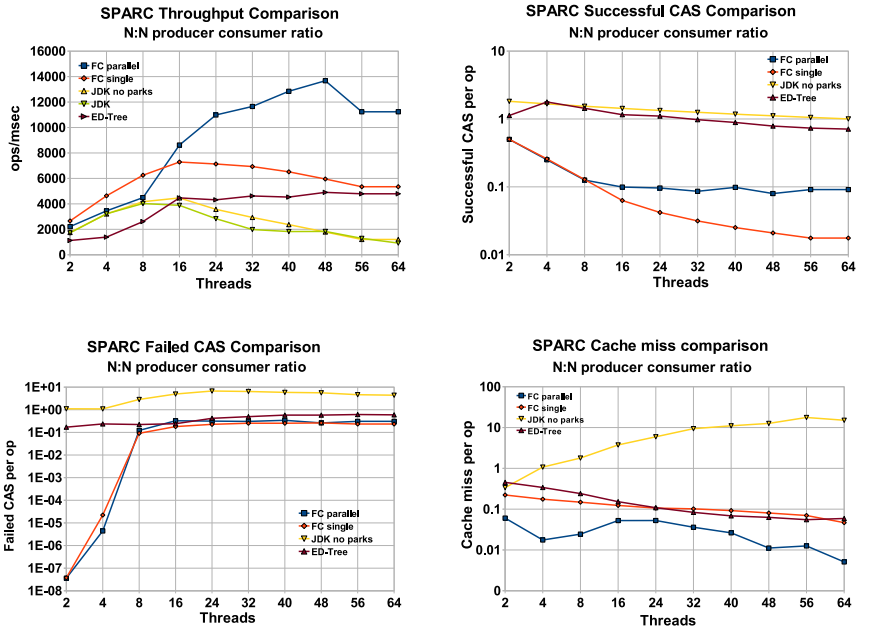


Fig. 3. A Synchronous Queue Benchmark with N consumers and N producers. The graphs show throughput, average CAS failures, average CAS successes, and L2 cache misses (all but the throughput are logarithmic and per operation). We show the throughput graphs for the JDK6.0 algorithm with parks, to make it clear that removing the parks helps performance in this benchmark.

The *Elimination-Diffracting tree* [1] (henceforth called ED-Tree), recently introduced by Afek et al., is a distributed data structure that can be viewed as a combination of an elimination-tree [10] and a diffracting-tree [11]. ED-Trees are randomized data-structures that distribute concurrent thread requests onto the nodes of a binary tree consisting of *balancers* (see [11]).

We compared the JDK and an ED-Tree based implementation of a synchronous queue to two versions of flat combining based synchronous queues, an FC synchronous queue using a single FC mechanism (denoted in the graphs as *FC single*), and our dynamic parallel version denoted as *FC parallel*. The FC parallel threshold was set to spawn a new combiner sublist for every 8 new threads.

4.1 Producer-Consumer Benchmarks

Our first benchmark, whose results are presented in Figure 3, is similar to the one in [7]. Producer and consumer threads continuously push and pop items from the queues. In the throughput graph in the upper lefthand corner, one can clearly see that both FC implementations outperform JDK's algorithm even with the park operations removed. Thus, in the remaining sections, we will no longer compare to the inferior JDK6.0 algorithm with parks.

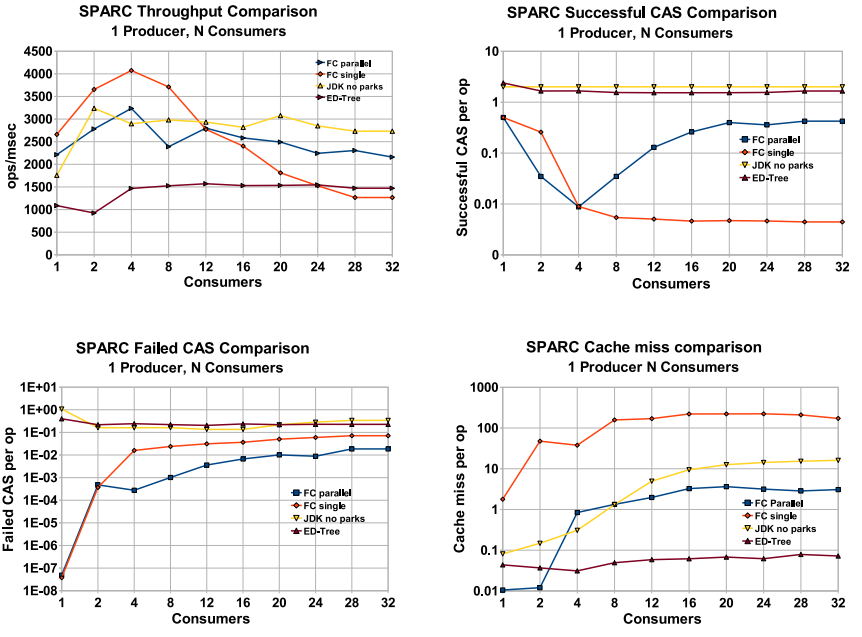


Fig. 4. Concurrent Synchronous Queue implementations with N consumers and 1 producer configuration: throughput, average CAS failures, average CAS successes, and L2 cache misses (all but throughput are logarithmic and per operation). Again, we show the throughput graphs for the JDK6.0 algorithm with parks, to make it clear that removing the parks helps performance in this benchmark.

As can be seen, the FC single version throughput exceeds the JDK's throughput by almost 80% at 16 threads, and remains better as concurrency levels grow up to 64. However, because there is only a single combiner, the FC single algorithm does not continue to scale beyond 16 threads. On the other hand, the FC parallel version is the same as the JDK up to 8 threads, and from 16 threads and onwards it continues to scale, reaching peak performance at 48 threads. At 64 threads, the FC parallel algorithm is about 11 times faster than the JDK.

The explanation for the performance becomes clear when one examines the other three graphs in Figure 3. The numbers of both successful and failed CAS operations in the FC algorithms are orders of magnitude lower (the scale is logarithmic) than in the JDK, as is the number of L2 cache misses. The gap between the FC parallel and the FC single and JDK continues to grow as its overall cache miss levels are consistently lower than their, and its CAS levels remain an order of magnitude smaller then theirs as parallelism increases. The low cache miss rates of the FC parallel when compared to FC single can be attributed to the fact that the combining list is divided and is thus shorter, having a better chance of staying in cache. This explains why the FC parallel algorithm continues to improve while the others slowly deteriorate as concurrency increases. At the highest concurrency levels, however, FC parallel's throughput also starts to

decline, since the cost incurred by a combiner thread that accesses the exchange increases as more combiner threads contend for it.

Since the ED-Tree algorithm is based on a static tree (that is, a tree whose size is proportional to the number of threads sharing the implementation rather than the number of threads that actually participate in the execution), it incurs significant overheads and has the worst performance among all evaluated algorithms in low concurrency levels.

However, for larger numbers of threads, the high parallelism and low contention provided by the ED-Tree allow it to significantly scale up to 16 threads, and to sustain (and even slightly improve) its peak performance in higher concurrency levels. Starting from 16 threads and on, the performance of the ED-Tree exceeds that of the JDK and it is almost 5-fold faster than the JDK for 64 threads.

Both FC algorithms are superior to the ED-Tree in all concurrency levels. Since ED-Tree is highly parallel, the gap between its performance and that of FC-single decreases as concurrency increases, and at 64 threads their performance is almost the same. The FC-parallel implementation, on the other hand, outperforms the ED-Tree implementation by a wide margin in all concurrency levels and provides almost three times the throughput at 48 threads.

Also here, the performance differences becomes clearer when we examine CAS and cache miss statistics (see Figure 3). Similarly to the JDK, the ED-Tree algorithm performs a relatively high number of successful CAS operations but, since its operations are spatially spread across the nodes of the tree, it incurs a much smaller number of failed CAS operations. The number of cache misses it incurs is close to that of FC-single implementation, yet is about 10-fold higher than FC-parallel implementation at high concurrency levels.

Figure 5-(a) shows the throughput on the Intel Nehalem architecture. As can be seen, the behavior is similar to that on SPARC up to 8 threads (recall that the Nehalem has 8 hardware threads): the FC-single algorithm performs better than the FC-parallel, and both FC algorithms significantly outperform the JDK and ED-Tree algorithms. The cache miss and CAS rate graphs, not shown for lack of space, provide a similar picture.

Our next benchmark, in Figure 4, has a single producer and multiple consumers. This is not a typical use of a synchronous queue since there is much waiting and little possibility of parallelism. However, this benchmark, introduced in [7], is a good stress test. In the throughput graph in Figure 4, one can see that the imbalance in the single producer test stresses the FC algorithms, making the performance of the FC parallel algorithm more or less the same as that of the JDK. However, we find it encouraging that an algorithm that can deliver up to 11 times the performance of the JDK in the balanced case, delivers comparable performance when there is a great imbalance among producers and consumers.

What is the explanation for this behavior? In this benchmark there is a fundamental lack of parallelism even as the number of threads grows: in all of the algorithms, all of the threads but 2 - the producer and its previously matched consumer – cannot make progress. Recall that FC wins by having a single low

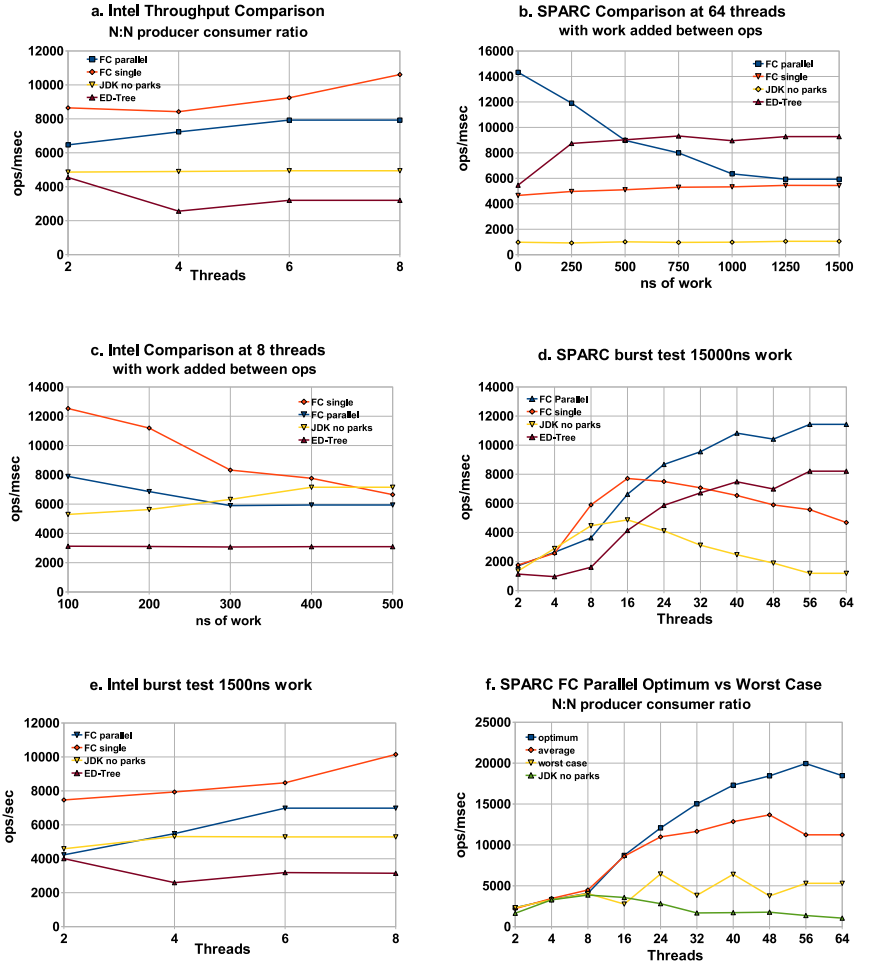


Fig. 5. (a) Throughput on the Intel architecture; (b) Decreasing request arrival rate on SPARC; (c) Decreasing request arrival rate on Intel; (d) Burst test throughput: SPARC; (e) Intel burst test throughput; (f) Worst-case vs. optimum distribution of producers and consumers.

overhead pass over the list service multiple threads. With this in mind, consider that in the single FC case, for every time a lock is acquired, about two requests are answered, and yet all the threads are continuously polling the lock. This explains the high cache invalidation rates, which together with a longer publication list traversed each time, explains why the single FC throughput deteriorates.

For the parallel FC algorithm, we notice that its failed CAS and cache miss rates are quite similar to those of the JDK. The parallel FC algorithm keeps cache misses and failed CAS rates lower than the single FC because threads are distributed over multiple locks and after failing as a combiner a thread goes to

the exchange. In most lists no combining takes place, and requests are matched at the exchange level (an indication of this is the successful CAS rate which is close to 1), not in the lists. Combiners accessing the exchange take longer to release their list ownership locks, and therefore cause other threads less cache misses and failed CAS operations. The exchange itself is again a single combiner situation (only accessed by a fraction of the participating threads) and thus with less overhead. The result is a performance very similar to that of the JDK.

4.2 Performance as Arrival Rates Change

In earlier benchmarks, the data structures were tested at very high arrival rates. These rates are common to some uses of concurrent data structures, but not to all.

Figures 5-(b) and 5-(c) show the change in throughput of the various algorithms as the method call arrival rates change when running on 64 threads on SPARC, or on 8 threads on Intel, respectively. In this benchmark, we inject a "work" period between calls a thread makes to the queue. The work consists of a loop which is measured to take a specific amount of time.

On SPARC, at all work levels, both FC implementations perform better or the same as the JDK, and on the Nehalem, where the cost of a CAS operation is lower, they converge to a point with JDK winning slightly over the FC parallel algorithm. The ED-Tree is the worst performer on Nehalem. On SPARC, on the other hand, ED-Tree is consistently better than JDK and FC single and its performance surpasses that of FC parallel as the amount of work added between operations exceeds 500 nanoseconds.

In Figures 5-(d) an 5-(e) we stress the FC implementations further. We show a burst test in which a longer "work period" is injected frequently, after every 50 operations. This causes the nodes on the combining lists to be removed frequently, thus putting more stress onto the combining allocation algorithm. Again this slows down the FC algorithms, but, as can be seen, they still perform well.[2]

4.3 The Pitfalls of the Parallel Flat Combining Algorithm

The algorithmic process used to create the parallel combining lists is another issue that needs further inspection.

Since the exchange shared by all sublists is at its core a simple flat combining algorithm, its performance relies on the fact that on average not many of the requests are directed to it because of an imbalance in the types of operations on the various sublists. This raises the question of what happens at the best case - when every combiner enjoys an equal number of producers and consumers, and the worst case - in which each combiner is unfortunate enough to continuously have requests of the same type.

Figure 5-(f) compares runs in which the combining lists are prearranged for the worst and best cases prior to execution. As can be seen, the worst case scenario

[2] Due to the different speeds of the SPARC and INTEL machines, different "work periods" were required in the tests on the two machines in order to demonstrate the effect of bursty workloads.

performance is considerately poor compared to the average and optimal ones. In cases were the number of combiners is even (16, 32, 48, 64 threads), performance solely relies on the exchange, and at one point it is worse than the JDK - this is most likely due to the overhead introduced by the parallel FC algorithm prior to the exchange algorithm. When the number of combiners is odd, there is a combiner which has both consumers and producers, which explains the gain in performance at 8, 24, 40, and 56 threads when compared to their successors. This yields the "saw" like pattern seen in the graph. Unsurprisingly, the regular run (denoted as "average") is much closer to the optimum.

In summary, our benchmarks show that the parallel flat-combining synchronous queue algorithm has the potential to deliver in the most common cases scalability beyond that achievable using fastest prior algorithms, and in the exceptional worst cases, under stress, they continue to deliver comparable performance.

5 Discussion

We presented a new parallel flat combining algorithm and used it to implement synchronous queues. The full code of our Java based implementation is available at `http://mcg.cs.tau.ac.il/projects/parallel-flat-combining`

We believe that, apart from providing a highly scalable implementation of a fundamental data structure, our new parallel flat combining algorithm is an example of the potential for using multiple instances of flat combining in a data structure to allow continued scaling of the overhead-reducing properties provided by the flat combining technique. Applying the parallel flat combining paradigm to additional key data-structures is an interesting venue for future research.

Acknowledgments

We thank Doug Lea for allowing us to use his Sun Niagara 2 multicore machine. We also thank the anonymous reviewers for their many helpful comments.

References

1. Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: Euro-Par '10 (June 2010) (to appear)
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37(3), 524–548 (1990)
3. Hanson, D.R.: C interfaces and implementations: techniques for creating reusable software. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)
4. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA '10: Proceedings of the Twenty Third annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 355–364 (2010)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, NY (2008)

6. Lea, D.: util.concurrent.ConcurrentHashMap in java.util.concurrent the Java Concurrency Package, http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/-src/main/java/util/concurrent/
7. Scherer III, W.N., Lea, D., Scott, M.L.: Scalable synchronous queues. ACM Commun. 52(5), 100–111 (2009)
8. Scherer III., W.N.: Synchronization and concurrency in user-level software systems. PhD thesis, Rochester, NY, USA, Adviser-Scott, Michael L (2006)
9. Scherer III, W.N., Scott, M.L.: Nonblocking concurrent data structures with condition synchronization. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 174–187. Springer, Heidelberg (2004)
10. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory of Computing Systems 30, 645–670 (1997)
11. Shavit, N., Zemach, A.: Diffracting trees. ACM Trans. Comput. Syst. 14(4), 385–428 (1996)
12. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (April 1986)
13. Tzafrir, M.: C++ multi-platform memory-model solution with java orientation, http://groups.google.com/group/cpp-framework

Fast Randomized Test-and-Set and Renaming*

Dan Alistarh[1], Hagit Attiya[1,2],
Seth Gilbert[3], Andrei Giurgiu[1], and Rachid Guerraoui[1]

[1] Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[2] Technion, Haifa, Israel
[3] National University of Singapore

Abstract. Most people believe that renaming is easy: simply choose a name *at random*; if more than one process selects the same name, then try again. We highlight the issues that occur when trying to implement such a scheme and shed new light on the read-write complexity of randomized renaming in an asynchronous environment. At the heart of our new perspective stands an adaptive implementation of a randomized test-and-set object, that has poly-logarithmic step complexity per operation, with high probability. Interestingly, our implementation is anonymous, as it does not require process identifiers. Based on this implementation, we present two new randomized renaming algorithms. The first ensures a tight namespace of n names using $O(n \log^4 n)$ total steps, with high probability. This significantly improves on the complexity of the best previously known namespace-optimal algorithms. The second algorithm achieves a namespace of size $k(1+\epsilon)$ using $O(k \log^4 k/\log^2(1+\epsilon))$ total steps, both with high probability, where k is the total contention in the execution. It is the first *adaptive* randomized renaming algorithm, and it improves on existing deterministic solutions by providing a smaller namespace, and by lowering step complexity.

1 Introduction

Names, or identifiers, are instrumental for efficiently solving a variety of problems that arise in distributed systems. And yet, in many cases, names are not available. Participants may be anonymous, or may wish to hide their true identity for reasons of privacy. Alternatively, participants may have names, but they may be taken from a very large namespace. For example, nearly every networked device has an ethernet address, and yet the namespace is so large as to reduce the usefulness of such names. Thus, a significant amount of research (e.g., [1–5]) has analyzed the feasibility and complexity of the *renaming* problem in a crash-prone distributed system.

Unfortunately, renaming in a fault-prone system can be expensive, if not impossible. For example, wait-free *tight* renaming—where the namespace exactly matches the set of n participants—is impossible for deterministic algorithms that tolerate crash failures [4–6]. Even *loose* renaming, where the namespace is of size $(2n-1)$, can be quite expensive, as the best known solutions require at least $\Theta(n^3)$ total steps [2, 7].

* The work of Dan Alistarh is supported by the *Swiss NCCR MICS* project. The work of Hagit Attiya is supported in part by the *Israel Science Foundation* (grant number 953/06).

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 94–108, 2010.

Yet in practice, most people believe that renaming is relatively easy: simply choose a name *at random*; if more than one process selects the same name, then try again. Several subtle problems occur when trying to implement such a scheme:

- How are processes scheduled? If the processes are scheduled in a synchronous fashion, then resolving contention among processes may not be difficult. However, in an asynchronous system, processes can be scheduled in any order. Worse, for a strong (adaptive) adversarial scheduler, which we consider in this paper, the choice of a schedule may depend on the random choices being made by the processes. Thus the random choices made by the processes are not entirely independent in the usual sense.
- From how big a namespace should the name be chosen? If the namespace is large, say $\Theta(n^2)$, where n is the number of participants, then such schemes are trivial. However, when the namespace is smaller—e.g., $(1+\epsilon)n$, for some $0 < \epsilon < 1$—the efficiency is less clear. And when the namespace is tight, i.e., of size precisely n, then some participants may have to retry repeatedly in order to find a free name.
- How does a participant claim a name? How does a participant determine whether its chosen name is unique? Effectively, when more than one participant selects the same name, participants must agree on which participant wins the name. Such agreement must be fault-tolerant, i.e., succeed even if processes fail; and it must be irrevocable, meaning that once a participant is assigned a name, it cannot later be forced to abandon it. The simple solution would be to run a distributed *consensus* algorithm to agree on which process owns each name. However, asynchronous, wait-free deterministic consensus is impossible [8], and the randomized version is inherently expensive, requiring $\Omega(n^2)$ total steps [9].

In this paper, we present two efficient *randomized* renaming algorithms for an asynchronous, fault-prone system subject to a strong, adaptive adversary.

The key building block for both algorithms is a new efficient implementation of a randomized test-and-set object. This algorithm answers the question of how a process can claim a name: a test-and-set object allows multiple processes to compete (for example, for a name), ensuring that there will be exactly one winner. The algorithm, which we call RatRace, is more efficient than consensus [9]: if there are k competitors, the total step complexity for a test-and-set is $O(k \log^2 k)$ read/write operations, with high probability. Of note, the RatRace algorithm is *adaptive*: the step complexity depends on k, the actual number of competitors (not on n, the total number of possible competitors). The algorithm efficiently combines the idea of a randomized splitter tree, first used in [10], with the tournament tree algorithm by Afek et al. [11]. Our renaming algorithms rely on both the *adaptivity* and *anonymity* properties of this implementation. Given the power of test-and-set to simplify coordination in a distributed system, and the efficiency of our solution, we expect that the RatRace algorithm may well be useful in other settings as well.

Tight Renaming. Our first renaming algorithm, called ReShuffle, produces unique names from a *tight* namespace of n names, using $O(n \log^4 n)$ total steps (reads and writes), with high probability. The algorithm uses a simple random process to compete for

names: each process repeatedly chooses a name at random, and attempts to claim it via a randomized test-and-set, stopping when it wins a name.

While the scheme is surprisingly simple, its analysis is rendered non-trivial by the fact that the scheduling and the failure pattern are controlled by the strong, adaptive adversary. For example, the adversary may look at a process's random choices prior to deciding whether it should be scheduled to perform a read or write operation. So the adversary might attempt to delay each process that chooses an unclaimed name until there are several other processes competing for the same name; since only one process can win, the adversary can, in this way, create a significant number of wasted steps. Since the schedule depends on the random choices, we cannot treat the processes' random steps as being uncorrelated, as needed for a standard analysis. We overcome this difficulty by carefully assessing the number of extra steps that the algorithm has to take because of adversarial scheduling or crashes.

Our algorithm improves significantly on the total step complexity of previous randomized or deterministic namespace-optimal implementations [7, 12], which have at least $\Theta(n^3)$ total step complexity. It guarantees unique names in a range from 1 to n in every execution, and terminates with probability 1.

Adaptive Renaming. Our second renaming algorithm, called AdaptiveSearch, is the first randomized *adaptive* renaming solution. That is, the algorithm's namespace and complexity depend on k, the number of processes competing, rather than n, the total number of possible participants. Given any constant $\epsilon > 0$, the AdaptiveSearch algorithm ensures unique names from a namespace of size $k(1+\epsilon)$, where k is the total contention in the current execution, using $O(k \log^4 k/\log^2(1+\epsilon))$ total steps, both with high probability. The main idea behind the algorithm is that processes try to acquire a name in intervals of increasing size; we prove that contention in intervals towards the edge of the namespace is low enough so that each process is successful with high probability. The algorithm improves on the adaptive deterministic solutions known so far by providing a smaller namespace than is otherwise feasible, and by improving the total step complexity. (The most efficient adaptive algorithm to date [13] has total step complexity $O(k^2)$, and renames in $(8k - \log k - 1)$ names.)

Discussion. Both algorithms are within logarithmic factors from the immediate lower bound of $\Omega(n)$ (or $\Omega(k)$) on the total step complexity of renaming. They also have the interesting property that they do not require unique process identifiers at the beginning of the execution—as such, the algorithms are *anonymous*.

Also of note, the ReShuffle algorithm is the first randomized algorithm to attain tight renaming with total step complexity less than $\Theta(n^2)$. Since $\Omega(n^2)$ is known to be the (tight) lower bound for the step complexity of randomized *consensus* [9], our algorithm yields the first clear separation in terms of complexity between randomized tight renaming and randomized consensus in asynchronous shared-memory.

The impossibility of wait-free renaming in a namespace smaller than $(2n-1)$ [4, 5] is circumvented by the use of randomization. There exist infinite length executions of infinitesimal probability weight, in which the algorithms do not terminate. Also, note that our test-and-set implementation cannot solve consensus for more than two processes, which is why it is not subject to the $\Omega(n^2)$ lower bound shown in [9].

Roadmap. In Section 2, we present the model and the problem statement, while Section 3 presents a detailed account of related work. Section 4 presents the implementation of adaptive test-and-set. Based on this, we introduce the ReShuffle algorithm in Section 5, and analyze its complexity. Section 6 presents the adaptive renaming implementation. We conclude in Section 7, stating some limitations of our approach, together with a host of open problems. Due to space constraints, some of the proofs have been deferred to the full version of this paper [14].

2 Model and Problem Statement

We assume an asynchronous shared memory model with n processes, $t < n$ of which may fail by crashing. Let M be the size of the space of initial identifiers that processes in the system may have[1]. For the adaptive algorithm, we consider k to denote total *contention*, i.e. the total number of processes that take steps during a certain execution. We assume that processes know n, but do not know k. Processes communicate through multiple-writer-multiple-reader atomic registers. Our algorithms are randomized, in that the processes' actions may depend on random local coin flips. We assume that the process failures and the scheduling are controlled by a strong adaptive adversary. In particular, the adversary knows the results of the random coin flips that the processes make and can adjust the schedule and the failure pattern accordingly.

In this context, the *renaming problem* requires that each correct process should eventually return a name, and the names returned should be unique. The size of the resulting namespace should only depend on n and on t. Note that, in our algorithms, we relax the assumption of unique initial identifiers, made in the original problem statement [6]. We assume $t \leq n - 1$, hence our solutions are *wait-free*. The complexity of our solutions is measured in terms of total steps (reads and writes, including random coin flips).

In the following, we say that an event happens "with high probability" (whp) if it occurs with probability $\geq 1 - 1/n^c$, with $c \geq 1$ constant. In the case of the adaptive algorithms, the probability bound is at least $\geq 1-1/k^c$, with $c \geq 1$. Note that the failure probability in the adaptive case may be tuned to depend on n, at the cost of increased complexity (i.e., a $\log n$ factor).

3 Related Work

Our test-and-set implementation re-uses ideas from the efficient randomized collect algorithm of Attiya et al. [10] and from the wait-free implementation of randomized test-and-set by Afek et al. [11]. We make use of the splitter object, originally introduced in [2], and its randomized version introduced in [10]. Overall, the structure of RatRace is similar to the adaptive algorithm for mutual exclusion by Anderson et al. [15], although the problem and the fault model we analyze are different. We use the two-process randomized test-and-set algorithm by Tromp and Vitànyi [16] as a building block.

[1] Note that some earlier work (e.g., [12]), uses n to denote the total number of identifiers that processes may have, which may also be seen as the maximum total number of processes in the system. They use k for the maximum number of processes that may participate in an execution (which we denote by n).

The renaming problem has been introduced by Attiya et al. [6]. In the original paper, the authors present a wait-free solution using $(2n-1)$ names in an asynchronous message-passing system, and show that at least $(n+1)$ names are required in the wait-free case. The lower bound was improved to $(2n-2)$ in a landmark paper by Herlihy and Shavit [4]. Recent work by Rajsbaum and Castañeda [5] shows that deterministic wait-free renaming may be possible for $\leq (2n-2)$ names for specific parameter values.

The complexity of deterministic shared-memory renaming implementations has been an active research topic. Burns and Peterson [17], Borowski and Gafni [18], Anderson and Moir [2], Moir and Garay [3] were among the first to propose wait-free, one-shot, deterministic algorithms into a namespace of size $(2n-1)$. These solutions have very high total step complexity; for some, the total step complexity is exponential (e.g. [3, 17]). Anderson and Moir [1] propose a variant of renaming that attains a tight namespace of n names using stronger *set-first-zero* objects. Note that their algorithm could be rephrased using our one-shot test-and-set implementation, although it would have at least total $\Theta(n^3)$ total step complexity.

Later work analyzed *adaptive* renaming algorithms, in which the step complexity and the size of the namespace depend only on total contention k, not on the maximum number of participating processes n. The first adaptive algorithm was introduced by Attiya and Fouren [19]. They achieve a namespace of $(6k-1)$ names, with a total complexity of $O(k^2 \log k)$. Afek and Merritt [7] build on the previous algorithm in order to achieve adaptive wait-free $(2k-1)$-renaming with total step complexity $O(k^3)$.

In a recent paper, Chlebus and Kowalski [13] improve the complexity bounds for deterministic renaming by providing a non-adaptive implementation with local step complexity roughly $O(\log n \log M)$, renaming into a namespace of size $O(n)$. The local step complexity of their algorithm is better than that of ReShuffle, although we achieve a tight namespace of n names, and comparable total step complexity. They also introduce an adaptive implementation with $O(k)$ local step complexity, which achieves renaming in $8k - \log k - 1$ names, and show the first non-trivial deterministic lower bound on step complexity, of $(1 + \min(k-2, \log_{2r} \frac{M}{2T}))$, where r is the number of shared registers used by the algorithm, and T is the size of the target namespace to rename into. One of the advantages of the algorithms from this reference is that they use little total memory $O(n \log(M/n))$. In comparison, our algorithms pre-allocate $O(n^2)$ memory, and use $O(n \text{ polylog } n)$ total memory, without assuming any bound M on the initial namespace.

Ellen et al. [20] analyze the complexity of *long-lived* adaptive renaming (i.e., processes may release their names) in shared-memory, under various synchrony assumptions. Their asynchronous algorithm ensures $\Theta(k)$ overhead for acquiring a new name, although assumes that stronger LL/SC primitives are available; hence their results are not directly comparable with ours. This reference also contains an excellent overview of prior work on renaming.

The feasibility of *randomized* renaming in an asynchronous system has been first considered by Panconesi et al. [21]. They present a wait-free solution that ensures a namespace of size $n(1+\epsilon)$ for $\epsilon > 0$, with expected $O(M \log^2 n)$ total step complexity, using only single-writer multiple-reader registers. This solution is shown to work against a strong adaptive adversary. Their strategy is similar to that of this paper: they

introduce a one-shot test-and-set implementation, and processes obtain names based on which test-and-set they manage to acquire. Note that their test-and-set implementation is not adaptive, which is why the complexity of the solution depends on M. Moreover, the namespace they obtain is not tight. Interestingly, a strategy similar to that of ReShuffle is mentioned in this reference (Section 4.1), but is considered "too hard to analyze." Note that our adaptive algorithm uses a different strategy than that of this reference, although the bounds on the namespace size look similar.

The second paper to analyze randomized renaming is by Eberly et al. [12]. The authors obtain a *tight* non-adaptive renaming algorithm based on the randomized wait-free implementation of test-and-set by Afek et al. [11]. Their algorithm is long-lived, and is shown to have amortized step complexity of $O(n \log n)$ per process. However, a simple analysis shows that their algorithm has average-case total step complexity of at least $\Theta(n^3)$, even if processes do not release their names.

4 An Adaptive Test-and-Set Implementation

We start by presenting an adaptive one-shot implementation of a randomized adaptive test-and-set object. The object exports a single Test-and-Set operation, whose sequential specification is provided in Figure 1.

Note that one-shot test-and-set cannot be implemented deterministically wait-free in asynchronous shared memory, since it has consensus number 2 (see [22] for details). We present an efficient randomized implementation that guarantees the desired properties with probability 1, and is linearizable, following the definition in [23]. Our implementation is *adaptive*, in that the complexity of an operation depends on the total contention k at the object, and not on n, the total number of processes.

4.1 The RatRace Algorithm

The RatRace implements the one-shot test-and-set object as defined above. Any operation on the object has step complexity $O(\log^2 k)$ per process with high probability, where k denotes the total contention at the object. The algorithm pre-allocates $O(n^3)$ memory, and uses $O(k)$ memory with high probability. A sketch of the algorithm's structure can be found in Figure 3.

Algorithm Structure. We begin from the randomized splitter object, as previously defined in [10]. Recall that the randomized splitter object is defined as follows: a process entering the splitter returns either *stop*, *left*, or *right*. If only one process enters the splitter, it is guaranteed to *stop*. If two or more processes enter the splitter, then zero or one processes *stop*, and the remaining processes each get a return value of *left* or *right*, independently and uniformly at random.

We build a binary tree of randomized splitters, of height $3 \log n$, which we call the *primary tree*. Each process starts the algorithm at the root splitter in the primary tree; if it does not manage to acquire the current splitter, it goes either left or right, each with probability $1/2$, until it manages to acquire a splitter. If a process reaches a leaf of

the primary tree without having acquired a splitter, it accesses a *backup grid*, which we describe below. To simplify the exposition, assume that, in this execution, all processes either obtained randomized splitters in the first tree, or crashed.

Once it managed to obtain a splitter, the process tries to work its way up back to the root, through a series of three-process "tournaments," one at each splitter node. Each splitter in the primary tree has associated with it a three-player "tournament," which is played between the owner of the splitter and the winners of the three-player test-and-sets corresponding to the two child nodes of the splitter. A three-player test-and-set is decided as follows: the two child nodes play each other, and the owner of the current splitter plays the winner of the first match. Each two-player match is decided using the randomized two-process test-and-set algorithm of Tromp and Vitànyi [16]. (Alternatively, we could use a randomized consensus algorithm with $n = 2$, e.g. [24], although the properties stay the same.) Note that the matches are decided in a wait-free manner, since a process wins automatically if the opponent does not show up.

The Backup Grid. The backup grid is an $n \times n$ grid of *deterministic* splitters, identical to that of Anderson and Moir [1], where the two children of a splitter are the splitter to its right, and the one below. Each process starts the backup algorithm at the top left splitter. As such, the structure guarantees that any correct process that accesses it eventually acquires a deterministic splitter. Just as in the previous case, once a process acquires a splitter, it tries to backtrack to the entry point through a series of three-player test-and-sets. The winner of the test-and-set at the entry splitter is also the winner of the backup grid.

Decision. The winner of the three-player test-and-set at the root of the primary tree plays the winner of the entry splitter in the backup grid. The winner of this last match returns winner. Every process that loses in a three-player test-and-set returns loser.

Linearization. In order to maintain the linearization guarantees of the test-and-set object, a process that loses a three-player test-and-set writes true to a multi-writer-multi-reader $Resolved$ register associated with the root of the primary tree, *before* returning loser. Processes read the register as the first step in their Test-and-Set invocation: if they read true, they automatically return loser.

4.2 Analysis of the RatRace Algorithm

It is relatively straightforward to check that the RatRace algorithm guarantees the correctness properties of the *test-and-set* object as stated in Section 4, therefore we omit the proof from this extended abstract. Termination with probability 1 is ensured since we use wait-free elements and the two-process test-and-set algorithm of [16], which terminates with probability 1. We next focus on the linearizability of the implementation, and on its performance in terms of total step complexity. Our first result shows that our implementation is linearizable, in the sense of Herlihy and Wing [23]. The proof is based on the observation that, before a loser indication is returned by RatRace, a potential winner has to take at least one step in the algorithm.

```
1 Variable:
2 Value, a binary MWMR atomic
  register, initially ⊥
3 procedure Test-and-Set()
4   if Value = ⊥ then
5       Value ← 1
6       return winner
7   else
8       return loser
```

Fig. 1. Sequential specification of a one-shot test-and-set object

```
1 Shared:
2 TS[], a vector of n RatRace objects
3 procedure rename( n )
4   List ← {1, 2, ..., n}
5   while true do
6       try ← element uniformly at
        random from List
7       res ← TS[try].test-and-set()
8       if res ← winner then return try
9       else List ← List \ {try}
```

Fig. 2. The ReShuffle algorithm

Lemma 1 (Linearization). *The* RatRace *algorithm is* linearizable*: for every execution of* RatRace*, there exists a total order over all the complete* Test-and-Set *operations together with a subset of the incomplete* Test-and-Set *operations such that every operation is immediately (atomically) followed by a response, and the sequence of operations given by that total order is consistent with a sequential execution of a test-and-set object, i.e. the order respects the real-time order of non-overlapping operations.*

We now analyze the performance of RatRace. Let k denote the number of processes that enter the RatRace in an execution $\mathcal{E}$, i.e. the total contention. The next result states that, with high probability, every process acquires a splitter in the primary tree. As a consequence of this fact, for the rest of the performance analysis, we will assume that all processes acquire nodes in the primary tree, since the backup case is extremely unlikely. We provide the intuition for why this holds; an exact proof follows from the analysis in [10], Lemma 8.

Lemma 2. *The probability that there exists a process p that does not acquire a randomized splitter in the primary tree of the* RatRace *object is at most $1/n$.*

Proof (Sketch). Let q be a process that does not manage to acquire any splitter in the primary tree. Hence, q did not manage to acquire the leaf splitter it reached. Since a process always acquires a splitter if it accesses it alone, this implies that another process q' accessed the same leaf splitter. However, the leaf splitter is accessed by q as a consequence of $3 \log n$ random choices of bits. Hence process q' must have performed the exact same random choices. Since the choices are independent, the probability that this occurs is $(1/2)^{3 \log n} = 1/n^3$. Hence the probability that there exists a process that performs exactly the same random choices as q is at most $1/n^2$. By the union bound, it follows that the probability that there exists a process p that "falls off" the primary tree is at most $1/n$. □

Let the *active primary tree* denote the minimum subtree of the primary tree containing all splitters that are acquired in the execution. The second result bounds the number of nodes in the active primary tree, and shows that the tree is well balanced, with high probability. The proof is similar to that of Lemma 2. For a complete argument, please

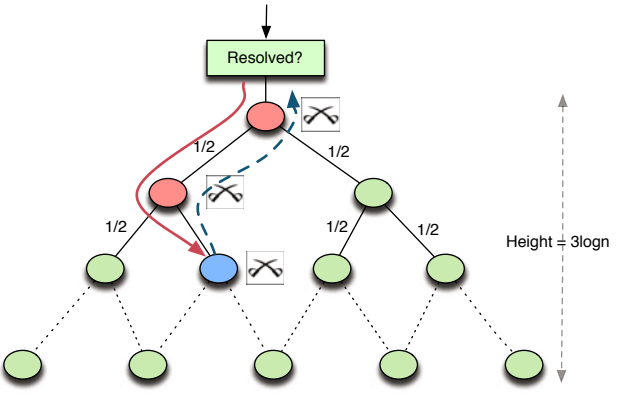


Fig. 3. Structure of the RatRace protocol. A process first checks the *Resolved* register, and then walks down the randomized splitter tree trying to acquire a splitter. In this figure, the process followed the solid path and acquired a splitter. Next, the process works its way back up the tree, participating in three-player tournaments at each node (the dotted path). If it loses along this path, then it marks the *Resolved* register and returns loser. Otherwise, if it wins the root tournament, it plays the winner from the backup grid (not shown here). The winner of this last match returns winner.

see reference [10], Lemma 11. Note that this lemma also bounds the space complexity the primary tree.

Lemma 3. *The number of nodes in the* active primary tree *is at most* $7k$*, and its height is at most* $3 \log k$*, both with high probability.*

Next, we look at the read-write complexity of the two-process test-and-set algorithm of Tromp and Vitányi [16] that we use to decide the two-process games. The following bounds follow from an analysis of the algorithm.

Lemma 4. *The randomized two-process test-and-set algorithm of [16] has expected constant read-write complexity, and performs less than* $\alpha \log k$ *reads and writes with high probability, for a constant* $\alpha > 1$.

Proof (Sketch). Please recall that the algorithm of [16] is composed of asynchronous "rounds" of computation, and performs a constant number, say β, of reads and writes per round. In every round, the probability of success is $1/2$. Thus, the expected step complexity is constant. The probability that the algorithm performs at least $2\beta \log k$ total steps is at most $1/k^{2\beta}$, from which the claim follows. □

The next result analyzes the total step complexity of RatRace.

Lemma 5. *The* RatRace *algorithm uses* $O(\log^2 k)$ *steps per process, with high probability. Hence, the total step complexity is* $O(k \log^2 k)$*, with high probability.*

Proof. Without loss of generality, we analyze the number of steps performed by a winning process. First note that, by Lemma 2, it is enough to bound the complexity in the case where the process only accesses the primary tree. By Lemma 3, a process

performs $O(\log k)$ steps, with high probability, when going down the tree in order to acquire a randomized splitter, since each splitter has constant step complexity. When climbing back up, the process may play up to $O(\log k)$ three-player test-and set games. By Lemma 4, we obtain that the process performs up to $O(\log^2 k)$ steps, with high probability. □

5 A Randomized Algorithm for Tight Renaming

In this section, we present ReShuffle, a randomized algorithm which ensures tight renaming using $O(n \log^4 n)$ total steps, with high probability. The pseudocode of the algorithm can be found in Figure 2.

5.1 The ReShuffle Algorithm

The n processes share n test-and-set objects, each implemented using the RatRace algorithm. These shared objects are numbered from 1 to n. Computation proceeds in local phases. In each phase, the process chooses uniformly at random a test-and-set from 1 to n that it has not chosen previously, and competes in it. If the process wins the test-and-set (i.e., the chosen RatRace instance returns winner), then it takes the number associated with the test-and-set as a name and returns. Otherwise, if it lost the test-and-set, the process marks the current test-and-set as lost, and tries again in the next phase.

5.2 Analysis of ReShuffle

In this section, we analyze the correctness of the algorithm and its performance guarantees. First, note that name uniqueness is satisfied trivially, since a process stops after it has won its first test-and-set, and no two processes may win the same test-and-set object. We first show termination with probability 1. The proof is based on the observation that if a process accesses all n test-and-set objects, it will certainly win one of them, and hence terminate. The latter claim is based on the linearizability of our test-and-set implementation.

Lemma 6 (Termination). *With probability* 1*, each correct process eventually returns from* ReShuffle*.*

The next Theorem provides precise bounds for the total step complexity of ReShuffle. This is the main technical result of this paper. Due to space restrictions, we only provide a detailed sketch of the proof in this extended abstract.

Theorem 1 (Complexity). *The total step complexity of* ReShuffle *is* $O(n \log^4 n)$ *with high probability.*

Proof (Sketch). The first idea in the proof is to consider the total number of Test-and-Set calls (or accesses) that the processes perform as part of ReShuffle. We will consider all the accesses in their *linearization order* over all n test-and-set objects. Note that such an order exists, and is coherent at each object, because each test-and-set object

is linearizable (by Lemma 1), and thus the objects are composable (or *local* [23]). We will show that the algorithm performs $O(n \log^2 n)$ total accesses in any execution, with high probability. To simplify the exposition, we modify the algorithm so that processes always pick the next test-and-set to access uniformly at random, without discarding test-and-set objects that have been accessed previously. ReShuffle can be seen as a slightly more efficient version of this scheme, in which a process receives immediately a loser indication if its random choice indicates a test-and-set object that it has accessed before.

Fix a constant $\alpha > 4$. We show that if the algorithm performs more than $\alpha n \log^2 n$ total Test-and-Set accesses during an execution, then, with high probability, each test-and-set object is accessed at least once. Since every such test-and-set object will have a unique, distinct winner, we conclude that the algorithm terminates after $\alpha n \log^2 n$ total accesses, with high probability.

A tempting, yet unsuccessful approach to bound the total number of calls before each test-and-set is accessed once would be to use the well-known *coupon collector* process [25, 26], which guarantees that n distinct coupons will be discovered using $O(n \log n)$ independent random trials. Note, however, that the strong adversary controls the scheduling of the trials, which causes this simple version of the analysis to fail. Our analysis takes this factor into account, and proves that, even though the adversary may re-order calls using its knowledge of the processes' random choices, all the objects are accessed after $O(n \log^2 n)$ random calls.

Let U to be the number of test-and-set objects that have not been accessed by any process, at a certain point in the execution. We split the execution into phases. For $1 \leq i \leq \log n$, we define phase i as the time interval in which $n/2^{i-1} \geq U > n/2^i$. (Recall that we consider the linearized execution.) We prove that, by performing $\alpha n \log n$ total test-and-set accesses, the algorithm progresses for at least one phase, with high probability. Also, the number of processes that take steps in phase i or later is at most $n/2^i$, with high probability.

We proceed by induction. In this sketch of proof, we only consider the induction step (the base case is similar). Assume that the claim holds at all phases $\leq i$, and we prove that it also holds at phase $i + 1$. First note that, if the adversary schedules at most $\alpha n \log n$ processes to access test-and-set objects during phase $i + 1$, then the processes will make at most $n/2^i + \alpha n \log n$ total random choices during this phase. This is because, by the induction step, there are at most $n/2^i$ processes that have not terminated up to phase $i + 1$, and each of them might make a random choice in this phase prior to accessing a test-and-set object. Also, for every access of a test-and-set that the adversary schedules, at most one more choice is made.

We will show that, since these choices are uniformly random, it is extremely improbable that the adversary finds $\alpha n \log n$ random choices made in this phase, which it can schedule without allowing the algorithm to move to the next phase. Let D_i be the set of test-and-set objects accessed prior to the beginning of phase $i + 1$. Notice that the algorithm stays in phase $i + 1$ after $\alpha n \log n$ total accesses if there exist 1) a set C of $\alpha n \log n$ random choices made during this phase, and 2) a set S of less than $n/2^{i+1}$ test-and-set objects not in D_i, such that all the choices in C are made on test-and-set objects from S, or on the $n(1 - 2^i)$ test-and-set objects in D_i. (Note that this

formulation slightly increases the power of the adversary by allowing it to "see" all the $(n/2^i + \alpha n \log n)$ random choices made in the phase when choosing the schedule.)

To bound the probability that the algorithm fails to move to phase $i+2$, we first *fix* a selection C of $\alpha n \log n$ random choices from this phase, and a set S of less than $n/2^{i+1}$ objects not in D_i. The probability that all the choices in C fall in S or in D_i is at most $\left(1 - 1/2^{i+1}\right)^{\alpha n \log n}$. Using the Bernoulli inequality, we obtain an upper bound of $(1/n)^{\alpha n/2^{i+1}}$ on this probability.

On the other hand, there are at most $2^{n/2^{i+1}}$ possible choices for the set S. Also, there are at most $\binom{n/2^i + \alpha n \log n}{\alpha n \log n}$ ways in which to select the set C. Using the union bound, after some calculation, we obtain that the probability that the algorithm stays in phase $i+1$ after $\alpha n \log n$ accesses is at most $(1/n)^{(\alpha-4)n/2^{i+1}}$. Since we analyze only the first $\log n$ phases, we obtain that the algorithm moves to phase $i+2$ with high probability for $1 \leq i \leq \log n$. This concludes the induction step for the first part of the claim.

For the second part, let $T_1, T_2, \ldots, T_{n/2^{i+1}}$ be $n/2^{i+1}$ test-and-set objects newly accessed by the algorithm in this phase, which were just shown to exist with high probability. From the properties of test-and-set, it follows that, for every T_j, $1 \leq j \leq n/2^{i+1}$, there exists a process q_j that accesses T_j, but never returns loser from it, i.e. either wins T_j or crashes in T_j. All q_j's must be distinct: a process stops taking steps after winning a test-and-set, and cannot crash in two test-and-sets. Since we consider the accesses in the linearization order, i.e. the winners are the first processes to return from the object, it follows that, with high probability, the processes q_j never take steps in the next phase, as required by the second part of the claim.

To conclude, notice that the second part of the claim proves that all processes return or crash by the end of phase $\log n$. This implies that the algorithm performs a total of $O(n \log^2 n)$ test-and-set accesses, with high probability, before each process terminates. A test-and-set access costs at most $O(\log^2 n)$ steps per process, since repeated accesses by the same process do not add to the complexity of the object. We obtain that the total step complexity is $O(n \log^4 n)$, with high probability. This concludes the proof of Theorem 1. □

6 A Randomized Adaptive Algorithm

In this section, we present a new adaptive randomized renaming algorithm, which we call AdaptiveSearch. Given a constant $\epsilon > 0$, the algorithm guarantees unique names, a namespace of size $\min(k(1+\epsilon), n)$ with high probability, and has $O(k \log^4 k / \log^2(1+\epsilon))$ total step complexity, with high probability.

6.1 The AdaptiveSearch Algorithm

As in the previous algorithm, each process p attempts to choose a name from a vector of n test-and-set objects. We assume that processes share n adaptive test-and-set objects, implemented through the RatRace algorithm, which are numbered from left to right. Since the contention is not known, each process starts with an estimate k_{est}

of contention, initially 1, which is increased as needed. Computation proceeds in local phases. In a phase, process p tries to win a randomly chosen test-and-set between 1 and k_{est} for $3 \log k_{est} / \log(1 + \epsilon/4)$ times. If it does not succeed by the end of a phase, then the process multiplies k_{est} by a constant factor $(1 + \epsilon/4) > 1$, and tries again in the next iteration. Once it succeeds in winning a test-and-set, the process takes the name associated with that test-and-set and returns. We enforce the name returned to be within a namespace of 1 to n as follows: once a process detects that k_{est} is larger than n, it starts to run the ReShuffle algorithm on the n test-and-set instances.

6.2 Analysis of AdaptiveSearch

In this section, we prove the correctness of AdaptiveSearch and its performance guarantees. Note that name uniqueness is ensured since no two processes may win the same test-and-set object. Also, AdaptiveSearch ensures termination with probability 1, since we run ReShuffle as a backup. Let k be the contention in the current execution. In this analysis, we assume that the namespace parameter ϵ is less than two (a similar argument holds for $\epsilon \geq 2$).
The first lemma provides an upper bound on the generated namespace with high probability.

Lemma 7 (Namespace). AdaptiveSearch *solves renaming in a namespace from* 1 *to* $k(1 + \epsilon)$*, with high probability. The maximum size of the namespace is* n*.*

Proof (Sketch). We consider a process p that obtains a name larger than $k(1 + \epsilon)$, and show that the probability that this occurs is very low. First, note that p's estimate of contention k_{est} when it obtained the name must have been at least $k(1 + \epsilon)$. Let k_ℓ be the last estimate on contention that process p tried which had the property that $k_\ell < k$. By definition, it follows that $k/(1+\epsilon/4) \leq k_\ell < k$. Since $\epsilon < 2$, we obtain that process p tried to obtain a random name in a namespace of size at least $1, 2, \ldots, k(1 + \epsilon/4)$ for at least $3 \log k(1 + \epsilon/4) / \log(1 + \epsilon/4)$ times, and did not succeed.

Next, we notice that, since at most k processes participate in the algorithm, there are at least $k\epsilon/4$ test-and-set objects that are not accessed throughout the entire execution. This follows since, for any test-and-set object that is accessed, there exists at least one process that either wins it or crashes while executing the object, and a process stops taking steps once it acquired a test-and-set. Therefore, irrespective of the adversarial schedule, each process has probability at least $(\epsilon/4)/(1 + \epsilon/4)$ to access a test-and-set that is not accessed by another process in the current execution. Also, in this case, the process stops accessing new test-and-set objects. We bound the probability that process p fails to acquire a name within a namespace of size at least $k\epsilon/4 + 1$ after $3 \log(k(1 + \epsilon/4)) / \log(1 + \epsilon/4)$ independent trials. After some calculation, we obtain that this probability is at most $(1/k)^3$. Therefore, with high probability, every process chooses a name between 1 and $k(1 + \epsilon)$. □

The next result provides an upper bound on the total step complexity of the algorithm.

Lemma 8 (Complexity). *The* AdaptiveSearch *algorithm takes* $O(k \log^4 k / \log^2(1 + \epsilon/4))$ *total steps with high probability.*

Proof. We analyze the total number of steps performed by a process p. By Lemma 7, every process runs for at most $\log_{1+\epsilon/4} k(1+\epsilon)$ local phases, with high probability. In each of these phases, the process performs at most $O(\log k(1+\epsilon)/\log(1+\epsilon/4))$ test-and-set accesses. In turn, each test-and-set is accessed by at most $O(k)$ distinct processes, which implies that one test-and-set access, implemented using the RatRace algorithm, will cost $O(\log^2 k)$ steps per process, with high probability. We thus obtain that the total step complexity is bounded by $O(k \log^4 k/\log^2(1+\epsilon/4))$. □

7 Future Work

Our algorithms outline a new approach for solving renaming efficiently in an asynchronous system. One direction for future work is to improve the local (per-process) step complexity of our algorithms, which may be super-linear in some executions of the ReShuffle algorithm (AdaptiveSearch has poly-logarithmic local step complexity, with high probability). This, together with a multiple-use version of RatRace, would allow our algorithms to be turned into efficient *long-lived* renaming algorithms. Another direction would be to study the lower bounds on the complexity of randomized renaming—we suspect that the lower bound threshold for total step complexity is super-linear. A third direction would be to study whether *tight adaptive* renaming can be achieved efficiently using randomization. It could also be interesting to study whether our approach may be applied to obtain efficient solutions to the well-known Do-All and Write-All (e.g., [27–29]) problems.

References

1. Anderson, J.H., Moir, M.: Using local-spin k-exclusion algorithms to improve wait-free object implementations. Distrib. Comput. 11(1), 1–20 (1997)
2. Moir, M., Anderson, J.H.: Fast, long-lived renaming (extended abstract). In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 141–155. Springer, Heidelberg (1994)
3. Moir, M., Garay, J.A.: Fast, long-lived renaming improved and simplified. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 287–303. Springer, Heidelberg (1996)
4. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM 46(2), 858–923 (1999)
5. Castañeda, A., Rajsbaum, S.: New combinatorial topology upper and lower bounds for renaming. In: PODC '08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 295–304. ACM, New York (2008)
6. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. Journal of the ACM 37(3), 524–548 (1990)
7. Afek, Y., Merritt, M.: Fast, wait-free (2k-1)-renaming. In: PODC '99: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 105–112. ACM, New York (1999)
8. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32, 374–382 (1985)
9. Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. J. ACM 55(5), 1–26 (2008)

10. Attiya, H., Kuhn, F., Plaxton, C.G., Wattenhofer, M., Wattenhofer, R.: Efficient adaptive collect using randomization. Distrib. Comput. 18(3), 179–188 (2006)
11. Afek, Y., Gafni, E., Tromp, J., Vitányi, P.M.B.: Wait-free test-and-set (extended abstract). In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 85–94. Springer, Heidelberg (1992)
12. Eberly, W., Higham, L., Warpechowska-Gruca, J.: Long-lived, fast, waitfree renaming with optimal name space and high throughput. In: Kutten, S. (ed.) DISC 1998. LNCS, vol. 1499, pp. 149–160. Springer, Heidelberg (1998)
13. Chlebus, B.S., Kowalski, D.R.: Asynchronous exclusive selection. In: PODC '08: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, pp. 375–384. ACM, New York (2008)
14. Alistarh, D., Attiya, H., Giurgiu, A., Gilbert, S., Guerraoui, R.: Fast randomized test-and-set and renaming, https://infoscience.epfl.ch/record/149943
15. Anderson, J.H., Kim, Y.-J.: Adaptive mutual exclusion with local spinning. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 29–43. Springer, Heidelberg (2000)
16. Tromp, J., Vitányi, P.: Randomized two-process wait-free test-and-set. Distrib. Comput. 15(3), 127–135 (2002)
17. Burns, J.E., Peterson, G.L.: The ambiguity of choosing. In: PODC '89: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, pp. 145–157. ACM, New York (1989)
18. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. In: PODC '93: Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, pp. 41–51. ACM, New York (1993)
19. Attiya, H., Fouren, A.: Adaptive and efficient algorithms for lattice agreement and renaming. SIAM J. Comput. 31(2), 642–664 (2001)
20. Brodsky, A., Ellen, F., Woelfel, P.: Fully-adaptive algorithms for long-lived renaming. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 413–427. Springer, Heidelberg (2006)
21. Panconesi, A., Papatriantafilou, M., Tsigas, P., Vitányi, P.M.B.: Randomized naming using wait-free shared variables. Distributed Computing 11(3), 113–124 (1998)
22. Herlihy, M.: Wait-free synchronization. ACM Trans. Programming Languages and Systems 13, 123–149 (1991)
23. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
24. Aspnes, J., Waarts, O.: Randomized consensus in expected $o(nlog^2n)$ operations per processor. SIAM J. Comput. 25(5), 1024–1044 (1996)
25. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, New York (2005)
26. Motwani, R., Raghavan, P.: Randomized algorithms. Cambridge University Press, New York (1995)
27. Georgiou, C., Russell, A., Shvartsman, A.A.: The complexity of synchronous iterative do-all with crashes. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 151–165. Springer, Heidelberg (2001)
28. Kowalski, D.R., Shvartsman, A.A.: Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. In: SPAA '04: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 311–320. ACM, New York (2004)
29. Georgiou, C., Shvartsman, A.A.: Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity. Springer, Heidelberg (2008)

Concurrent Computing and Shellable Complexes

Maurice Herlihy[1] and Sergio Rajsbaum[2]

[1] Brown University,
Computer Science Department,
Providence, RI 02912
mph@cs.brown.edu*
[2] Instituto de Matemáticas, Universidad Nacional Autónoma de México
Ciudad Universitaria, D.F. 04510
Mexico
rajsbaum@math.unam.mx**

Abstract. Roughly speaking, a simplicial complex is *shellable* if it can be constructed by gluing a sequence of n-simplexes to one another along $(n-1)$-faces only. Shellable complexes have been studied in the combinatorial topology literature because they have many nice properties.

It turns out that many standard models of concurrent computation can be captured either as shellable complexes, or as the simple union of shellable complexes. We consider general adversaries in the synchronous, asynchronous, and semi-synchronous message-passing models, as well as asynchronous shared memory augmented by consensus and set agreement objects.

We show how to exploit their common shellability structure to derive new and remarkably succinct tight (or nearly so) lower bounds on connectivity of protocol complexes and hence on solutions to the k-set agreement task in these models.

1 Introduction

For models of concurrent computation in which processes may fail by crashing, computations can be characterized as a *simplicial complex*, a geometric structure constructed by "gluing together" simplexes in a regular manner [14]. Informally, a complex is *k-connected* if it has no "holes" in dimension k or lower. It is known that if the complex corresponding to every such computation is k-connected, then one cannot solve $(k+1)$-set agreement [9,10,14].

Roughly speaking, a simplicial complex is *shellable* if it can be constructed by gluing a sequence of n-simplexes to one another along $(n-1)$-faces only. Shellable complexes have been studied in the combinatorial topology literature [2,16] because they have many nice combinatorial properties.

We show how to exploit these nice combinatorial properties of shellable complexes to derive new and remarkably succinct lower bounds both on the connectivity of the associated complexes, and on solutions to the k-set agreement task in these models, running against general adversaries.

* Supported by NSF 000830491.
** Supported by UNAM-PAPIIT.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 109–123, 2010.

We use a round-by-round model of computation, where processes act in a sequence of (sometimes asynchronous) rounds. The advantage of the round-by-round approach is that we can treat connectivity as an *invariant* established in the first round and preserved in later rounds. Understanding the computational power of a model splits into two tasks: analyzing the connectivity of the single-round complex (usually straightforward) and then analyzing how multiple rounds compose (usually not so straightforward).

The principal technical contribution of this paper, expressed in Theorems 4 and 5, is a proof that if the single-round complex is *shellable*, then multi-round compositions preserve connectivity under certain easily-checkable conditions. These are theorems of combinatorial topology, independent of any model of computation.

We then show that for many classical models of computation, each single-round complex is indeed shellable, so it becomes a straightforward exercise to derive tight (or nearly tight) bounds on when and if one can solve k-set agreement. For asynchronous shared-memory models in which processes have access to "black-box" objects that solve consensus or k-set agreement, matters are a little more complicated. We show that the one-round complex, while not itself shellable, is a simple union of shellable complexes, with a shellable nerve (explained below), and the same consequences follow. Moreover, our results apply not just to the usual wait-free or t-resilient failure models, but to general *adversary* schedulers that can cause certain subsets of processes to fail, perhaps in a non-uniform way.

Here is a summary of our contributions. We are the first to draw the connection between the topological notion of shellability and concurrent computation. This connection yields remarkably succinct lower bounds on the complexity (or possibility) of solutions to k-set agreement in a variety of models. While bounds are well-known for the wait-free and t-resilient failure models, we are the first to generalize these bounds to general adversaries[1].

Naturally, our results build on a long chain of predecessors. Combinatorial Topology was first used to analyze wait-free asynchronous shared-memory systems [4,14,18]. The round-by-round approach has been used before many times, in particular to unify and simplify the analysis of different models of computation [5,17]. Herlihy, Rajsbaum, and Tuttle [12,13] showed that connectivity arguments for message-passing models could be cast into a common framework. They introduced the notion of pseudospheres, and later proposed the notion of an *absorbing sequence*, a notion similar to but not identical with shellability. We extend that approach by encompassing both shared-memory models and general adversaries.

Junqueira and Marzullo [15] introduced the core/survivor-set formalism for characterizing general adversaries used here, and derived the first lower bounds for synchronous consensus against such an adversary. Delporte-Gallet et al. [7] were the first to prove several important lower bounds on k-set agreement in

[1] Except for the asynchronous shared read-write model, where we proved the same bound using model-specific techniques [11]

asynchronous shared memory against an adversary. Herlihy and Rajsbaum [11] gave the first direct application of combinatorial topology to the asynchronous read-write memory model against an adversary.

These results illustrate the power and continuing usefulness of topological methods for analyzing concurrent computation. Using a few well-known concepts from Combinatorial Topology, such as connectivity, nerves, and shellability, we have imposed a common framework on a collection of heretofore unrelated models of computation, resulting in remarkably succinct proofs, not only of known results in each of these models, but also of new, previously-unknown results that extend classical wait-free and t-resilient bounds to general adversaries.

Some have argued that while techniques from topology may be required as a foundation, most other results can be derived from simulations and reductions, such as the *BG-simulation* [6]. We disagree. The topological approach, with its powers of abstraction and vast armory of prior results, has shown that it can abstract away from model-dependent detail to provide a concise mathematical framework unifying many classical models. Simulations and reductions, however clever, are good at transforming the details of one particular model to another's, but have not, so far, demonstrated the same ability to reveal the "deep structure" common to these disparate models of computation.

2 Model

We consider systems where processes can fail by crashing. It is convenient to think of crashes as being controlled by an *adversary* that attempts to keep the protocol running for as long possible, perhaps forever. Adversaries are a natural way to extend the classical wait-free and t-faulty models of computation. Following Junqueira and Marzullo [15], we specify an adversary in terms its cores and survivor sets. A *core* C for adversary $\mathcal{A}$ is a set of processes such that (1) in every execution, some process in C does not fail, and (2) C is minimal: for every proper subset C' of C, there is an execution in which every process in C' fails. A *survivor set* S for adversary $\mathcal{A}$ is a set of processes such that (1) in some execution, the set of processes that do not fail is exactly S, and (2) S is minimal: for every proper subset S' of S, there is no execution in which the set of processes that do not fail is exactly S'. An adversary's cores and survivor sets are dual properties: each can be derived from the other.

An adversary is not uniquely characterized by its set of cores (or survivor sets). For example, Delporte-Gallet et al. [7] characterize an adversary by specifying *all* possible non-faulty sets, not just the minimal ones. For the range of models considered here, however, our results imply that the circumstances under which an adversary admits a protocol for k-set agreement (and hence for any colorless task [7,11]) are determined *only* by the adversary's minimum core size.

A *simplicial complex* is a finite set V along with a collection of subsets $\mathcal{K}$ of V closed under containment. An element of V is called a *vertex* of $\mathcal{K}$, and is usually denoted with vector notation: $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$. Each set in $\mathcal{K}$ is called a *simplex*, usually denoted by lower-case Greek letters: $\sigma, \tau, \ldots$. We sometimes abuse notation by

using σ to refer both to a simplex, and to the complex consisting of σ and its subsets. The *dimension* $\dim \sigma$ of a simplex σ is $|\sigma| - 1$. A maximal simplex in $\mathcal{K}$ is called a *facet* of $\mathcal{K}$, and *Facets*($\mathcal{K}$) denotes the set of facets of $\mathcal{K}$. A complex is *pure* if all its facets have the same dimension. For a pure complex $\mathcal{K}$ of dimension n, the *codimension* $\operatorname{codim} \sigma$ of a simplex $\sigma \in \mathcal{K}$ is $n - \dim \sigma$. We sometimes omit mention of $\mathcal{K}$ when it is clear from context.

We usually add one or more *labels* to vertexes, $\lambda : V \to D$, where D is an arbitrary domain. In particular, we have a set Π of process ids, and a label $id : V \to \Pi$ associating each vertex with a process id. Typically, each simplex is *well-colored* by these ids: if $\boldsymbol{u}, \boldsymbol{v} \in \sigma$ and $\boldsymbol{u} \neq \boldsymbol{v}$, then $id(\boldsymbol{u}) \neq id(\boldsymbol{v})$.

A simplex ρ is *between* two simplexes τ and σ if $\tau \subseteq \rho \subseteq \sigma$. We use $[\tau, \sigma]$ to denote the set of simplexes between τ and σ.

Protocols and tasks are modeled as follows. A vertex $\boldsymbol{v}$ is a pair consisting of a process ID and a process state, a k-dimensional simplex σ is a set of $k+1$ vertexes labeled with distinct process IDs and process states compatible in the sense there is some execution in which those processes end up with those states. Each possible initial state of the system is given by an *input simplex*, assigning an input value to each process. Together, all possible input simplexes make up the task's *input complex*. An *output simplex* and the task's *output complex* are defined similarly. A relation Δ carries each input simplex to a set of output simplexes. It associates with each initial state of the system the corresponding set of legal final states. Δ must be *order-preserving*: for $\sigma \in \mathcal{I}$, if $\sigma' \subseteq \sigma$, then $\Delta(\sigma') \subseteq \Delta(\sigma)$.

The duality of cores and survivor sets implies that if c is the minimum core size, and s the max survivor size size, then $c = n - s + 1$, implying that:

Lemma 1. *If $\mathcal{I}$ is an input complex and $\mathcal{A}$ an adversary with minimum core size c, then for any simplex $\sigma \in \mathcal{I}$ such that* $\operatorname{codim} \sigma < c$, *ids*($\sigma$) *contains a survivor set for $\mathcal{A}$.*

Any protocol has an associated *protocol complex* $\mathcal{P}$, in which each vertex is labeled with a process id and that process's final state (called its *view*). Each simplex thus corresponds to an equivalence class of executions that "look the same" to the processes at its vertexes. For $0 \leq m \leq n$, we understand $\mathcal{P}(\sigma)$ for a given input simplex σ, where *ids*(σ) contains a survivor set, to be the complex generated by all executions starting in σ in which only the processes in *ids*(σ) take part (the rest fail before taking any steps).

A complex $\mathcal{K}$ is *k-connected* if every continuous map of the k-sphere to $\mathcal{K}$ can be extended to a continuous map of the $(k+1)$-disk. (Informally, if it has no "holes" in dimensions k or less.) By convention, a complex is *(-1)-connected* if and only if it is nonempty, and every complex is *k-connected* for $k < -1$. Thus, a complex is 0-connected if it is path-connected. As a special case, a complex is *contractible* if it can be continuously deformed to a single point. A contractible complex is k-connected for all k. For our purposes, it suffices to know that a single simplex is contractible, as is any *cone*, a complex where every n-simplex shares a common vertex.

The following *Nerve Theorem* [3] is a powerful tool for establishing the connectivity of a complex from the connectivity of its components. Informally, it says that if we can decompose a complex into components that each has a "simple" topology, then the topology of the complex as a whole is determined by how the components intersect. More precisely, let $\mathcal{K}$ be a finite complex, I a finite index set, and $\{\mathcal{K}_i | i \in I\}$ a set of complexes that cover $\mathcal{K}$. The *nerve* $\mathcal{N}(\mathcal{K}_i | i \in I)$ is the complex with vertex set I, where a subset $J \subset I$ is a simplex if $\cap_{j \in J} \mathcal{K}_j$ is non-empty.

Theorem 1. *Suppose each intersection*

$$\mathcal{L}_J = \bigcap_{j \in J} \mathcal{K}_j$$

is either empty or $(k - |J| + 1)$-connected. Then $\mathcal{K}$ is k-connected if and only if the nerve $\mathcal{N}(\mathcal{K}_i | i \in I)$ is k-connected.

A *pseudosphere* [13] is a combinatorial structure in which each process from a set of processes is independently assigned a value from a set of values.

Definition 1. *Let $\sigma = (\boldsymbol{s}_0, \ldots, \boldsymbol{s}_m)$ be a simplex and $U_0, \ldots, U_m$ be a sequence of finite sets. In the* pseudosphere *complex $\psi(\sigma; U_0, \ldots, U_m)$, each vertex is a pair $\langle \boldsymbol{s}_i, u_i \rangle$, where $\boldsymbol{s}_i$ is a vertex of σ and $u_i \in U_i$. Vertexes $\langle \boldsymbol{s}_{i_0}, u_{i_0} \rangle, \ldots, \langle \boldsymbol{s}_{i_\ell}, u_{i_\ell} \rangle$ span a simplex if and only if the $\boldsymbol{s}_i$ are distinct. We use $\psi(\sigma; U)$ as shorthand for $\psi(\sigma; U, \ldots, U)$.*

A simplicial complex $\mathcal{X}$ is *shellable* [16, ch. 12] if its facets can be arranged in a linear order $\phi_0, \ldots, \phi_t$, called a *shelling order*, in such a way that the subcomplex $(\cup_{i=0}^{k-1} \phi_i) \cap \phi_k$ is pure and $(\dim \phi_k - 1)$-dimensional, for $0 < k \leq t$.

Theorem 2. *If $\mathcal{X}$ is shellable, and every facet has dimension at least k, then $\mathcal{X}$ is $(k-1)$-connected.*

The following alternative formulation for shellability [16, Prop. 12.2], while less concise, is easier to use in proofs. Let $\mathcal{K}$ be a simplicial complex with facets $\phi_0, \ldots, \phi_t$. This sequence is a shelling order if and only if, for $0 \leq i < j \leq t$, there exists ϕ_k, $0 \geq k < j$, such that $\phi_i \cap \phi_j \subseteq \phi_k \cap \phi_j$, and $|\phi_j \setminus \phi_k| = 1$.

Pseudospheres are perhaps the most basic examples of shellable complexes. Suppose we are given a pseudosphere $\psi(\sigma; U_0, \ldots, U_n)$, where $\sigma = \{\boldsymbol{s}_0, \ldots, \boldsymbol{s}_n\}$, and each U_i has a reflexive partial order $\preceq_i$. These partial orders $\preceq_i$ on the elements of each U_i induce a *canonical order* on pseudosphere facets as follows.

Definition 2. *First, we define a canonical partial order. For facets ϕ, ϕ' of $\psi(\sigma; U_0, \ldots, U_n)$, where*

$$\phi = \{\langle \boldsymbol{s}_0, u_0 \rangle, \ldots, \langle \boldsymbol{s}_n, u_n \rangle\}$$
$$\phi' = \{\langle \boldsymbol{s}_0, u'_0 \rangle, \ldots, \langle \boldsymbol{s}_n, u'_n \rangle\},$$

let $\phi \preceq \phi'$ if, for $0 \leq i \leq n$, $u_i \preceq_i u'_i$. The canonical order $\leq$ is any linear extension of $\preceq$.

Theorem 3. *Any pseudosphere $\psi(\sigma; U_0, \ldots, U_n)$ is shellable.*

3 Carrier Maps and Shellable Complexes

Definition 3. *A* carrier map $\mathcal{M} : \mathcal{C} \to \mathcal{D}$ *carries one pure complex to another by mapping each simplex in the domain* $\mathcal{C}$ *to a subcomplex of the codomain* $\mathcal{D}$*, such that*

$$\text{For all } \sigma, \sigma' \in \mathcal{C}, \quad \mathcal{M}(\sigma) \cap \mathcal{M}(\sigma') = \mathcal{M}(\sigma \cap \sigma') \tag{1}$$

We decompose computations into a sequence of *rounds*. Each round is defined by a carrier map that carries the complex representing the system state before the round to the complex after the round. Our impossibility results require showing that the carrier map preserves a particular level of connectivity. When we examine a round, we do not need to consider all executions permitted by the model, only the "worst-case" ones that preserve connectivity.

Informally, each model of computation defines its own carrier map. Formally, since each such map has a distinct range and domain, it is convenient to treat them below as separate maps.

Theorem 4. *Let* $\mathcal{M} : \mathcal{X} \to \dagger$ *be a carrier map such that for each simplex* $\sigma \in \mathcal{X}$*,* $\mathcal{M}(\sigma)$ *is* $(\ell - \operatorname{codim} \sigma - 1)$*-connected. If* $\mathcal{X}$ *is shellable with facets of dimension at least* n*, then* $\mathcal{M}(\mathcal{X})$ *is* $(\ell - 1)$*-connected.*

Proof. Because $\mathcal{X}$ is shellable, $\mathcal{X} = \cup_{i=0}^{t} \phi_i$, where $\phi_0, \ldots, \phi_t$ is a shelling order on the facets of $\mathcal{X}$. To show that $\mathcal{M}(\mathcal{X}) = \cup_{i=0}^{t} \mathcal{M}(\phi_i)$ is $(\ell - 1)$-connected, we argue by induction on t.

For the base case for t, because ϕ_0 is a facet of $\mathcal{X}$ of codimension 0, $\mathcal{M}(\phi_0)$ is $(\ell - 1)$-connected.

For the induction step for t, let

$$\mathcal{K} = \bigcup_{i=0}^{t-1} \mathcal{M}(\phi_i), \text{ and } \mathcal{L} = \mathcal{M}(\phi_t),$$

and assume $\mathcal{K}$ is $(\ell - 1)$-connected. Because ϕ_t is a facet of $\mathcal{X}$, $\mathcal{L} = \mathcal{M}(\phi_t)$ is $(\ell - 1)$-connected by the induction hypothesis for ℓ.

By the properties of shellable complexes,

$$\begin{aligned} \mathcal{K} \cap \mathcal{L} &= \left(\bigcup_{i=0}^{t-1} \mathcal{M}(\phi_i) \right) \cap \mathcal{M}(\phi_t) = \bigcup_{i=0}^{t-1} (\mathcal{M}(\phi_i) \cap \mathcal{M}(\phi_t)) \\ &= \bigcup_{i=0}^{t-1} \mathcal{M}(\phi_i \cap \phi_t) = \bigcup_{i \in I} \mathcal{M}(\tilde{\phi}_i) \end{aligned}$$

where $\left\{ \tilde{\phi}_i | i \in I \right\}$ is a set of $(\dim \phi_k - 1)$-faces of ϕ_t for some non-empty index set I. Each $\tilde{\phi}_i$ has codimension 1 in $\mathcal{X}$.

We now use the Nerve Theorem to compute the connectivity of $\mathcal{K} \cap \mathcal{L}$. Because the $\tilde{\phi}_j$ are faces of ϕ_t,

$$\bigcap_{j \in J} \mathcal{M}(\tilde{\phi}_j) = \mathcal{M}(\bigcap_{j \in J} \tilde{\phi}_i) = \mathcal{M}(\tilde{\phi}_J)$$

where $\tilde{\phi}_J$ is a $(\dim \phi_t - |J|)$-dimensional face of ϕ_t. It has codimension $|J|$ in $\mathcal{C}_{\ell+1}$, so $\mathcal{M}(\tilde{\phi}_J)$ is $(\ell - |J| - 1)$-connected.

To apply the Nerve Theorem, note that each $\mathcal{M}(\tilde{\phi}_J)$ is $((k-1) - |J| + 1)$-connected. The $\mathcal{M}(\tilde{\phi}_j)$ subcomplexes cover $\mathcal{K} \cap \mathcal{L}$, and the intersection of any k is non-empty. It follows that their nerve has $|I|$ vertexes, of which any k form a simplex, so the nerve contains the $(k-1)$-skeleton of an $|I|$-simplex, which is $(k-2)$-connected. It follows from the Nerve Theorem that $\mathcal{K} \cap \mathcal{L}$ is $(k-2)$-connected.

Since $\mathcal{K}$ and $\mathcal{L}$ are $(k-1)$-connected, and their intersection is $(k-2)$-connected, their union $\mathcal{K} \cup \mathcal{L} = \mathcal{M}(\mathcal{X})$ is $(k-1)$-connected by a further application of the Nerve Theorem.

Theorem 5. *Consider a sequence of complexes and carrier maps*

$$\mathcal{C}_0 \xrightarrow{\mathcal{R}_0} \mathcal{C}_1 \xrightarrow{\mathcal{R}_1} \cdots \xrightarrow{\mathcal{R}_r} \mathcal{C}_{r+1}.$$

If, for all $0 \le i \le r$, and $\sigma \in \mathcal{C}_i$, $\mathcal{R}_i(\sigma)$ is shellable with facets of dimension at least $(k - \operatorname{codim} \sigma)$, then the composition $\mathcal{R}_r(\cdots \mathcal{R}_0(\sigma) \cdots)$ is $(k - \operatorname{codim} \sigma - 1)$-connected.

Proof. For brevity, let $\mathcal{R}^*_\ell(\cdot)$ denote the composition $\mathcal{R}_\ell(\cdots \mathcal{R}_r(\cdot) \cdots)$, for $1 \le \ell \le r$.

We argue by reverse induction on r. By hypothesis, for every $\sigma \in \mathcal{C}_r$, $\mathcal{R}_r(\sigma)$ is a shellable complex with facets of dimension at least $(k - \operatorname{codim} \sigma)$, and is therefore $(k - \operatorname{codim} \sigma - 1)$-connected by Theorem 2.

For the induction step, assume that for every $\tau \in \mathcal{C}_{\ell+1}$, $\mathcal{R}^*_{\ell+1}(\tau)$ is $(k - \operatorname{codim} \tau - 1)$-connected. From Theorem 4, replacing $\mathcal{M}(\cdot)$ with $\mathcal{R}^*_{\ell+1}(\cdot)$, $\mathcal{A}$ with $\mathcal{R}_\ell(\sigma)$, and ℓ with $k - \operatorname{codim} \sigma$, it follows that $\mathcal{R}^*_\ell(\sigma)$ is $(k - \operatorname{codim} \sigma - 1)$-connected.

4 Asynchronous Message-Passing

We consider a set of $n+1$ asynchronous, crash-prone processes that communicate by sending messages. In each round, each process sends its state to every other process, waits until it has received messages sent in that round from a survivor set, and undergoes a state transition. Because the model is asynchronous, a message m sent from p to q in round r may not be delivered in that round. Messages are delivered in FIFO order, meaning that when m is delivered, all previously undelivered messages sent from p to q are delivered at the same time.

Let $\mathcal{A}$ be an adversary with minimal core size c. Denote the set of faces of σ of codimension less than c by

$$F_{c-1}(\sigma) = \{\tau | \tau \subseteq \sigma \text{ and } |\sigma \setminus \tau| < c\}.$$

Here is the one-round carrier map for this model:

$$\mathcal{R}_a(\sigma) = \begin{cases} \psi(\sigma; F_{c-1}(\sigma)) & \text{if } \sigma \text{ contains a survivor set, and} \\ \emptyset & \text{otherwise.} \end{cases}$$

Theorem 6. *For any input simplex σ containing a survivor set, $\mathcal{R}_a(\sigma)$ is shellable with facets of dimension at least $(c - \operatorname{codim}\sigma - 1)$.*

Proof. This property is non-trivial when $\operatorname{codim}\sigma < c$, and $\mathcal{R}_a(\sigma)$ is a pseudosphere, which is shellable by Theorem 3. Because it is a pseudosphere over σ, each facet has dimension $\dim\sigma$. Because $n \geq c-1$, $\dim\sigma \geq c - \operatorname{codim}\sigma - 1$.

Corollary 1. *For $r > 0$, the r-round protocol complex starting from any input simplex is $(c-2)$-connected.*

Corollary 2. *The adversary $\mathcal{A}$ does not admit an asynchronous message-passing protocol for $(c-1)$-set agreement.*

The adversary $\mathcal{A}$ does admit an asynchronous message-passing protocol for c-set agreement: pick a core C of size c, have them broadcast their values, and every process waits until it hears a value from a process in C.

5 Synchronous Message-Passing

We consider a set of $n+1$ synchronous, crash-prone processes that communicate by sending messages. In each round, each process sends its state to every other process. If a process crashes, only a subset of its messages may be delivered in that round. The adversary $\mathcal{A}$ determines which sets of processes can crash. As before, let c be the size of a minimal core for $\mathcal{A}$.

We wish to derive a lower bound on the number of rounds needed to solve k-set agreement against $\mathcal{A}$ in this model. We know that k-set agreement is impossible while the protocol complex is $(k-1)$-connected, so the question can be rephrased: for how many rounds can $\mathcal{A}$ keep the protocol complex $(k-1)$-connected? We assume that $n > 2k$.

We consider sets of executions in which exactly $k < c$ processes fail in each round. By Lemma 1, such executions are all permitted by $\mathcal{A}$.

Let $G_k(\sigma) = \{\tau | \tau \subseteq \sigma \text{ and } \dim\tau = n-k\}$, the set of faces of σ of dimension $n-k$. Here is the one-round carrier map for this model:

$$\mathcal{R}_s(\sigma) = \begin{cases} \bigcup_{\tau \in G_f(\sigma)} \psi(\tau, [\tau, \sigma]) & \text{if } \operatorname{codim}\sigma \leq k, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

We order the facets of $\mathcal{R}_s(\sigma)$ as follows. First, order faces of σ by reverse inclusion: if $\alpha \supset \beta$, then $\alpha \prec \beta$. Order facets of single pseudosphere $\psi(\tau, [\tau, \sigma])$ using the canonical order induced by the reverse inclusion order on the labels.

Second, order the simplexes in $G_k(\sigma)$. Each simplex τ in $G_k(\sigma)$ is labeled with $n-k+1$ out of $(\dim\sigma + 1)$ IDs. Construct a *signature* for each τ in $G_k(\sigma)$ by concatenating its process IDs in ascending order. Index the simplexes $\tau_0, \ldots, \tau_t$ of $G_k(\sigma)$ in increasing lexicographical order by signature.

Finally, order facets of $\psi(\tau_i, [\tau_i, \sigma])$ before facets of $\psi(\tau_j, [\tau_i, \sigma])$ if $i < j$. Let $\phi_0, \ldots, \phi_t$ be the resulting order.

Theorem 7. *For any input simplex σ containing a survivor set, $\mathcal{R}_s(\sigma)$ is shellable with facets of dimension at least $(k - \operatorname{codim}\sigma)$.*

Proof. All facets of $\mathcal{R}_s(\sigma)$ have dimension $n-k$, which is at least $(k - \operatorname{codim}\sigma)$ if $n > 2k$.

If ϕ_i and ϕ_j are facets where $i < j$, we need to construct a ϕ_k, $k < j$, such that $\phi_i \cap \phi_j \subseteq \phi_k \cap \phi_j$, and $|\phi_j \setminus \phi_j| = 1$. If ϕ_i and ϕ_j are facets of the same pseudosphere, then we use the construction from Theorem 3.

Suppose ϕ_i and ϕ_j are facets of different pseudospheres, respectively $\psi(\tau_i, [\tau_i, \sigma])$ and $\psi(\tau_j, [\tau_j, \sigma])$.

Suppose $\phi_j \setminus \phi_i$ contains a vertex $\langle \boldsymbol{t}_\ell, \sigma_\ell \rangle$ where σ_ℓ is a proper face of σ. Relabel that simplex with σ:

$$\phi_k = (\phi_j \setminus \{\langle \boldsymbol{t}_\ell, \sigma_\ell \rangle\}) \cup \{(\boldsymbol{t}_\ell, \sigma)\}).$$

It is easy to check that ϕ_k satisfies the shellability conditions.

Finally, suppose every vertex in $\phi_j \setminus \phi_i$ is labeled with σ. Because $i < j$, $\tau_i \setminus \tau_j$ contains a vertex $\boldsymbol{t}_\ell$ and $\tau_j \setminus \tau_i$ contains a vertex $\boldsymbol{t}_m$, such that $id(\boldsymbol{t}_\ell) < id(\boldsymbol{t}_m)$. Replace these vertexes in τ_j and ϕ_j:

$$\tau_k = (\tau_j \setminus \{\boldsymbol{t}_\ell\}) \cup \{\boldsymbol{t}_m\} \text{ and } \phi_k = (\phi_j \setminus \{\langle \boldsymbol{t}_\ell, \sigma \rangle\}) \cup \{\langle \boldsymbol{s}_m, \sigma \rangle\}$$

Because every vertex in ϕ_k is labeled either with σ or a simplex containing $\tau_i \cup \tau_j$, ϕ_k is a facet of $\psi(\tau_k, [\tau_k, \sigma])$, and hence a facet of $\mathcal{R}_s(\sigma)$. It is easy to check that $\tau_k < \tau_j$, hence the condition $\phi_k < \phi_j$ is satisfied, as well as the rest of the shellability conditions.

Corollary 3. *For $r \leq \lfloor c - 1/k \rfloor$, the r-round protocol complex starting from any input simplex is k-connected.*

Corollary 4. *If $n \geq 2k$, the adversary $\mathcal{A}$ does not admit a synchronous message-passing protocol for k-set agreement in fewer than $\lfloor \frac{c-1}{k} \rfloor + 1$ rounds.*

This bound is tight: $\mathcal{A}$ does permit a synchronous message-passing protocol for k-set agreement that takes $\lceil c/k \rceil$ rounds. Divide the processes in a core C of minimal size c into k groups of size at most $\lceil c/k \rceil$. Each group performs consensus among its members, which takes $\lceil c/k \rceil$ rounds. The others wait for some member of C to complete its protocol, and chooses that value.

6 Asynchronous Shared Memory

We consider a set of $n+1$ asynchronous, crash-prone processes that communicate though a *two-dimensional* array $m[\cdot, \cdot]$, indexed by round number and by process ID. In round r, each P_i writes its state to $m[r, i]$, waits until the set of processes that have written to row r includes a survivor set, and then takes an *atomic snapshot* of that row. Gafni and Rajsbaum [8] show that this round-by-round model is equivalent to one in which process simply share a read-write memory.

Let σ be an n-simplex where each vertex is labeled with a distinct process ID. A *survivor chain* for σ is a sequence of faces of σ, $\sigma_0, \ldots, \sigma_k$ such that $ids(\sigma_0)$ contains a survivor set for $\mathcal{A}$, and $\sigma_0 \subset \cdots \subset \sigma_k = \sigma$. We denote the set of survivor chains for σ by $Chains(\sigma)$.

For a survivor chain $\overline{\sigma} = \sigma_0 \subset \cdots \subset \sigma_k$, and $P_i \in \Pi$, let $\overline{\sigma}_i$ be the suffix of $\overline{\sigma}$ of sets containing P_i. The round map is defined by:

$$\mathcal{R}_m(\sigma) = \bigcup_{\overline{\sigma} \in Chains(\sigma)} \psi(\sigma; \overline{\sigma}_0, \ldots, \overline{\sigma}_n).$$

Theorem 8. *For any input simplex σ containing a survivor set, $\mathcal{R}_m(\sigma)$ is shellable with facets of dimension at least $(c - \operatorname{codim}\sigma - 1)$.*

Proof. Each facet has dimension $\dim\sigma \geq c - \operatorname{codim}\sigma - 1$.

Let $\sigma = \{\boldsymbol{s}_0, \ldots, \boldsymbol{s}_m\}$. Index facets $\phi_i = \{\langle \boldsymbol{s}_0, \eta_0\rangle, \ldots, \langle \boldsymbol{s}_m, \eta_m\rangle\}$, and $\phi_j = \{\langle \boldsymbol{s}_0, \theta_0\rangle, \ldots, \langle \boldsymbol{s}_m, \theta_m\rangle\}$, in any total order such that if each $\eta_\ell \supseteq \theta_\ell$, for $0 \leq \ell \leq m$, then $i \leq j$.

Suppose $|\phi_j \setminus \phi_i|$ contains a vertex $\langle \boldsymbol{s}_\ell, \theta_\ell\rangle$ where $\theta_\ell \subset \sigma$. Construct ϕ_k by relabeling that vertex with σ. Note that $\phi_k \in \mathcal{R}_m(\sigma)$ because replacing any element of a survivor chain with σ is still a survivor chain. Every label of ϕ_k is a superset of the corresponding label of ϕ_j, so $k < j$, and $|\phi_j \setminus \phi_k| = 1$.

Corollary 5. *For $r > 0$, the r-round protocol complex starting from any input simplex is $(c-2)$-connected.*

Corollary 6. *The adversary $\mathcal{A}$ does not admit an asynchronous shared-memory protocol for $(c-1)$-set agreement.*

We show elsewhere [11] that $\mathcal{A}$ admits an asynchronous shared-memory protocol for c-set agreement.

7 Semi-synchronous Message-Passing

In this model, processes exchange messages, but the time between two consecutive process steps is at least c_1 and at most c_2, and the time to deliver a message is at most d. The values c_1, c_2, and d are known constants, and we define $C = c_2/c_1$. Failures are controlled by an adversary with minimal core size c. We wish to derive a lower bound on the time needed to solve k-set agreement against $\mathcal{A}$ in this model. For how long can $\mathcal{A}$ keep the protocol complex $(k-1)$-connected?

We first consider *fast* executions where each round takes exactly time d. All messages sent during a round are delivered at the very end of that round (at multiples of time d). All processes take steps in lock-step as quickly as possible (at multiples of time c_1). The interval between process steps is called a *microround*, and there are $\mu = \lfloor d/c_1 \rfloor$ microrounds per round. Each message is labeled with the microround in which it was sent. The *view* of a process P_i at the end of a round is a map $f : \Pi \to [0, \mu]$, where $f(P_j)$ is the microround of the last message

received from P_j (or 0 if no message was received). View f *dominates* view f', written $f \geq f'$, if for $0 \leq i \leq n$, $f(P_i) \geq f'(P_i)$.

We consider sets of executions in which exactly $k < c$ processes fail in each round. By Lemma 1, such executions are all permitted by $\mathcal{A}$.

A *failure pattern* is a function F mapping each process to the microround μ_j in which it fails, or to μ if it does not fail. At the end of the round, there are a number of views consistent with F, since a process P_j failing in microround μ_j will send its last message to P_i either in microround μ_j or $\mu_j - 1$. We define $[F]$ to be the set of possible views produced by F:

$$f(P_i) = \begin{cases} F(P_i) - 1 \text{ or } F(P_i) & \text{if } P_i \text{ fails} \\ \mu & \text{otherwise} \end{cases}$$

As before, let $G_k(\sigma) = \{\tau | \tau \subseteq \sigma \text{ and } \dim \tau = n - k\}$, the set of faces of σ of dimension $n - k$. For $\tau \subseteq \sigma$, let $FP(\tau)$ be the set of failure patterns in which the processes in τ do *not* fail.

Here is the one-round carrier map for this "fast" execution:

$$\mathcal{R}_{ss}(\sigma) = \begin{cases} \bigcup_{\tau \in G_f(\sigma)} \bigcup_{F \in FP(\tau)} \psi(\tau, [F]) & \text{if } \operatorname{codim} \sigma \leq k, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}$$

We order the facets of this complex as follows. First, we partially order the facets ϕ, ϕ' of each $\cup_{F \in FP(\tau)} \psi(\tau, [F])$. If, for every pair of matching vertexes $\langle \boldsymbol{t}_i, f \rangle \in \phi$ and $\langle \boldsymbol{t}_i, f' \rangle \in \phi'$, and every $P_i \in \Pi$, $f(P_i) \geq f'(P_i)$, then $\phi \preceq \phi'$.

Second, order the simplexes in $G_k(\sigma)$ as in Section 5. Finally, order facets of $\cup_{F \in FP(\tau_i)} \psi(\tau_i, [F])$ before facets of $\cup_{F \in FP(\tau_i)} \psi(\tau_i, [F])$ if $i < j$. Let $\phi_0, \ldots, \phi_t$ be the resulting order.

Theorem 9. *For any input simplex σ containing a survivor set, $\mathcal{R}_{ss}(\sigma)$ is shellable with facets of dimension at least $(k - \operatorname{codim} \sigma)$.*

Proof. Given ϕ_i, ϕ_j, $i < j$, we construct ϕ_k satisfying the shelling condition. A vertex $\langle \boldsymbol{t}_\ell, f_\ell \rangle \in \phi_j$ is *minimal* (*maximal*) at P_j if, for every vertex $\langle \boldsymbol{t}_m, f_m \rangle \in \phi_j$, $f_\ell(P_j) \leq f_m(P_j)$ ($f_\ell(P_j) \geq f_m(P_j)$). Because views in ϕ_j can have only two possible values for each P_j, each vertex is either minimal, maximal, or both.

If $\phi_j \setminus \phi_i$ contains a vertex $\langle \boldsymbol{t}_\ell, f_\ell \rangle$ minimal for some P_j, construct ϕ_k by replacing f_ℓ with f'_ℓ, which agrees with f_ℓ except that $f'_\ell(P_j) = f_\ell(P_j) + 1$. If no vertex in $\phi_j \setminus \phi_i$ is minimal for any P_j, then for each P_j, some vertex in $\phi_j \cap \phi_i$ has a view that assigns $f_\ell(P_j) - 1$ to P_j. It follows that for each P_j, every vertex in ϕ_i assigns either $f_\ell(P_j) - 1$ or $f_\ell(P_j) - 2$ to P_j.

If ϕ_i, ϕ_j are both elements of $\cup_{F \in FP(\tau)} \psi(\tau, [F])$, then $j < i$, a contradiction. Otherwise, suppose $\phi_i \in \psi(\tau_i, [F_i])$ and $\phi_j \in \psi(\tau_j, [F_j])$. Because $i < j$, $\tau_i \setminus \tau_j$ contains a vertex $\boldsymbol{t}_\ell$ and $\tau_j \setminus \tau_i$ contains a vertex $\boldsymbol{t}_m$, such that $id(\boldsymbol{t}_\ell) < id(\boldsymbol{t}_m)$. Replace these vertexes in τ_j and ϕ_j:

$$\tau_k = (\tau_j \setminus \{\boldsymbol{t}_\ell\}) \cup \{\boldsymbol{t}_m\}$$
$$\phi_k = (\phi_j \setminus \{\langle \boldsymbol{t}_\ell, [F_j] \rangle\}) \cup \{\langle \boldsymbol{s}_m, [F_j] \rangle\}$$

It is easy to check that the condition ϕ_k is a facet of $\psi(\tau_k, [\tau_k, \sigma])$ is satisfied, as well as the rest of the shellability conditions.

We can compose this single-round execution $\lfloor \frac{c-1}{k} \rfloor$ times before we use up our "failure budget."

Corollary 7. *For $r \leq \lfloor \frac{c-1}{k} \rfloor$, the r-round "fast" protocol complex starting from any input simplex is $(k-1)$-connected, hence it cannot solve k-set agreement.*

By a standard construction [13], an r-round "fast" execution that takes time rd can be "stretched" so that after time $rd + Cd - \epsilon$, there is one process that cannot decide.

Theorem 10. *The adversary $\mathcal{A}$ does not admit a k-set agreement protocol that runs in time less than $\left(\lfloor \frac{c-1}{k} \rfloor - 1\right) d + Cd$.*

Proof. By a standard construction [13], an r-round "fast" execution that takes time rd can be "stretched" as follows: at the end of round r, fail all processes but P, and run P as slowly as possible, (taking steps at multiples of time c_2). At time $rd + Cd$, P will time out, but at time $rd + Cd - \epsilon$, this execution is indistinguishable to P from the corresponding "fast" execution in which it is about to receive the next round of messages. The result follows by setting $r = \lfloor \frac{c-1}{k} \rfloor$.

This result is almost tight.

Theorem 11. *The adversary $\mathcal{A}$ admits a semisynchronous message-passing protocol for k-set agreement that runs in time $2 \lfloor \frac{c-1}{k} \rfloor d + Cd$.*

Proof. Divide a core C of minimal size c into k groups of size at most $\lceil c/k \rceil$. The members of each of these groups perform consensus among themselves, using the protocol of Attiya, Lynch, Dolev, and Stockmeyer [1], each of which takes time $2 \lfloor \frac{c-1}{k} \rfloor + Cd$. Every other process waits until some member of C completes its protocol, and chooses that value.

8 Asynchronous Set Agreement Memory

We now consider an asynchronous round-by-round model of computation, against a general adversary, in which the processes can solve k-set agreement (or consensus, if $k = 1$) in a single round.

For example, given an adversary $\mathcal{A}$ with a minimal core C of size c, it is enough for the members of C to solve k-set agreement wait-free in one round. For example, c-set agreement with read-write memory is trivial: each process in C writes its input to shared memory, and the rest wait and choose any such value. Given wait-free "black box" objects that permit m processes to solve ℓ-set agreement, dividing C into groups of m that share a single object yields a protocol for k-set agreement where $k = \ell \cdot \lfloor \frac{c}{m} \rfloor + \min(\ell, c \bmod m)$.

Theorem 5 can be generalized to encompass cases where the single-round complex is not itself shellable, but can be expressed as the union of shellable complexes having a shellable nerve. We assume $\dim \sigma > k$. Let $vals(\sigma)$ be the set of input values for input n-simplex σ, and $vals_k(\sigma)$ the set of subsets of $vals(\sigma)$ of size exactly k. For each set $K \in vals_k(\sigma)$, the set of executions in which the values chosen are a subset of K is $\mathcal{P}_K(\sigma) = \psi(\sigma, K)$. Here is the one-round carrier map for this model:

$$\mathcal{P}_k(\sigma) = \begin{cases} \bigcup_{K \in vals_k(\sigma)} \mathcal{P}_K(\sigma) & \text{if } \sigma \text{ contains a survivor set, and} \\ \emptyset & \text{otherwise.} \end{cases}$$

In general, this complex is not shellable, but it is covered by shellable subcomplexes, and this covering has a shellable nerve.

Let $\mathcal{P}_k^r(\cdot)$ denote the r-round composition of $\mathcal{P}_k(\cdot)$. For $K \in vals_k(\sigma)$, let

$$\mathcal{K} = \mathcal{P}_k(\sigma) \qquad \mathcal{K}_K = \mathcal{P}_K(\sigma) \qquad \mathcal{L}_J = \bigcap_{K \in J} \mathcal{K}_K$$

$$\mathcal{K}^r = \mathcal{P}_k^r(\sigma) \qquad \mathcal{K}_K^r = \mathcal{P}_k^{r-1}(\mathcal{K}_K) \qquad \mathcal{L}_J^r = \mathcal{P}_k^{r-1}(\mathcal{L}_J)$$

The $\mathcal{K}_K$ cover $\mathcal{K}$, and the $\mathcal{K}_K^r$ cover $\mathcal{K}^r$.

Lemma 2. *The nerve complexes $\mathcal{N}(\mathcal{K}_K | K \in vals_k(\sigma))$ and $\mathcal{N}(\mathcal{K}_K^r | K \in vals_k(\sigma))$ are isomorphic.*

Proof. It is enough to show that $\cap_{K \in J} \mathcal{K}_K \neq \emptyset$ if and only if $\cap_{K \in J} \mathcal{K}'_K \neq \emptyset$. The complex

$$\mathcal{L}_J = \bigcap_{K \in J} \mathcal{K}_K = \bigcap_{K \in J} \psi(\sigma, K) = \psi(\sigma, \bigcap_{K \in J} K)$$

is non-empty if and only if $\cap_{K \in J} K$ is non-empty. The complex

$$\mathcal{L}'_J = \bigcap_{K \in J} \mathcal{K}'_K \quad = \bigcap_{K \in J} \mathcal{P}_k^{r-1}(\mathcal{K}_K) = \mathcal{P}_k^{r-1}(\bigcap_{K \in J} \psi(\sigma, K)) = \mathcal{P}_k^{r-1}(\psi(\sigma, \bigcap_{K \in J} K))$$

is also non-empty if and only if $\cap_{K \in J} K$ is non-empty.

Lemma 3. *The nerve complex $\mathcal{N}(\psi(\sigma, K) | K \in vals_k(\sigma))$ is shellable.*

Proof. Order the subsets of $vals(\sigma)$ by sorting each set and comparing them lexicographically.

Each vertex of the nerve is associated with $\psi(\sigma, U)$, where $U \in vals_k(\sigma)$. Each edge is associated with

$$\psi(\sigma, U) \cap \psi(\sigma, V) = \psi(\sigma, U \cap V),$$

where $U \cap V$ is non-empty, and so on. It easier to reason about the barycentric subdivision of the nerve, where each facet has has the form $\phi = \{\tau^0, \ldots, \tau^{k-1}\}$, where $\tau^0 \subset \cdots \subset \tau^{k-1}$.

Here is the shelling order. Given simplexes $\phi_i = \{\tau_i^0, \ldots, \tau_i^{k-1}\}$, where $\tau_i^0 \subset \cdots \subset \tau_i^{k-1}$, $\phi_j = \{\tau_j^0, \ldots, \tau_j^{k-1}\}$, where $\tau_j^0 \subset \cdots \subset \tau_j^{k-1}$, we order them by top-down lexicographic order: $\phi_i < \phi_j$ if there exists $0 \leq q \leq k-1$ such that $\tau_i^q = \tau_j^q$ for $\ell < q \leq k-1$, and $\tau_i^\ell < \tau_j^\ell$.

Given $\phi_i < \phi_j$, we construct a ϕ_k satisfying the shellability conditions. Let ℓ be the largest index such that $\tau_i^\ell < \tau_j^\ell$. If $\ell = k-1$, the maximal value, then $\tau_j^\ell = \tau_j^{\ell-1} \cup \{y\}$, and there must exist $x < y$ such that $x \notin \tau_j^{\ell-1}$. Otherwise, τ_j^ℓ consists of the $k-1$ least elements of U, contradicting the assumption that $\tau_i^\ell < \tau_j^\ell$. Construct ϕ_k by replacing y with x:

$$\phi_k^q = \begin{cases} (\phi_j^{k-1} \setminus \{y\}) \cup \{x\} & \text{if } q = k-1 \\ \phi_j^q & \text{otherwise.} \end{cases}$$

Otherwise, if $\ell < k-1$,

$$\phi_j^{\ell+1} = \phi_i^{\ell+1} = \phi^{\ell-1} \cup \{y, z\}, \phi_j^\ell = \phi^{\ell-1} \cup \{z\}, \phi_i^\ell = \phi^{\ell-1} \cup \{y\}$$

where, because $\phi_i < \phi_j$, $y < z$. Construct ϕ_k by replacing z with y:

$$\phi_k^q = \begin{cases} (\phi_j^\ell \setminus \{z\}) \cup \{y\} & \text{if } q = \ell \\ \phi_j^q & \text{otherwise.} \end{cases}$$

In both cases, it is easy to check that $\phi_k < \phi_j$, $\phi_i \cap \phi_j \subset \phi_k$, and $|\phi_j \setminus \phi_k| = 1$.

Corollary 8. *The nerve complex $\mathcal{N}(\mathcal{K}_K^r | K \in vals_k(\sigma))$ is $(k-2)$-connected.*

Lemma 4. *If σ contains a survivor set, then for $r \geq 0$, $\mathcal{P}^r(\sigma)$ is $(k-2)$-connected.*

Proof. By induction on r. For the base case, when $r = 0$, $\mathcal{P}_k^r(\sigma) = \sigma$, which is contractible.

For the induction hypothesis, assume $\mathcal{P}^{r-1}(\sigma)$ is $(k-1)$-connected. For each $K \in vals_k(\sigma)$, $\mathcal{K}_K = \mathcal{P}_K(\sigma) = \psi(\sigma, K)$ is shellable with facets of dimension $\dim \sigma$, and by Theorem 4, each $\mathcal{K}'_K = \mathcal{P}^{r-1}(\mathcal{K}_K)$ is $(\dim \sigma - 1)$-connected. For each $J \subseteq vals_k(\sigma)$, $\mathcal{L}_J = \cap_{K \in J} \mathcal{K}_K = \psi(\sigma, \cap_{K \in J} K)$ is either shellable with facets of dimension $\dim \sigma$ or empty, and by Theorem 4, each $\mathcal{L}'_J = \cap_{K \in J} \mathcal{P}^{r-1}(\mathcal{P}_K(\sigma))$ is either $(\dim \sigma - 1)$-connected or empty, hence $((k-1) - |J| + 1)$-connected.

The Nerve Theorem implies that $\mathcal{P}^r(\sigma) = \cup_{K \in vals_k(\sigma)} \mathcal{K}'_K$ is $(k-1)$-connected if and only if the nerve complex $\mathcal{N}(\mathcal{K}'_K | K \in vals_k)$ is $(k-1)$-connected, which is true by Corollary 8.

Corollary 9. *$\mathcal{A}$ does not permit a $(k-1)$-set agreement protocol.*

Since $\mathcal{A}$ permits a single-round k-set agreement protocol by hypothesis, this result implies that the "set consensus power" of such a model is established in the first round.

References

1. Attiya, H., Dwork, C., Lynch, N., Stockmeyer, L.: Bounds on the time to reach agreement in the presence of timing uncertainty. J. ACM 41(1), 122–152 (1994)
2. Björner, A.: Shellable and Cohen-Macaulay partially ordered sets. Transactions of the American Mathematical Society 260(1), 159–183 (1980)
3. Björner, A.: Topological methods, pp. 1819–1872. MIT Press, Cambridge (1995)
4. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC '93: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pp. 91–100. ACM, New York (1993)
5. Borowsky, E., Gafni, E.: A simple algorithmically reasoned characterization of wait-free computations (extended abstract). In: PODC '97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 189–198. ACM, New York (1997)
6. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG distributed simulation algorithm. Distributed Computing 14(3), 127–146 (2001)
7. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Tielmann, A.: The disagreement power of an adversary: extended abstract. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 288–289. ACM, New York (2009)
8. Gafni, E., Rajsbaum, S.: Distributed programming with tasks. Technical Report 100001, UCLA Computer Science Department, Los Angeles, CA, USA, november (2009)
9. Herlihy, M., Rajsbaum, S.: Set consensus using arbitrary objects (preliminary version). In: PODC '94: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 324–333. ACM, New York (1994)
10. Herlihy, M., Rajsbaum, S.: Algebraic spans. Mathematical Structures in Computer Science 10(4), 549–573 (2000)
11. Herlihy, M., Rajsbaum, S.: The topology of shared-memory adversaries. In: PODC '10: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (to appear, 2010)
12. Herlihy, M., Rajsbaum, S., Tuttle, M.: An axiomatic approach to computing the connectivity of synchronous and asynchronous systems. Electron. Notes Theor. Comput. Sci. 230, 79–102 (2009)
13. Herlihy, M., Rajsbaum, S., Tuttle, M.R.: Unifying synchronous and asynchronous message-passing models. In: PODC '98: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, pp. 133–142. ACM, New York (1998)
14. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM 46(6), 858–923 (1999)
15. Junqueira, F., Marzullo, K.: Designing algorithms for dependent process failures. In: Future Directions in Distributed Computing, pp. 24–28 (2003)
16. Kozlov, D.: Combinatorial Algebraic Topology. Springer, Heidelberg (2007)
17. Moses, Y., Rajsbaum, S.: A layered analysis of consensus. SIAM J. Computing 31(4), 989–1021 (2002)
18. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000)

Brief Announcement: Hybrid Time-Based Transactional Memory*

Pascal Felber[2], Christof Fetzer[1], Patrick Marlier[2],
Martin Nowack[1], and Torvald Riegel[1]

[1] Technische Universität Dresden, Germany
first.last@inf.tu-dresden.de
[2] Université de Neuchâtel, Switzerland
first.last@unine.ch

Abstract. Transactional Memory (TM) is a speculative shared-memory synchronization mechanism used to speed up concurrent programs. Most current TM implementations are software-based (STM) and incur noticeable overheads for each transactional memory access. Hardware TM proposals (HTM) address this issue but typically suffer from other restrictions such as limits on the number of data locations that can be accessed in a transaction. In this paper, we introduce new *hybrid TM* algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms belong to the class of time-based TM designs and exploit the ability of some HTMs to have both transactional and non-transactional memory accesses within a transaction to decrease the transactions' runtime overhead, abort rates, and hardware capacity requirements.

Introduction. Current *software* transactional memory implementations have a relatively large performance overhead. While there is certainly room for further optimizations, it is believed by many that only hardware transactional memory (HTM) implementations can have a sufficiently good performance for TM to become widely adopted by developers.

Of the many published HTMs, designs such as AMD's Advanced Synchronization Facility (ASF) that have been proposed by industry for possible inclusion in high-volume microprocessors have low complexity and provide best-effort HTM in the sense that only a subset of all reasonable transactions are expected to be supported by hardware. They have several limitations (e. g., their capacity, that is the number of cache lines that can be accessed in a transaction, can be as low as four) and have to be complemented with software fallback solutions that execute in software the transactions that cannot run in hardware. It is therefore desirable to develop *hybrid* TM (HyTM) in which multiple hardware and software transactions can run concurrently.

In this paper, we introduce two novel HyTM algorithms that use ASF as HTM. These algorithms use two state-of-the-art time-based STM algorithms, LSA [3] and NOrec [2], for software transactions. LSA and NOrec focus on different workloads in their optimizations but are both lock-based designs. NOrec uses a single lock and provides better

* This work is supported in part by the European Commission FP7 VELOX project (ICT-216852).

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 124–126, 2010.

performance for low thread counts because it does not have to pay the runtime overheads associated with accessing multiple locks. In contrast, LSA uses an array of locks and is expected to provide better scalability with large thread counts or frequent but disjoint commits of software transactions. Therefore, both algorithms are of practical interest depending on the target architecture and workload.

Advanced Synchronization Facility. AMD's ASF is a proposal of hardware extensions for x86_64 CPUs. It essentially provides hardware support for the speculative execution of regions of code. These speculative regions are similar to transactions in that they take effect atomically. We have shown in previous work [1] that ASF can be used as an efficient pure HTM in a TM software stack. The HyTM algorithms that we introduce in this paper are based on ASF, use the same software stack, and provide the same guarantees as the respective STMs.

Most previous HyTM proposals have assumed HTMs in which every memory access inside a transaction is speculative, i. e., it is transactional, isolated from other threads until transaction commit, and will be rolled back on abort. In contrast, ASF uses selective annotation, which means that memory accesses inside transactions have to be explicitly marked as being speculative and are non-speculative otherwise. Selective annotation requires more work on the compiler side, but allows the TM to use the limited speculative accesses, and thus precious ASF capacity, only where necessary. We make heavy use of this feature to improve the efficiency of our HyTM algorithms by decreasing the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions.

Speculative accesses always have the granularity of a cache line. Non-speculative loads are allowed to read state that is speculatively updated in the same speculative region, but non-speculative stores must not overlap with previous speculative accesses.

The ordering guarantees that ASF provides for mixed speculative and non-speculative accesses are important for the correctness of our algorithms. In short, aborts are instantaneous with respect to the program order of instructions in speculative regions. A consequence is that memory lines are monitored early for conflicting accesses. Further, atomic instructions such as compare-and-set or fetch-and-increment retain their ordering guarantees.

HyLSA and HyNOrec Algorithms. LSA is a lock-based STM with time-based validation. To improve on previous HyTM extensions for this class of STMs, we wanted to decrease the number of memory locations that have to be read speculatively. This reduces the hardware capacity needed to successfully run transactions using ASF, and thus increases overall performance. In HyLSA, software transactions execute the LSA algorithm. Hardware transactions follow a similar algorithm with the following changes.

HyTM loads first perform an ASF-protected read of the lock associated with the memory location. This lets ASF immediately monitor the lock for subsequent changes and will lead to an abort if the lock is updated by another thread. If the lock is free, the transaction uses an ordinary nonspeculative load operation to read the target value.

HyTM stores proceed as loads, first monitoring the lock and verifying that it is not acquired. The transaction then starts monitoring the lock for writes and reads, which effectively ensures eager detection of conflicts with concurrent transactions. Finally, the updated memory location is speculatively written to.

Upon commit, a HyTM update transaction firsts acquires a unique commit timestamp from the global time base, speculatively writes all updated locks, and finally tries to commit the hardware transaction. ASF allows us to acquire the commit timestamp using a nonspeculative atomic-increment of a global counter, which makes this algorithm feasible because otherwise the speculative access to the counter would lead to an excessive number of aborts.

Our second HyTM is based on NOrec. Roughly speaking, NOrec differs from LSA in that it uses a single (global) versioned lock and relies on value-based validation in addition to time-based validation. The previous proposal [2] for a HyTM version of NOrec adds a second versioned lock that is used by software transactions to abort hardware transactions. We implemented this proposal but it did not perform well due to unnecessary aborts of hardware transactions. With the changes described next, performance and scalability increased significantly.

First, in the previous proposal, software transactions had to acquire the main and second locks atomically using a small hardware transaction. However, we can modify these locks separately and nonspeculatively. This both decreases the number of conflicts on these locations and avoids depending on guaranteed progress of hardware transactions in the STM part.

Second, hardware transactions are validated constantly because any update to speculatively accessed memory locations will abort the transaction. Thus, a hardware transaction does a conservative form of value-based validation, and is thus consistent whenever no (software) transaction is writing back its tentative updates. To guarantee consistency, HyTM load and store operations therefore only have to wait until the global lock is free. This second optimization allows for software transactions to commit without aborting nonconflicting hardware transactions (as in the previous proposal).

Evaluation. We evaluated our algorithms using a near-cycle-accurate simulator with eight x86 CPU cores on a single socket, extended to support ASF. We compiled benchmarks using the Dresden TM Compiler [1]. For each transaction, the compiler creates a hardware and a software TM code path, and the HyTM decides at runtime when starting or restarting a transaction which code path is to be executed. We compared pure HTM implementations with our two HyTM algorithms.

Our early evaluation results indicate that HTM performs best when its capacity is sufficient to handle all transactions. However, when capacity is not sufficient, HyTM algorithms are more efficient. As expected, HyNOrec has lower overhead than HyLSA, but the latter seems to scale better with large thread counts. These results are encouraging and indicate that ASF provides a sound basis for implementing efficient HyTM.

References

1. Christie, D., Chung, J.W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Riviere, E.: Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In: EuroSys 2010 (2010)
2. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: PPoPP 2010 (2010)
3. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: PPoPP 2008 (2008)

Brief Announcement: Quasi-Linearizability: Relaxed Consistency for Improved Concurrency

Yehuda Afek, Guy Korland, and Eitan Yanovsky

Computer Science Department
Tel-Aviv University, Israel
afek@cs.tau.ac.il, guy.korland@cs.tau.ac.il, eitanyan@post.tau.ac.il

Abstract. Many linearizable and optimized concurrent algorithms are available for known algorithms and data structures, such as, Queue, Tree, Stack, Counter and HashTable. However, sometimes these implementations are used in a more relaxed environment, provided as part of larger design pattern where a relaxed linearizability suffices rather than a strict one.

Here we provide a quantitative definition of limited non-determinism, a notion we call Quasi Linearizability. Roughly speaking an implementation of an object is quasi linearizable if each run of the implementation is at a bounded "distance" away from some linear run of the object.

Linearizability[2] is a useful and intuitive consistency correctness condition that is widely used to reason and prove common data structures implementations. Intuitively it requires each run to be equivalent in some sense to a serial run of the algorithm. This imposes strong synchronization requirements that in many cases result in limited scalability and synchronization bottlenecks. In order to overcome this limitation, more relaxed consistency conditions have been introduced. But, the semantics of these relaxed conditions is less intuitive and the results are usually unexpected from a layman point of view. In this paper we offer a relaxed version of linearizability that preserves some of the intuition, provides a flexible way to control the level of relaxation and supports the implementation of more concurrent and scalable data structures.

For example, SEDA[4], the motivating and initiating reason for the current research is a common design pattern for highly concurrent servers, which is heavily based on thread pools. Such thread pools are composed from two elements (i) a set of threads ready to serve tasks and (ii) a task queue from which the threads take their tasks. Such a queue, which is not part of the server logic in a highly concurrent system, can become by itself a bottleneck limiting the overall SEDA system utilization. However, thread pool does not really need a strict FIFO queue, what is really required is a "fair" queue that does not allow one task to by pass another by too much.

The above example as well as other examples have motivated us to provide a quantitative definition of the limited non-determinism that an application may allow. We define a consistency condition with an upper bound on the amount of non-determinism. Each operation must be linearizable at some bounded distance

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 127–129, 2010.

from its strict linearization point. For example, it is okay to dequeue t even if a task that has been enqueued $k-1$ places "before" it, has not yet been dequeued. Our definition is strong and flexible enough to define at the same time (continuing the above example) that a dequeue that returns empty may not be reodered, i.e., it has to be in its strict linearizable order.

We start by defining *quasi linearizable history* and then define what a *quasi-linearizable* data structure is.

Definition 1. – *For each event e in a sequential history H, we define $H[e]$ to be its index in the history.*
– *Let $Events(H)$ be the set of all the events in H.*
– *Let $Distance(H', H)$ be the distance between two histories H and H' which is a permutation of H, to be $\max_{e \in Events(H)}\{|H'[e] - H[e]|\}$.*
– *Let $Objects(H)$ be the set of all the objects that participate in H.*

Definition 2. Object domain *The set of all possible operations that may operate on the object. We distinguish between operations that have different arguments or returned results. For example for an object O =stack, $Domain(O) = \{< O.push(x), void >, < O.pop(), x > | x \in X\} \cup \{< O.pop(), \phi >\}$ (X is the set of all the possible elements in the stack).*

Definition 3. History domain *Let H be a history, $Domain(H) = \bigcup_{O \in Objects(H)} Domain(O)$.*

A sequential history $H|D$, is the projection of history H on a subset D of the events, i.e., H after removing from it all the events which are not in D. $H|O = H|Domain(O)$.

Definition 4. Q-Quasi Linearizable history *We need a way to measure the Quasi Linearizable property of a history, for that we use a function that we name the* quasi linearization factor*. It is a set of functions, each operating on a subset of a history domain, defining that subset "quasi factor" by specifying an upper bound on the relative movement allowed among the members of the subset in order to make it a legal sequential history. We denote the quasi linearization factor by Q and specify $D = \{d_1, d_2, \ldots\}$[1] as the set containing the subsets of the domain that Q assigns a specific bound to. Each upper bound can be a function by itself that depends on different system parameters such as number of threads, or hardware parameters, we denote the possible bounds set as F^N which is the set of all functions into $\mathbb{N}$[2]. Formally, a history H is Q-Quasi Linearizable if it has an extension H' and there is a sequential history S' and a legal sequential history S such that:*

1. *$Q : D \to F^N$ (Quasi linearization factor).* [3]
2. *$Complete(H')$ is a prefix of* some *history which is equivalent to S'.*
3. *If method invocation m_0 precedes method invocation m_1 in H, then the same is true in S'.*

[1] $D \subset Powerset(Domain(H))$
[2] F^N=$\{f | f$ is a function and $Range(f) = \mathbb{N}\}$
[3] $d_1, d_2, \ldots$ are not necessarily disjoint sets.

4. S is a permutation of S' and $\forall i$: $Distance(S'|d_i, S|d_i) \leq Q(d_i)$

We notice that a linearizable history H has Q-quasi-linearizable factor 0, i.e., $Q(Domain(H)) = 0$.

Definition 5. Q-quasi linearizable object*; An object implementation A is Quasi Linearizable with Q if for every history H of any run of A (not necessarily sequential), H is Q-Quasi-Linearizable history.*

As an example we offer a simple quasi linearizable queue that has some fixed bound on the order of it's enqueue operations. Formally, the queue implementation satisfies $(Q(\{< enq(x), void > | x \in \mathbb{X}\}) = k, Q(\{< deq(), x > | x \in \mathbb{X}\} \cup \{< deq(), null >\} = 0)$-Quasi Linearizable queue implementation. The idea is to spread the contention of the dequeue method by allowing to dequeue an element which is not at the head of the queue, but not more than k places away from the head. We change the dequeue operation to pick a random index between 0 and k (the quasi factor), if the picked index is larger than 0 it iterates over the list from the head to the item at the specified index, it attempts to dequeue it by doing a single CAS which attempts to mark it as deleted. If failed it retries a few times and eventually falls back to the scenario as if index 0 is picked. If it succeeds, this is the dequeued item. We evaluated the performance of our new algorithms on a multicore machine. At low quasi factor our implementation performs similar to the Michael and Scott[3] queue, however at higher quasi factores (>5) it has a better speedup.

This model can be used to specify a more relaxed concurrent model for other known data structures, such as stack, heap etc', and allow a more concurrent implementation of these. In the full version of this paper[1] we show other examples of quasi linearizable implementation as well. Also we can easily show that Bitonic Counting Network is Q-Quasi, $Q(D_{inc} = \{< O.getAndInc(), n > | n \in \mathbb{N}\}) \leq N * W$ (where N is the number of working threads and W is the network width).

Acknowledgements. This paper was supported in part by grants from Sun Microsystems, Intel Corporation, as well as a grant 06/1344 from the Israeli Science Foundation and European Union grant FP7-ICT-2007-1 (project VELOX).

References

1. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency, http://www.cs.tau.ac.il/research/guy.korland/
2. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
3. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 267–275. ACM, New York (1996)
4. Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev. 35(5), 230–243 (2001)

Brief Announcement: Fast Local-Spin Abortable Mutual Exclusion with Bounded Space

Hyonho Lee

Department of Computer Science
University of Toronto, Toronto, ON, Canada, M5S 3G4
hlee@cs.toronto.edu

Introduction. Abortable mutual exclusion is a variant of classical mutual exclusion, in which a process performing a trying protocol to enter the critical section (CS) is allowed to stop waiting for the CS to become available, by performing an abort protocol that takes a bounded number of steps.

In the *distributed shared memory* and *cache-coherent* models, the cost for a process to access its own shared memory or cache is considered to be much less than the cost to access memory located remotely. Hence, in these models, counting only *remote memory accesses (RMAs)* is a good measure of the time complexity of an algorithm. To achieve a bounded number of RMAs, each process accesses only its own local memory or cache during *busy-waiting*. Such algorithms are called *local-spin*.

Scott [4] presented the first local-spin abortable mutual exclusion algorithms. In his algorithms, waiting processes form a queue and, when no processes abort, each invocation performs only a constant number of RMAs in the trying protocol, even under high contention. However, Scott's algorithms use unbounded space, and there is no bound on the number of RMAs an invocation may perform, even if only two processes can abort. Jayanti [3] presented the first local-spin abortable mutual exclusion algorithm with bounded space. In his algorithm, waiting processes form a tree and each invocation performs $\Theta(\min\{\log N, k\})$ RMAs, where N is the number of processes and k is the contention. Danek and Lee [1] presented another local-spin abortable mutual exclusion algorithm in which each invocation performs $\Theta(\log N)$ RMAs. It uses only registers, whereas the previous algorithms use more powerful primitives such as COMPARE_AND_SWAP or LL/SC.

Summary of Results. We present a new local-spin abortable mutual exclusion algorithm for the cache-coherent model. It is based on a queue, uses bounded space, performs a bounded number of RMAs per invocation in the worst case, and performs $O(k^2)$ RMAs per invocation if at most k processes abort concurrently.

We first define an object type, *S-HAD*, from which it is easy to construct a local-spin abortable mutual exclusion algorithm. S-HAD is a sequence of records, each owned by a different process. It supports three operations: APPEND, which appends a record at the end of the sequence, DELETE, which deletes a record from the sequence, and HEAD, which checks whether a record is at the head of the sequence. Only the process that owns a record can perform these operations with that record.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 130–132, 2010.

To enter the CS, a process APPENDS a new record to an S-HAD object and then keeps performing HEAD until its record is at the head of the sequence. If the process wants to exit from the CS or abort, it DELETES its record.

We give two wait-free implementations of an S-HAD object in which each process performs $O(N^2)$ RMAs between beginning a call to APPEND and completing its subsequent call to DELETE. Hence, the resulting abortable mutual exclusion algorithms have $O(N^2)$ RMA complexity. Our first implementation uses unbounded space and the second improves it to have $O(N^2)$ space complexity.

We represent an S-HAD object by an intree of records, each with a pointer to a previously appended record and a flag that indicates whether the record is in the S-HAD sequence or has been *logically deleted*. The root of the tree is a dummy record, which is never deleted. All records in the S-HAD sequence are on the same path to the root and the one that is closest to the root is at the head of the sequence. There is also a FETCH_AND_STORE (or SWAP) object, $Tail$ that initially points to the root. To perform APPEND(R), a process atomically reads $Tail$ and updates $Tail$ to point to R. Hence, $Tail$ always points to the record that was appended most recently.

During HEAD(R), if R points to a logically deleted record R', then R's pointer is set to where R' points. Note that different processes may perform such pointer updates simultaneously. Pointer updates are performed until R points to a non-deleted record, which is the root if and only if R is at the head of the S-HAD sequence. During DELETE(R), a process first sets R's flag to indicate that R is logically deleted and then repeatedly performs pointer updates as in HEAD.

We prove that, if a process owns a record R and also owns an ancestor of R, then some record R' from R upto but not including that ancestor is either still in the S-HAD sequence, or has been logically deleted but its owner is still performing DELETE(R'). This implies that the tree has $O(N^2)$ height and a process performs $O(N^2)$ RMAs between a call to APPEND and completing its subsequent call to DELETE. Moreover, if $O(k)$ processes perform DELETE concurrently, then a process performs $O(k^2)$ RMAs in this period. In particular, if there are no aborts, then the resulting mutual exclusion algorithm performs $O(1)$ RMAs per invocation.

In our first implementation, each time a process performs APPEND, it uses a new record. DELETED records are not deallocated. Thus, it uses unbounded space. Our second implementation uses only $O(N^2)$ records. It achieves this by a new, wait-free memory reclamation method that generalizes reference counting.

The *reference counter* (RC) of a record stores an upper bound on the number of pointers in shared memory that point to it. A record can be deallocated only when its RC is zero. To avoid incrementing the RC of a record that has been deallocated, a process can use DOUBLE_COMPARE_AND_SWAP to simultaneously verify that some other pointer still points there [2]. Unfortunately, DOUBLE_COMPARE_AND_SWAP is not usually available. Valois [5] presented a reference counting method using FETCH_AND_ADD, but our algorithm combined with his method has $\theta(N^4)$ RMA complexity.

In our new memory reclamation method, each record has two counters in addition to an RC: a *proactive counter* (PC), which is stored together with its pointer in a single variable, and a *distributed reference counter* (DC), which is stored together with its RC in a single variable. They are used to keep track of pointers that have been read and may be written to shared memory in the future. A PC stores the number of times its associated pointer has been read since it was last updated. The value of a pointer's PC is transferred to the DC of the record it pointed to after the pointer is updated. The sum of the PC's of all records that point to R plus the DC of R is at most the number of times a pointer to R has been read minus the number of times a pointer to R has been overwritten. When a new record is created, its RC and DC are initialized to one and its PC is initialized to zero.

Before writing a pointer to a record, a process must read a pointer to that record and then increment its RC. When a process reads a pointer, it also increments its PC at the same time, using FETCH_AND_ADD. When a pointer to R is overwritten to point to another record or to NIL, the pointer's PC is also read and set to zero at the same time, using FETCH_AND_STORE. Then it atomically adds the value it read from the pointer's PC to R's DC and decrements both R's RC and R's DC, using FETCH_AND_ADD.

A record R can be deallocated only after both its RC and DC have been set to zero and its owner has finished DELETE(R). To ensure this, when a process sets both R's RC and R's DC to zero and when R's owner finishes DELETE(R), they perform TEST_AND_SET on a variable, DONE, in record R. If a process sees that DONE was already set, it updates R's pointer to NIL and then deallocates R. Note that this pointer update may cause another record's RC and DC to become zero. This is handled recursively.

Acknowledgments. I thank my advisor Professor Faith Ellen for numerous helpful suggestions during the writing of this paper.

References

1. Danek, R., Lee, H.: Brief Announcement: Local-Spin Algorithms for Abortable Mutual Exclusion and Related Problems. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 512–513. Springer, Heidelberg (2008)
2. Detlefs, D.L., Martin, P.A., Moir, M., Steele Jr., G.L.: Lock-Free Reference Counting. In: The 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 190–199 (2001)
3. Jayanti, P.: Adaptive and Efficient Abortable Mutual Exclusion. In: Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (July 2003)
4. Scott, M.L.: Non-blocking Timeout in Scalable Queue-based Spin Locks. In: The 21st Annual Symposium on Principles of Distributed Computing (July 2002)
5. Valois, J.D.: Lock-Free Linked Lists Using Compare-and-Swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222 (1995)

What Is the Use of Collision Detection (in Wireless Networks)?

Johannes Schneider and Roger Wattenhofer

Computer Engineering and Networks Laboratory,
ETH Zurich, 8092 Zurich, Switzerland

Abstract. We show that the asymptotic gain in the time complexity when using collision detection depends heavily on the task by investigating three prominent problems for wireless networks, i.e. the maximal independent set (MIS), broadcasting and coloring problem. We present lower and upper bounds for all three problems for the Growth-Bounded Graph such as the Unit Disk Graph. We prove that the benefit of collision detection ranges from an exponential improvement down to no asymptotic gain at all. In particular, for the broadcasting problem our deterministic algorithm is running in time $O(D \log n)$. It is an exponential improvement over prior work, if the diameter D is polylogarithmic in the number of nodes n, i.e. $D \in O(\log^c n)$ for some constant c.

1 Introduction

When studying distributed algorithms for wireless networks, the algorithm designer usually chooses between two models. The popular *radio network model* buys into worst-case thinking: Concurrent transmissions cancel each other because of interference, usually to a degree such that a potential receiver cannot even sense that there has been a message collision. On the other hand, the *local model* is used to abstract away from media access issues, allowing the nodes to concurrently communicate with all neighbors.

Clearly, the local model is too optimistic. The radio network model, however, often is too pessimistic. Most wireless devices can distinguish at least four states: (i) either the wireless node is transmitting itself and is therefore not capable of noticing any other communication, or it is silently listening, usually allowing it to differentiate between the other three states: (ii) the media is free because nobody is transmitting, (iii) at least one node is transmitting and the message can be decoded, and (iv) more than one node is transmitting but no message can be decoded. In the last case the listening node can sense that there are transmissions happening, e.g. there is energy on the channel in a wireless network. This model is called the *collision detection model*.

Furthermore, many algorithms for wireless networks are designed for general graphs. This model does not capture the nature of (somewhat) circular transmission ranges of wireless devices. Therefore, within the wireless computing community the so-called *Unit Disk Graph (UDG)* and variations of it, e.g. the *Quasi Unit Disk Graph*, have been widely adopted. In the UDG two nodes

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 133–147, 2010.

are adjacent if their distance is at most 1. We use a generalized model of these geometric graphs, i.e. *Growth-Bounded Graphs (GBG)*, which restrict the size of an independent set in the neighborhood of a node. Interestingly, we show that the lower bound for general graphs without collision detection for deterministic broadcasting can be adapted to GBG without any asymptotic change. The lower bound for randomized algorithms can be adapted as well yielding no asymptotic change already for graphs of polylogarithmic diameter (in the number of nodes n). Thus, the choice of the GBG model does not seem to render the problem more simple.

We make the same assumptions about the graph, wake-up, topology etc. in both models. In particular, we assume that an estimate of n is known. Without an estimate of n a transmission takes $\Omega(\frac{n}{\log n})$ in the radio network model, yielding a clear advantage for algorithms employing collision detection. For an overview of lower and upper bounds see Table 1. All in all, an advantage of collision detection is that it allows to design fast deterministic algorithms giving reliable bounds on the time complexity. For example, our MIS algorithm is asymptotically optimal, and also considerably faster (i.e. a factor of $\log n/\log\log n$) than the best possible MIS algorithm for the radio network model. For broadcasting, our deterministic algorithm can be exponentially faster than the best deterministic counter part in the radio network model. For coloring, the current lower and upper bound show that there cannot be an asymptotic gain for randomized algorithms for graphs of maximal degree $\Delta \in \Omega(\log^2 n)$.

Table 1. Comparison of deterministic (det.) and randomized (ra.) algorithms with/ without collision detection for various problems in GBG

Upper and Lower Bounds		
Problem	With Collision Detection	Without
MIS	$O(\log n)$ det. [This paper]	$O(\log^2 n)$ ra. [14]
	$\Omega(\log n)$ [This paper]	$\Omega(\log^2 n/\log\log n)$ [9]
$\Delta+1$ Col.	$O(\Delta+\log^2 n)$ ra. [16]	$O(\Delta+\log^2 n)$ ra. [16]
	$\Omega(\Delta+\log n)$ [This paper]	$\Omega(\Delta+\log n)$ [This paper]
Broadcast	$O(D\log n)$ det. [This paper]	$O(n\log n)$ [10] det.
	$\Omega(D+\log n)$ [This paper]	$\Omega(n\log_{n/D} n)$ det. [10] [This paper]

2 Related Work

The MIS problem has been studied in many types of graphs using many different models, e.g. the UDG and its generalization the GBG [12] or geometric radio networks (GRN), e.g. [5]. In the weaker GRN model nodes are positioned in the plane and each node knows its coordinates by a GPS device or some other means (and sometimes also the coordinates of its neighbors or a bound on the distances). A node v is connected to all other nodes within some distance $dist(v)$. Often the distance is equal for all nodes, e.g. [5], and thus connectivity

is the same (up to a scaling factor) as for UDG. In the message passing model, where all nodes can exchange messages at the same time (without collisions), an asymptotically optimal MIS algorithm was stated in [15] needing $O(\log^* n)$ communication rounds for GBG. We extend this algorithm in several ways in this paper. If collisions can occur, but may not be detected, in [14] a randomized algorithm taking time $O(\log^2 n)$ was given, which is optimal up to a factor of $O(\log \log n)$ [9]. It even works for arbitrary wake-up, i.e. nodes do not share global time.

For the well-studied broadcasting problem under the assumption of unknown topology and conditional (also called non-spontaneous) wake-up, i.e. a node can perform any computation only after detecting some activity (e.g. receiving the message or detecting energy on the channel) an optimal randomized algorithm was given in [10] running in $O(D \cdot \log(n/D) + \log^2 n)$ assuming collisions (but no detection) in general (undirected) graphs. In the deterministic case in the same paper an algorithm is described requiring $O(n \cdot \log^2 D)$ steps, which is optimal up to factor of $O(\log D)$ [3]. We extend the lower bound for deterministic algorithms [10] as well as the $\Omega(D \cdot \log(n/D))$ bound for randomized algorithms [13] to GBG. The $\Omega(\log^2 n)$ lower bound [1] cannot be extended in the same manner as discussed in Section 5.

In [2] it was shown how to broadcast a message of size $O(k)$ in time $O(k \cdot D)$ by using collision detection to forward a message bit by bit in arbitrary graphs. In the same paper the currently fastest deterministic algorithm for arbitrary message size also using collision detection was given taking time $O(n \cdot D)$. Thus for the crucial class of GBG in the area of wireless networks our algorithm is an exponential improvement for graphs of polylogarithmic diameter. In [8] broadcasting is discussed with and without collision detection using "advice", i.e. each node is given some number of bits containing arbitrary information about the network. It is shown that for graphs, where constant broadcasting time is possible, $O(n)$ bits of advice suffice to achieve optimal broadcasting time without collision detection, whereas $o(n)$ bits are not enough (even with collision detection at hand). In case of GRN only a constant number of bits is sufficient.

[5] assumes a GRN, where every node can detect collisions and knows its position. This allows to assign nodes into grid cells, which is the key to achieve asymptotically optimal broadcasting time of $\Theta(D + \log n)$.

In [11] an $O(n)$ time deterministic algorithm for the problem of leader election with collision detection for arbitrary networks was given. In [19] a randomized leader election protocol is given for single-hop networks running in expected time $O(\log \log n)$. In [4] deterministic algorithms for consensus and leader election were studied for single-hop networks, i.e. the underlying graph forms a clique. With collision detection both tasks can be performed in $\Theta(\log n)$, whereas without collision detection time $\Omega(n)$ is required. As in this paper (see Algorithm *Asynchronous MIS*), round coding was used to synchronize rounds. For single-hop networks time $\Omega(k(\log n)/\log k)$ [7] is needed by any deterministic algorithm until k stations out of n transmit using collision detection.

3 Model and Definition

Communication among nodes is done in synchronized rounds. In every round a node v can either listen or transmit. A listening node v can successfully receive a message in round i, if exactly one neighbor $u \in N(v)$ was transmitting in round i. We say a node v *detected transmission (dT)* in round i, if the node was listening in round i, if it has at least one transmitter in its neighborhood $N(v)$.

We assume that n is known and all nodes have unique IDs from the interval $[1, n]$ using the same number of bits, i.e. small IDs have a prefix with 0s such that all IDs have equal length.[1] All our algorithms are shown to work in case of asynchronous wake-up, i.e. each node wakes-up at an unknown point in time. Only after its wake-up it is able to follow ongoing communication. The time complexity of an algorithm denotes the number of rounds until a solution has been computed for all nodes, i.e. it denotes the time from the wake-up of the last node until all nodes have computed a solution. For broadcasting we focus on conditional (or non-spontaneous) wake-up, where nodes wake-up and can perform computations (and transmissions) only after they detected transmission for the first time.

A set S is a maximal independent set (MIS), if any two nodes $u, v \in S$ have hop distance at least 2 and every node $v \in V \setminus S$ is adjacent to a node $u \in S$. A MIS S of maximum cardinality is called a maximum independent set. We model the communication network using undirected growth-bounded (also known as bounded-independence) graphs(GBG):

Definition 1. *A graph $G = (V, E)$ is growth-bounded if there is a polynomial bounding function $f(r)$ such that for each node $v \in V$, the size of a MaxIS in the neighborhood $N^r(v)$ is at most $f(r)$, $\forall r \geq 0$.*

In particular, this means that for a constant c the value $f(c)$ is also a constant. A subclass of GBGs are (Quasi)UDGs, which have $f(r) \in \mathrm{O}(r^2)$.

We denote by $\log^{(j)} n$ the binary logarithm taken j times recursively. Thus $\log^{(1)} n = \log n$, $\log^{(2)} n = \log \log n$, etc. To improve readability we assume that $\log^{(j)} n$ is an integer for any j. The term $\log^* n$ denotes how often one has to take the logarithm to get down to 1, i.e. $\log^{(\log^* n)} n \leq 1$.

4 MIS Algorithm

We present an algorithm containing the most essential ideas assuming simultaneous wake-up of all nodes in Section 4.1. In Section 4.2 we show how to extend the algorithm to allow for arbitrary wake-up times.

4.1 MIS, Synchronous Wake-Up

In our deterministic algorithm a node performs a sequence of competitions against neighbors. After a competition a node might immediately compete again or it

[1] A polynomial bound n^c of the number of nodes n and IDs chosen from the range $[1, n^c]$, would yield the same asymptotic run time for all our algorithms.

might drop out and wait for a while or it might join the MIS. During a competition a node transmits a value in a bit by bit manner, i.e. one bit per round only.

Since messages cannot be exchanged in parallel among interfering nodes, it looks like one communication round of a competition in the local model requires potentially $\Delta + 1$ rounds in the collision detection model. However, concurrent communication despite interference is possible, if a node v transmits its value r_v^j (with $r_v^0 := ID_v$) bit by bit (line 8 to 14), to get value r_v^{j+1} which is used to update its state and for the next competition. In case bit k of r_v^j is 1, node v transmits otherwise it listens. It starts from the highest order bit of r_v^j and proceeds bit by bit down to bit 0. As soon as it detected a transmission for the first time, say for bit l, node v sets its value r_v^{j+1} to l (line 11) and does not transmit for the remaining bits. If it has never detected a transmission while communicating r_v^j, its result is $\log^{(j)} n$. For example, consider the first competition of three nodes u, v, w, which form a triangle. Let ID_u be 1100, ID_v be 1001 and ID_w be 1101. Initially, all assume to have have highest ID, i.e. result $r_u^1 = r_v^1 = r_w^1 = 4$. In the first round all nodes transmit. In the second u and w transmit. Node v detects a transmission and sets its result to 1 and waits. In the third round no node transmits and in the fourth round w transmits and node u sets its result to 3, while w keeps its (assumed) result 4.

After each competition the states are updated in parallel (see algorithm Update State). A node starts out as *undecided* and competes against all *undecided* neighbors. For the first competition, which is based on distinct IDs, we can be sure that when node v transmitted its whole ID, i.e. has result $r_v^1 = \log n$, no other node $u \in N(v)$ has the same result $r_v^1 = r_u^1 = \log n$, since IDs differ. Thus, node v joins the MIS and informs its neighbors. All nodes in the MIS and their neighbors remain quiet and do not take part in any further competitions. For any competition $j > 1$ several nodes might be able to transmit their whole result bit by bit without detecting a transmission, e.g. $r_u^{j+1} = r_v^{j+1} = \log^{(j)} n$ for two adjacent nodes u, v. In this case, node v changes its state to marked M. A marked node is on its way into the MIS but it will not necessarily join. A neighbor of a marked node remains quiet for a while. More precisely, the algorithm can be categorized into stages (lines 3 to 17), consisting of $f(2) + 1$ phases (lines 4 to 13), being composed of a sequence of $\log^* n + 2$ competitions. A node changes its state from undecided to some other state within a phase. An M node changes back to undecided after a phase (line 16). A neighbor of a marked node, i.e. an N_M node, changes back to undecided after a stage (line 18) and competes again in the next stage.

In order to update the state of neighbors of nodes having joined the MIS or having become M, two rounds are reserved. One round is used by M nodes to signal their new presence (line 4 in Algorithm Update State) and the other by *MIS* nodes (line 7). All other nodes listen during these rounds and update their states if required (lines 10 and 11).

Theorem 1. *The total time to compute a MIS is in* $O(f(f(2) + 2) \log n) = O(\log n)$ *and messages of one bit are sufficient.*

The proof for Algorithm MIS can be found in the technical report [17].

Algorithm MIS

For each node $v \in V$
1: State $s_v := \textit{undecided}$
2: **for** l:=1 to $f(f(2)+2)$ by 1 **do**
3: **for** i:=0 to $f(2)$ by 1 **do**
4: $r_v^0 := ID_v$
5: **for** j:=1 to $\log^* n + 2$ by 1 **do**
6: $r_v^j := \log^{(j)} n$
7: **for** k:=0 to $\log^{(j)} n$ by 1 **do**
8: **if** $s_v = \textit{undecided}$ **then**
9: **if** (Bit k of $r_v^{j-1} = 1$) $\wedge (r_v^j = \log^{(j)} n)$ **then** transmit
10: **else if** (Detected transmission) $\wedge (r_v^j = \log^{(j)} n)$ **then** $r_v^j := k$
11: **end if**
12: **end if**
13: **end for**
14: Update state s_v
15: **end for**
16: **if** $s_v = M$ **then** $s_v := \textit{undecided}$ **end if**
17: **end for**
18: **if** $s_v = N_M$ **then** $s_v := \textit{undecided}$ **end if**
19: **end for**

4.2 MIS, Asynchronous Wake-Up

Unfortunately, asynchronism introduces some difficulties. For instance, if a node wakes up and transmits without having any information about the state of its neighbors then it might disturb and corrupt an ongoing computation of a MIS. Therefore, all nodes inform their neighbors concurrently about their state and current activity. We guarantee a synchronous execution of Algorithm MIS without disturbance of woken-up nodes by using a schedule repeating after six rounds (see Algorithm Asynchronous MIS).

The idea is that nodes involved in a computation (or in a MIS) transmit periodically and thereby, force woken-up neighbors to wait. More precisely, upon wake-up a node listens until no neighbor has transmitted for 7 rounds. If a node has detected transmission for two consecutive rounds it knows that there is a neighbor in the MIS. A node executes Algorithm MIS by iterating the six round schedule as soon as it has not detected transmission for 7 rounds. A node transmits in the first round, if it executes or is about to execute Algorithm MIS (during round 3 of the schedule). This ensures that for a node v either a neighbor starts executing Algorithm MIS concurrently with v or it waits until v has completed the algorithm. In the second and fourth round no transmissions occur. A node transmits in the fifth and sixth, if it is in the MIS. The schedule is iterated endlessly in order that nodes in the MIS continuously inform woken-up neighbors about their presence. This prevents them from attempting to join the MIS.

Let t_{MIS} denote the time Algorithm (synchronous) MIS takes for computing a MIS when all nodes start synchronously.

Algorithm Update State

For each node $v \in V$
1: **if** $(s_v = \textit{undecided}) \wedge (r_v^j = \log^{(j)} n)$ **then**
2: **if** $j = 1$ **then**
3: $s_v := MIS$
4: Wait 1 round and transmit
5: **else**
6: $s_v := M$
7: Transmit and wait 1 round
8: **end if**
9: **else**
10: **if** (Detected transmission) $\wedge (s_v = \textit{undecided})$ **then** $s_v := N_M$ **end if**
11: **if** Detected transmission **then** $s_v := N_{MIS}$ **end if**
12: **end if**

Asynchronous MIS

Upon wake-up:
1: **Listen until** no transmission detected for 7 consecutive rounds
2: **if** ever detected transmission for 2 consecutive rounds **then** $s_v := N_{MIS}$ **else** $s_v := \textit{executing}$; SixRoundSchedule() **end if**

SixRoundSchedule():
3: **loop forever**
4: **if** $s_v = \textit{executing}$ **then** Transmit **else** Sleep **end if**
5: Sleep
6: **if** $s_v = \textit{executing}$ **then** Execute 1 step in Algorithm MIS (Section 4.1) **else** Sleep **end if**
7: Sleep
8: **if** $s_v = MIS$ **then** Transmit twice **else** Sleep two rounds **end if**
9: **end loop**

Theorem 2. *Algorithm Asynchronous MIS computes a MIS in time* $O(t_{MIS})$.

Proof. If a set of nodes $U \subseteq V$ start Algorithm MIS synchronously and are not disturbed by any node $w \notin U$ interfering the computation then a correct MIS is computed (see Analysis Algorithm MIS). A node v computing a MIS transmits a message at least every six rounds, since a neighbor $u \in N(v)$ must not start Algorithm MIS if it detected transmission within seven rounds, it cannot start a computation if it woke-up while v is active. Consider an arbitrary pair u, v of neighboring nodes, e.g. $u \in N(v)$ that are awake but not executing Algorithm MIS. If node v has not detected a transmission for seven rounds, it starts transmitting a message periodically every six rounds and executes Algorithm MIS. Any neighbor of v, i.e. u, that does not start at the same time, detects a transmission from v and waits.

If a node v detects two consecutive transmissions a neighbor must be in the MIS and does not take part in any new computation of a MIS. In case, it detects

transmissions (but non-consecutive) ones, some neighbor $u \in N(v)$ is executing Algorithm MIS. Thus within time $O(t_{MIS})$ a node $w \in (N(u) \cup u) \subseteq N^2(v)$ within distance 2 from v joins the MIS. Since the size of a maximum independent set within distance 2 is bounded by $f(2)$ (see Model Section) within time $O(f(2)t_{MIS}) = O(t_{MIS})$ node v is in the MIS or it has a neighbor in the MIS.

4.3 Broadcast Algorithm

Our deterministic algorithm iterates the same procedure, i.e. the same schedule, using a fixed number of rounds. First, the current set of candidates (rounds 1 and 2) for forwarding the message is determined. A *candidate* is a node having the message and also having a neighbor lacking it. Second, some candidates are selected using a leader election algorithm, i.e. by computing a MIS. Finally, the chosen nodes transmit the message to all their neighbors without collision. If all nodes in the MIS transmitted the message concurrently, then no node might receive the message. This is because any node can be adjacent to more than one node in the MIS and suffer from a collision if all of them transmit concurrently. For that reason we must select subsets of the nodes in the MIS and let the nodes in each subset transmit in an assigned round. Explicitly constructing such sets is difficult in a distributed manner because a node in the MIS is unaware of the identities of the other nodes in the MIS. However, we can use the combinatorial tool of so called (n, k)-strongly selective families[6] of sets $\mathcal{F} = \{F_0, F_1, ..., F_{m-1}\}$ with $F_i \subseteq V$, which yield a direct transmission schedule of length $|\mathcal{F}| = m$ for each node in the MIS, such that every node out of the given set of k nodes can transmit to all its neighbors without collision. A node v transmits in round i if $v \in F_i$.

An essential point for making fast progress is that we distinguish between nodes that have (just) received the message and never participated in a leader election and nodes that have the message and already did so. The former ones, i.e. new candidates, are preferred for forwarding the message, since, generally, they have more neighbors lacking the message.

More precisely, in our deterministic Algorithm DetBroadcast (see Table 2) a node lacking the message (state *LackMsg*) that receives the message immediately joins the computation of leaders, i.e. of a MIS, in the next execution of the schedule by switching to state *CompMIS*. After it has participated in the leader election once, it switches to state *HaveMsg* if it has not become a leader. If it has become a leader, i.e. is in the MIS, it transmits the message and exits. A node can only reattempt to become a leader, i.e. switch back to state *CompMIS*, in case no neighbor of it has just received the message, i.e. changed from state *LackMsg* to *CompMIS*.

Next, we show that all neighbors of a candidate get the message within logarithmic time.

Theorem 3. *Any candidate v ends the algorithm in time $O(\log n)$.*

Proof. For any node v at most $f(2)$ nodes $u \in N^2(v)$ are in state $CompMIS$ in round $t_M + 3$, i.e. after the execution of the MIS algorithm. This follows from the

Table 2. Algorithm DetBroadcast, where dT stands for (*has*) *detected transmission* and returns true, if a node has listened and detected a transmission.

Schedule	State $s_v = CompMIS$	$s_v = HaveMsg$	$s_v = LackMsg$
Round 1	Transmit		Listen
2	Listen if not dT then exit		if dT then Transmit
3	Transmit	if not dT then $s_v := CompMIS$	Sleep
FOR i=1..t_{MIS}			
$3+i$	Compute step i of Algorithm MIS	Sleep	
ENDFOR			
still round $t_{MIS}+3$	If not joined MIS then $s_v := HaveMsg$		
FOR i=1..$\lvert\mathcal{F}\rvert$			
$3+t_M+i$	if $v \in F_i$ then Transmit msg	Sleep	if received msg then $s_v := CompMIS$
ENDFOR			
still $3+t_M+\lvert\mathcal{F}\rvert$	$s_v = HaveMsg$		

correctness of the MIS algorithm and the definition of GBG. For the existence of a strongly related $(n, f(2))$ family of size $\mathrm{O}(f(2)^2 \log n)$, we refer to [6]. The time to compute a MIS is $\mathrm{O}(f(f(2)+2)\log n)$ as shown in Theorem 1. Thus one execution of the schedule takes time $O((f(f(2)+2)+f(2)^2)\log n)$.

Either a candidate v is computing a MIS itself or at least one neighbor $u \in N(v)$ does so. Assume neighbor u participates in computing a MIS S_0, joins the MIS and transmits. At least a subset of the neighbors $U \subseteq N(u)$ receives the message for the first time and any candidate $w \in U$ participates in the next computation of a MIS S_1. Assume at least one candidate $w \in U$ exists, i.e. $|U| > 0$, and a neighbor $x \in N(w)$ joins the MIS. Note that $x \notin N(u)$, since u transmitted the message to all its neighbors $w \in N(u)$. Therefore, all nodes $w \in N(u)$ have changed to state *HaveMsg* before the computation of S_1. Therefore, some node $x \in N(w) \setminus N(u)$ receives the message for the first time and participates in the next computation of a MIS S_2. Assume a node $y \in N(x) \cap x$ joins the MIS. This node y is not adjacent to u, i.e. $y \notin N(u)$, because no nodes $N(u) \cap N(x)$ participate together with y since all neighbors $N(x)$ changed to state *HaveMsg* before the computation of S_2. Thus node y is independent of all nodes in the MIS S_0 that transmitted and also any prior nodes that transmitted. Therefore node v gets a transmitting (independent) node with three computations of a MIS within distance 4. The maximum size of any independent set is bounded by $f(4)$ within distance 4. Therefore within time $\mathrm{O}((f(f(2)+2)+f(2)^2)f(4)\log n) = O(\log n)$ all neighbors of node v must have received the message and therefore node v cannot be a candidate any more.

Theorem 4. *Algorithm DetBroadcast finishes in time $O(D \log n)$ for a GBG.*

Proof. Due to Theorem 3 any neighbor of a node having the message also receives it within time $O(\log n)$. Therefore, within time $O(D \log n)$ any node receives the message.

5 Lower Bounds For MIS, Coloring and Broadcasting With Collision Detection

To begin with, we present two lower bounds. One showing that indeed $\Omega(\Delta)$ colors and time is needed to color a GBG and one that shows that for any Δ, i.e. also $\Delta \in O(1)$, time $\Omega(\log n)$ is needed even to make a successful transmission with high probability, i.e. $1-1/n$. The second lower bound implies a bound on the MIS and the same techniques imply an $\Omega(\log n)$ lower bound for broadcasting.

Theorem 5. *Any (possibly randomized) algorithm requires time $\Omega(\Delta)$ (in expectation) to compute a $\Delta+1$ coloring with high probability in a GBG (with or without collision detection).*

In the proof we use an argument based on information theory. Essentially, any node must figure out the identities of the nodes in its neighborhood. We show that the amount of possibly shared information about the neighborhood with $\Omega(\Delta)$ communication rounds is not sufficient to narrow down the options of distinct neighborhoods sufficiently.

Proof. Let the disconnected graph G consist of a clique C of Δ nodes and some other arbitrary subgraph such that no node $v \in C$ is adjacent to a node $u \notin C$. To color the clique C, any algorithm requires $\Delta+1$ colors. We restrict the possible choices of $\binom{n}{\Delta+1}$ cliques as follows. We pick $(\Delta+1)/2$ sets $S_0, S_1 ..., S_{(\Delta+1)/2-1}$, each consisting of 4 nodes, i.e. $|S_i| = 4$.(We assume that $4(\Delta+1) \leq n$.) The algorithm gets told all the sets S_i and that out of every set S_i consisting of four nodes, two nodes are in the clique C. However, it is unknown to the algorithm which two nodes out of the four are actually chosen. The algorithm must reserve two of the $\Delta+1$ colors for each set S_i, i.e. the (unknown) nodes in the set. Assume (without loss of generality) that the algorithm assigns colors $2i$ and $2i+1$ to the chosen nodes of set S_i.

Assume an algorithm could compute a correct coloring in time Δ/c_0 for some constant $c_0 \geq 6$. Within Δ/c_0 rounds at most Δ/c_0 out of the $\Delta+1$ nodes in the clique can transmit without collision. Assume that even if there is a collision due to some transmitters, say u, v, w, in a round i, all nodes in the clique receive one message of the same node, say all nodes receive v's message.(Note, that more information about the neighborhood can only benefit a node.) Additionally, any node can detect whether there was 0,1 or more than 1 transmitter in its neighborhood. For an upper bound assume a node $v \in C$ gets to know Δ/c_0 of its neighbors, i.e. receives one message of each of these Δ/c_0 nodes, and additionally, it receives two bits of information in each round, i.e. one of the values $\{0, 1, >1\}$ can be encoded by two bits of information, e.g. bits 11 correspond to >1 transmitters, bits 10 correspond to 1 transmitter and bits 00 correspond to none. Thus, in total a node v

gets to know at most $2\Delta/c_0$ bits. Observe that every node gets the same information, i.e. bits. The transmitted information is used to figure out, which two nodes of each set S_i are actually in the clique C in order to get a correct coloring. Since the algorithm is supposed to know already Δ/c_0 nodes, at least for $(1-1/c_0)\Delta/2$ leftover sets S_i no node of the set transmitted its identity. Therefore the algorithm can use the $2\Delta/c_0$ bits to figure out the identities of the $(1-1/c_0)\Delta$ nodes of the $(1-1/c_0)\Delta/2$ leftover sets S_i. Thus, any algorithm must decide on how many bits it spends on determining the two nodes out of the four possible in each set that are actually in its neighborhood. On average, it can use $\frac{(2\Delta/c_0)}{(1-1/c_0)\Delta/2} = 4/(c_0-1)$ bits per set. For $c_0 = 9$ for at least half all sets S_i the algorithm can use at most 1 bit. Assume set S_0 consists of nodes $\{a, b, c, d\}$. Any node in the set S_0 must make a decision, which of the two colors $\{0, 1\}$, it chooses based on its ID and a single bit. Assume node a decides in favor of color 1 given it received bit 0, i.e. $col(a|0) = 1$. Then all other nodes in the set must decide to pick color 0 if they receive bit 0, i.e. $col(b|0) = 0$, $col(c|0) = 0$ and $col(d|0) = 0$. If not, consider a node $x \in S_0$ that also decides in favor of color 1. In this case, if a and x are chosen to be in the clique C, both are adjacent and choose the same color. Thus the coloring is incorrect. Assume all nodes receive bit 1 and assume $col(a|1) = 0$ then $col(b|1) = 1$, $col(c|1) = 1$ and $col(d|1) = 1$. Thus, if out of the set S_0, nodes b, c are chosen then both decide on the same color, whatever the given bit is, i.e. they both pick color 0 if the given bit is 0 and color 1 if the bit is 1. If $col(a|1) = 1$ then $col(b|1) = 0$, $col(c|1) = 0$ and $col(d|1) = 0$ and nodes b, c decide on color 0 whatever the given bit is. Thus the coloring cannot be correct. Randomization can not increase the amount of exchanged information. Thus, in the end any algorithm must also decide on whether to choose color $\{0, 1\}$ based on a single bit. Through a case enumeration one can see that it is not possible to correctly guess the right colors with probability more than $1/4$. Assume $col(a|0) = 1$ with probability $p_{\geq \frac{1}{2}} \geq 1/2$. Then all other nodes in the set must decide to pick color 0 with probability $p_{\geq \frac{1}{2}}$ to have a chance higher than $1/4$ of a correct coloring. Using the same reasoning as for the deterministic case a maximum probability of $1/4$ for a correct coloring of a single set using only one bit follows. Since for at least half of all $(1-1/c_0)\Delta/2$ sets S_i with unknown nodes we can use at most one bit, we expect at least $3/4(1-1/c_0)\Delta/4$ to be colored incorrectly.

Theorem 6. *There exists a graph such that for any $\Delta > 1$, any (possibly randomized) algorithm using collision detection requires time $\Omega(\log n)$ to compute a MIS (in expectation).*

Proof. Consider a (disconnected) graph where every node has degree 1, i.e. a single neighbor. Assume every node $v \in V$ knows that its degree in the network is one but it is unaware of the identity of its neighbor u. Consider a sequence of $\frac{\log n}{8}$ rounds. For every node $v \in V$ we can calculate the probability that node v transmits in round $0 \leq i < \frac{\log n}{8}$ given that it has not yet received a message but transmitted itself or listened without detecting any transmission by its neighbor for rounds $0 \leq j < i$. There must exist a set U of $n^{7/8}$ nodes such that for every round i with $i \in [0, \frac{\log n}{8} - 1]$ all nodes in U transmit either with probability

$p_{\geq\frac{1}{2}}$ at least $\frac{1}{2}$ or with probability $p_{<\frac{1}{2}}$ less than $\frac{1}{2}$, since a node has only two choices in each round (transmit or not). Thus, out of n nodes on average at least $n/2^{\frac{\log n}{8}} \geq n^{7/8}$ must decide to transmit (and listen) in the same rounds for all $\frac{\log n}{8}$ rounds. Consider an arbitrary pair $u, v \in U$ and assume they are adjacent. The chance of a transmission from u to v or the other way around is at most $(1-p_{<\frac{1}{2}})\cdot p_{<\frac{1}{2}}+p_{\geq\frac{1}{2}}\cdot(1-p_{\geq\frac{1}{2}}) \leq \frac{1}{2}$ for one round, since any term $p\cdot(1-p)$ can be at most $1/4$. After all $\frac{\log n}{8}$ rounds the probability is at most $1-\frac{1}{4^{\frac{\log n}{8}}} \leq 1-\frac{1}{n^{1/4}}$. Assume, we randomly create $n^{7/8}/2$ pairs of nodes from the set U, such that the two nodes from each pair are adjacent. We expect for $n^{7/8}/2/n^{1/4} = n^{5/8}/2$ pairs that no message is exchanged. Thus the nodes from these pairs must make the decision, whether to join or not to join the MIS based on the same information. Let $U_1 \subseteq U$ be all nodes that decide to join the MIS with probability at least $1/2$ if they have never received a message, i.e. transmitted in the same rounds as their neighbor(s), and let all other nodes be in set $U_2 = U \setminus U_1$. Either U_1 or U_2 is of size at least $|U|/2$. Assume it holds for U_1. If we pick pair after pair then the probability that both nodes are taken from U_1 is at least $1/16$ for a pair independent of all previously picked pairs as long as at most $|U_1|/4 \geq |U|/8$ pairs have been chosen, i.e. for the remaining nodes in U_1 holds $|U_1| \geq |U|/4$. Thus, we expect $|U|/8/16 = |U|/128$ pairs to have both nodes in the same set U_1 or U_2. The chance that both join for a pair in U_1 is $p_{\geq\frac{1}{2}}p_{\geq\frac{1}{2}} \geq 1/4$ or none does for a pair in U_2 is $(1-p_{<\frac{1}{2}})(1-p_{<\frac{1}{2}}) \geq 1/4$. Thus, $1/4$ of all the pairs having transmitted in the same round and being from the same set U_1 or U_2 either both join the MIS or not in expectation. The probability that at least half of the expected $n^{5/8}/2/128/4 = n^{5/8}/1024$ pairs transmit in the same rounds, i.e. do not exchange a message, and both nodes from the pair join the MIS or both do not join is larger than $1-1/n^c$ for some arbitrary large constant c using a Chernoff bound. Thus the probability that the algorithm finishes in time less than $\log n/8$ is at most $1/n^c$.

The argument for the deterministic case is analogous, i.e. an equally large set (as in the randomized case) of nodes must transmit in the same rounds and it is not possible that all pairs correctly decide to join the MIS or not for all possible choices of neighborhoods. More precisely, consider three nodes u, v, w that all transmit in the same round (given their only neighbor is also one of u, v, w). If in a graph G_1 u, v are adjacent then either u or v must join the MIS without having received a message from its neighbor, i.e. v is unaware whether its neighbor is u, v or w. Assume u joins the MIS then v cannot join. If in a graph G_2 v, w are adjacent then both transmit the same sequence and since v does not join w has to. Now if in a third graph u, w are adjacent then both join the MIS violating the independence condition of a MIS.

Observe that the above theorem even holds for synchronous wake-up. Since one can compute a MIS from a coloring in constant time, the lower bound also holds for the MIS. More precisely, for the graph in the above proof with $\Delta = 1$ one can compute a MIS from a $\Delta+1$ coloring by putting all nodes with color 0 in the MIS.

Corollary 7. *Any (possibly randomized) algorithm using collision detection requires time $\Omega(\log n)$ to compute a coloring in GBG (in expectation).*

For broadcasting with conditional wake-up a lower bound of D is trivial. A lower bound of $\Omega(\log n)$ for networks of diameter two can be proven using the same idea as for the proof of Theorem 6.

Theorem 8. *There exists a graph such that for any $\Delta > 1$, any (possibly randomized) algorithm using collision detection requires time $\Omega(\log n)$ to make a transmission among all nodes with probability $1 - \frac{1}{n}$.*

Proof. Assume the following network of diameter two. The source is adjacent to two nodes and these two nodes in turn are adjacent to all other nodes. Consider a sequence of $\frac{\log n}{2}$ rounds. For every node $v \in V$ and every round i we can calculate the probability that node v transmits in round $0 \leq i < \frac{\log n}{2}$ given that it has received the message (and possibly other information). There must exist two nodes u, v such that for all $\frac{\log n}{2}$ rounds either they both send with probability at least $\frac{1}{2}$ or less than $\frac{1}{2}$ given they received the same information, since any node has only these two options and we have that $n > 2^{\frac{\log n}{2}}$. Thus the chance of a successful transmission of either u or v to its neighbor is at most $\frac{1}{2}$ for one round and at most $1 - \frac{1}{2^{\frac{\log n}{2}}} = 1 - \frac{1}{\sqrt{n}}$ after $\frac{\log n}{2} \in \Omega(\log n)$ rounds.

6 Lower Bound for Broadcasting without Collision Detection in GBG

The lower bounds for randomized [13] as well as for deterministic [10] algorithms for general graphs can be adapted to GBG. Both rely on constructing a graph with layers $L_0, L_1, ..., L_{\Omega(D)}$, where nodes L_i in layer i are independent and they are at distance i from the source, i.e. broadcast initiator. In [10] the graph consists of two alternating layers consisting of a single node in layer i that is connected to all nodes L_{i+1} in layer $i + 1$. Only a subset $W_{i+1} \subseteq L_{i+1}$ of the nodes in layer $i+1$ is connected to the single node in layer $i+2$. In the lower bound graph in [13] the nodes in layer i are connected to some nodes $W_{i+1} \subseteq L_{i+1}$ in layer L_{i-1} and L_{i+1}. There are no fixed layers of single nodes. The difficulty for the algorithm is figuring out the number of nodes in L_i. If it knows the number $|L_i|$ it can transmit with probability $1/|L_i|$, yielding an O(1) algorithm to get to the next layer. However, in [13] one could also use the same topology as in [10], i.e. every second layer consists only of a single node. Though this allowed the algorithm to pass every second layer in one round by transmitting with probability 1 for the other half of the layers the algorithm is unaware of the number of nodes in a layer.

For GBGs it is not possible for a node v in layer i to have an arbitrary number of independent nodes in layer $i+1$. Thus, we assume that all nodes in layer i form a clique. Therefore, a node in layer i knows all the successful transmissions that occurred in layer i. However, by elongating any protocol P by a factor of 4, the

nodes in layer i in a general graph also know all successful transmissions in layer i. The idea is to let all layers with single nodes repeat their received message to the other layers. Since a single node is adjacent to two layers, it might face a collision if a node from each of its adjacent layers transmits concurrently and thus is not able to tell any layer if the transmission was successful. Thus, we repeat 8 rounds in a round robin fashion. Any node in layer i executes one step of the algorithm in round t if $2i \mod 8 = t \mod 8$.[2] Any node in layer i transmits in round t if $2i - 1 \mod 8 = t \mod 8$ and if it received a message in the previous round. Thus a node in layer i having transmitted the message in round t knows that some node in layer i transmitted without collision, if and only if it detects transmission in round $t + 1$.

Thus, we have arrived at the following proposition:

Proposition 1. *Any deterministic broadcasting algorithm takes time* $\Omega(n \log_{n/D} n)$ *and any randomized broadcasting algorithm takes time* $\Omega(D \log \frac{n}{D})$ *for GBG.*

The lower bound of $\Omega(\log^2 n)$ [1] cannot be extended in the same manner. The lower bound graph consists of two layers L_1 and L_2, where nodes within a layer are independent. In particular, layer L_2 consists of $\Omega(\log n)$ nodes. Each node l in L_2 is connected to some set H_l of nodes in L_1. There are $\Omega(\log n)$ sets H and in some round exactly one node from each set H_l must transmit in order that all nodes in L_2 receive the message. In GBG this is not the case, since in L_2 some nodes are adjacent, i.e. by definition of a GBG at most $f(2)$ nodes in L_2 can be independent. Thus, a node in L_2 having received the message might forward it to other nodes in L_2 and it might be sufficient that for only $f(2)$ out of all $\Omega(\log n)$ sets H from L_1 a node transmit to its neighbor in layer L_2.

References

1. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. J. Comput. Syst. Sci. 43(2) (1991)
2. Chlebus, B., Gcasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in unknown radio networks. In: Symp. on Discrete Algorithms, SODA (2000)
3. Clementi, A.E.F., Monti, A., Silvestri, R.: Distributed broadcast in radio networks of unknown topology. Theor. Comput. Sci. 302(1-3) (2003)
4. Czyzowicz, J., Gasieniec, L., Kowalski, D.R., Pelc, A.: Consensus and mutual exclusion in a multiple access channel. In: Int. Symposium on Distributed Computing, DISC (2009)
5. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. Journal of Discrete Algorithms 5 (2007)
6. Erdoes, P., Frankl, P., Fiiredi, Z.: Families of finite sets in which no set is covered by the union of r others. Israel J. of Math. 51 (1985)

[2] Note, that all nodes having the message are synchronized, i.e have global clocks, if the message includes the current time t.

7. Greenberg, A.G., Winograd, S.: A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. J. ACM 32(3) (1985)
8. Ilcinkas, D., Kowalski, D.R., Pelc, A.: Fast radio broadcasting with advice. Theor. Comput. Sci. 411(14-15) (2010)
9. Jurdziński, T., Stachowiak, G.: Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 535–549. Springer, Heidelberg (2002)
10. Kowalski, D.R., Pelc, A.: Broadcasting algorithms in radio networks with unknown topology. In: Distributed Computing, vol. 18 (2005)
11. Kowalski, D.R., Pelc, A.: Leader election in ad hoc radio networks: A keen ear helps. In: ICALP (2) (2009)
12. Kuhn, F., Moscibroda, T., Nieberg, T., Wattenhofer, R.: Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 273–287. Springer, Heidelberg (2005)
13. Kushilevitz, E., Mansour, Y.: An omega(d log(n/d)) lower bound for broadcast in radio networks. In: Symp. on Principles of Distributed Computing (PODC) (1993)
14. Moscibroda, T., Wattenhofer, R.: Maximal Independent Sets in Radio Networks. In: Symp. on Principles of Distributed Computing, PODC (2005)
15. Schneider, J., Wattenhofer, R.: A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In: Symp. on Principles of Distributed Computing PODC (2008)
16. Schneider, J., Wattenhofer, R.: Coloring Unstructured Wireless Multi-Hop Networks. In: Symp. on Principles of Distributed Computing, PODC (2009)
17. Schneider, J., Wattenhofer, R.: What Is the Use of Collision Detection (in Wireless Networks). TIK Technical Report 322 (2010), ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-322.pdf
18. Tobagi, F.A., Kleinrock, L.: Packet Switching in Radio Channels: Part II - The Hidden Terminal Problem in Carrier Sense Multiple Access and the Busy Tone Solution. COM 23(12) (1975)
19. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM Journal on Computing 15 (1986)

Deploying Wireless Networks with Beeps

Alejandro Cornejo[1] and Fabian Kuhn[2]

[1] Massachusetts Institute of Technology
[2] University of Lugano

Abstract. We present the *discrete beeping* communication model, which assumes nodes have minimal knowledge about their environment and severely limited communication capabilities. Specifically, nodes have no information regarding the local or global structure of the network, do not have access to synchronized clocks and are woken up by an adversary. Moreover, instead on communicating through messages they rely solely on carrier sensing to exchange information. This model is interesting from a practical point of view, because it is possible to implement it (or emulate it) even in extremely restricted radio network environments. From a theory point of view, it shows that complex problems (such as vertex coloring) can be solved efficiently even without strong assumptions on properties of the communication model.

We study the problem of *interval coloring*, a variant of vertex coloring specially suited for the studied beeping model. Given a set of resources, the goal of interval coloring is to assign every node a large contiguous fraction of the resources, such that neighboring nodes have disjoint resources. A k-interval coloring is one where every node gets at least a $1/k$ fraction of the resources.

To highlight the importance of the discreteness of the model, we contrast it against a continuous variant described in [17]. We present an $\mathcal{O}(1)$ time algorithm that with probability 1 produces a $\mathcal{O}(\Delta)$-interval coloring. This improves an $\mathcal{O}(\log n)$ time algorithm with the same guarantees presented in [17], and accentuates the unrealistic assumptions of the continuous model. Under the more realistic discrete model, we present a Las Vegas algorithm that solves $\mathcal{O}(\Delta)$-interval coloring in $\mathcal{O}(\log n)$ time with high probability and describe how to adapt the algorithm for dynamic networks where nodes may join or leave. For constant degree graphs we prove a lower bound of $\Omega(\log n)$ on the time required to solve interval coloring for this model against randomized algorithms. This lower bound implies that our algorithm is asymptotically optimal for constant degree graphs.

1 Introduction

Communication models face the unavoidable tension between their practicality and their potential for designing interesting yet provably correct algorithms. With enough assumptions concerning the knowledge of the deployment environment and the communication capabilities of the devices used, it is not difficult to design efficient and elegant distributed algorithms. However, it is often difficult (if not impossible) to translate these algorithms to the real world. On the other hand, communication models which are cluttered with physical details encumber designing algorithms, and makes it significantly more complicated to prove correctness or efficiency.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 148–162, 2010.

This motivates the study of models such as the *discrete beeping* model considered in the present paper. This model makes little demands on the communication devices, nodes need only be able to do *carrier sensing* and differentiate between silence and the presence of a jamming signal. Carrier-sensing can typically be done much more reliably and requires significantly less energy and other resources than transmitting and receiving actual messages , see e.g. [7]. Besides requiring reliable carrier sensing, we make almost no assumptions. In particular, we do not assume knowledge of the local or global structure of the network or synchronized clocks. Further, we assume that an adversary controls when processors are woken up.

We show that even such a "weak" model allows for interesting algorithms for non-trivial tasks. In particular we focus on the problem of *interval coloring*, a variant of classic vertex coloring. Given a set of resources, the goal of interval coloring is to assign each node a large contiguous fraction of the resources such that neighboring nodes have disjoint resources. A k-interval coloring is one where every node gets at least a $1/k$ fraction of the resources. Similar to vertex coloring, interval coloring is a useful building block to establish a reliable Medium Access Layer (MAC), as it can be used to e.g. compute time or frequency division multiple access (TDMA or FDMA) schedules that avoid conflict between potentially interfering nodes. In some sense, interval coloring is even better suited for these tasks than standard graph coloring. While in a standard coloring, every node gets assigned a single color (a single slot or frequency), in an interval coloring, we can assign larger intervals to certain nodes (e.g. to nodes with a small degrees). An interval then corresponds to multiple consecutive colors in a standard coloring context.

Moreover, by relying exclusively on carrier sensing, the beeping model becomes specially well-suited for coordination tasks in wireless networks for various reasons, for example: ⋄ Most prior work [1, 3, 4, 9, 11, 14, 18, 24, 25] on coloring assumes some existing infrastructure to reliably exchange messages. If used as a building block to e.g. compute a TDMA schedule, these algorithms suffer from a chicken-and-egg problem; such colorings cannot be computed without a reliable MAC layer, however to achieve a reliable MAC layer one first needs to compute a coloring. A coloring algorithm for the beeping model would not suffer from this problem, since the model makes almost no assumptions on the communication infrastructure. ⋄ The presence of a signal can be reliably detected by carrier sensing at lower receiving power than would be required to correctly decode a message. Hence, carrier sensing can be used to communicate more energy efficiently and over larger distances than when transmitting regular messages. For example, by default the NS2 [26] simulator uses a carrier sensing range that is more than twice as large as the transmission range. Therefore, the beeping model (carrier sensing) can directly be used to compute a 2-hop interval coloring of the communication graph (for regular transmission), a necessity when using the coloring for a MAC layer that avoids hidden terminal collisions. ⋄ Although IEEE 802.11 and Bluetooth share the same frequency spectrum, they use incompatible modulation and encoding schemes. However since carrier sensing only detects the presence of a signal, it is potentially possible for a IEEE 802.11 radio to detect the presence of a Bluetooth jamming signal and vice versa. Therefore, algorithms for the beeping model could be used to allow these

two seemingly incompatible devices to agree on a non-conflict transmission schedule thereby allowing them to coexist in a non-destructive fashion.

Contributions. We assume that there is a common globally known period length T. This is a parameter of the algorithms which captures the number of resources to be shared (e.g. the number of available frequencies in FDMA). The paper has three main contributions.

First, we significantly improve a result from [17] for a continuous variant of the beeping model. The authors of [17] describe an algorithm that solves $\mathcal{O}(\Delta)$-interval coloring in $\mathcal{O}(\log n)$ periods is described in [17]. Specifically they assign every node v a $\Omega(1/d^{\max}(v))$ fraction of the resources, where $d^{\max}(v)$ is the largest degree in the 1-neighborhood of v. We describe a simpler algorithm that improves the results of [17] by computing an interval coloring with the same properties in a constant number of periods. Our result highlights the unrealistic assumptions behind the continuous model.

Second, we give a discrete variant of the beeping model and describe a Las Vegas randomized interval coloring algorithm for the discrete model. The algorithm computes a $\mathcal{O}(\Delta)$-interval coloring in $\mathcal{O}(\log n)$ periods with probability $1 - \frac{1}{n}$. Furthermore, we describe how to adapt the algorithm to work in a dynamic graph setting where nodes can join and leave arbitrarily. A new node obtains an interval at most $\mathcal{O}(\log n)$ periods after joining the network, and a node only recomputes its interval if the size of its neighborhood becomes drastically smaller. The correctness proof of both the static and dynamic versions of the algorithm rely on a balls and bins analysis.

Finally, for a local broadcast model with constant size messages, we prove a lower bound of $\Omega(\log n)$ time against randomized algorithms that solve $\mathcal{O}(\Delta)$-vertex coloring (or $\mathcal{O}(\Delta)$-interval coloring). For the discrete beeping model this implies a lower bound of $\Omega(\log n)$ periods for constant-degree graphs and $\Omega(\log n/\Delta)$ for general graphs. Moreover, if we restrict the number of beeps per period to $\mathcal{O}(1)$ it yields a lower bound of $\Omega(\log n/\log \Delta)$ for general graphs.

Related Work. Using carrier sensing for distributed computation is not novel. Scheideler et al. [21] considered a model where in addition to sending and receiving messages, nodes can perform physical carrier sensing, and described how to approximate the minimum dominating set problem under this model. Flury and Wattenhofer [7] demonstrate how to use carrier sensing as an elegant and efficient way for coordination in practice.

Our beeping model is a discretized variant of the desynchronization model first introduced by [6]. Degesys et al. [6] considered only complete graphs, and proved the eventual convergence of a biologically inspired algorithm DESYNC to a *'desynchronized state'* and conjectured a running time of $\mathcal{O}(n^2)$. Degesys and Nagpal [5] experimentally studied the performance of DESYNC in multi-hop topologies. They proved that a desynchronized state exists for 2-colorable graphs and Hamiltonian graphs, and posed the open problem of proving that a desynchronized state exists for all graphs. Later Motskin et al. [17] studied interval coloring under the same desynchronization model. In addition to assuming the continuous variant of the model, [17] assumes that nodes have knowledge of their own degree and that they are able to exchange this information to compute the maximum neighbor degree over their 1-hop neighbors. It is

not clear how nodes should obtain the maximum degree among their neighbors without reliably transmitting messages. Further, as we show in Section 4, their assumptions are too strong and allow for constant time solutions. This motivates studying the strictly weaker discrete beeping model.

Coloring the nodes of a graph is one of the most fundamental combinatorial optimization problems in computer science and has therefore been widely studied, also in a distributed context. The work on distributed coloring algorithms started with the seminal work of Linial [14] and includes a large number of papers (see e.g. [1, 3, 4, 9, 11, 13, 18, 24, 25]). The best bounds are known for randomized algorithm and they are $\mathcal{O}(\sqrt{\log n} + \log \Delta)$ for $(\Delta + 1)$-colorings (i.e., the number of colors needed by the sequential greedy algorithm) and $\mathcal{O}(\sqrt{\log n})$ for $\mathcal{O}(\Delta)$-colorings [11, 25]. Interesting in the context of TDMA schemes for wireless networks might be [12] where it is shown how to compute a coloring where each node with degree d obtains an $\Omega(1/d)$-fraction of the colors in a single communication round (i.e., nodes just need to learn the identifiers of all neighbors). Coloring in unstructured radio networks (with collisions) was considered by [16], where a randomized algorithm to compute $\mathcal{O}(\Delta)$-colorings in $\mathcal{O}(\Delta \log n)$ rounds is described (later improved in [23] to $\mathcal{O}(\Delta + \log \Delta \log n)$ rounds). In addition to the theoretical work on distributed coloring, there are many papers that describe some variant of coloring in order to compute TDMA schedules or similar MAC schemes (see e.g. [2, 8, 10, 15, 19, 20, 27]).

2 Model and Definitions

We consider a wireless network model that is as primitive as possible. In contrast to standard communication models, nodes cannot exchange messages reliably (message passing) or unreliably (unstructured radio networks), instead nodes rely entirely on carrier sensing. At any particular time, a node can be in beeping or listening mode. When a node is listening, it can only distinguish between silence or the presence of one or more beeps. This model is weaker than collision detection since nodes cannot distinguish between a single beep and a collision of two or more beeps. Moreover, a beep conveys less information than a bit, and although one could conceive coding schemes to encode bit messages using beeps, this would require additional overhead and be susceptible to collisions, thus we focus on different techniques.

We assume that nodes wake up asynchronously and the wake-up pattern is determined by an adversary. Upon waking up, a node does not know anything about the structure of the communication network, not even an estimate of its size. Similarly, nodes do not know their neighbors in the communication network or have an estimate of the size of this set. Furthermore, nodes do not have unique identifiers and the structure of the communication network is not restricted in any way (e.g. by requiring it to be a unit disk graph, a bounded independence graph, or any other special type of graph considered in the wireless networks literature [22]). Every node has access to a local clock, where the local clock of every node advances at the same rate and has no drift, however we do not assume clocks to be synchronized.

The communication network is modeled as an undirected graph $G = (V, E)$, $|V| = n$, where the set V of nodes of G represents the set of wireless devices. There is an

edge $\{u, v\} \in E$ if and only if u can listen to a beep emitted by v and viceversa. For a node $u \in V$, let $N(u) := \{v \in V | \{u, v\} \in E\}$ be the set of neighbors of u, and let $d(u) = |N(u)|$ be its degree. We denote by $\Delta = \max_{v \in V} d(v)$ the maximum degree of the graph. A *phase* refers to a time point (in the continuous model) or a time slot (in the discrete model) measured relative to the beginning of the last period. We will use phases to capture the time at which different beeps are heard with respect to the local clock of each node. Given a set S of phases, we define $S[a, b]$ to be the subset of phases in the range $[a, b]$ in S. To correctly account for ranges that cross the period boundary, we give a formal definition. Let τ be the period length (in the continuos model the period length is T time units, while in the discrete model the period length is Q time slots), and let $x = a \mod \tau$ and $y = b \mod \tau$. If $x \leq y$, $S[a, b] = \{p \in S \mid x \leq p \leq y\}$, otherwise $S[a, b] = \{p \in S \mid p \geq x \vee y \geq p\}$.

If t_u represents the time of occurrence of some event with respect to node u we use $\mathring{t}_u$ to represent the time of occurrence of the event in a global reference frame. For example, consider neighboring nodes u and v, and suppose that node u executes some event e_u at local time t_u which is instantaneously observed by node v at local time t_v. Since we do not assume synchronized clocks, then in general $t_u \neq t_v$, however $\mathring{t_u} = \mathring{t_v}$.

We say that an event happens almost surely if it happens with probability one, an event happens with high probability if it occurs with probability at least $1 - \frac{1}{n}$. Let $\mathfrak{U}(a, b)$ denote the continuous uniform distribution in the range $[a, b]$ and $\mathfrak{U}[a..b]$ denote the discrete uniform distribution in the range $[a..b]$.

We believe the model described is simple enough to be implemented or simulated in real hardware. However it is still complex enough to allow for the design of interesting algorithms with strong theoretical guarantees. We consider two variants of the basic model, a continuous version and a discrete version.

Discrete Model. Time is divided into slots of length μ, where μ depends on the physical characteristics of the wireless devices and of the communication medium. There is a known integer $Q > 0$ that denotes the number of slots per period, and is related to the number of resources available. Hence, the period length is $T = Q\mu$. Although we do not assume synchronized clocks, we assume that slots boundaries are synchronized, i.e., all nodes start new slots at the same time. Note that at the cost of small constant factors and more technical arguments, all results obtained in this paper can also be achieved in a model with unsynchronized slot boundaries.

In each slot s, each node v can either listen or beep for the whole duration of s. If a beep is emitted by node u at slot s, it is heard by any neighboring node $v \in N(u)$ that is in listening mode in slot s. In particular the operation `listen`$[m]$ puts the node in listening mode for the next m slots and returns the set of slots where it detected a beep. The operation `beep` emits a beep for the duration of the current slot.

Continuous Model. All nodes share some period length T and a beep can be infinitely short (i.e., a unit impulse function). If a beep is emitted by node u at time t, it is heard by any neighboring node $v \in N(u)$ that is in listening mode at time t. In particular the operation `listen`(δ) puts the node in listening mode for the next δ units of time and returns the set of time points where it detected beeps. The operation `beep` emits an infinitely short beep. We discuss the shortcomings of this variant in Section 4.

3 Interval Coloring

One of the central motivations behind vertex coloring in distributed environments is to use it as a building block for MAC protocols. In this setting the number of colors used translates to the number of communication channels used, and thus fewer colors imply higher throughput. In general we are interested in efficient (polylog or better) algorithms that produce vertex colorings with $\mathcal{O}(\Delta)$ colors, where Δ is the maximum degree. However, most known distributed algorithms for coloring are based on the assumption that there is already an infrastructure to reliably transmit messages with neighboring nodes, which makes them unsuitable for MAC protocols. This motivates studying coloring in the beeping model. We focus on interval coloring, a variant of vertex coloring specially well suited for the beeping model.

Given an ordered set of resources, an interval coloring assigns each node an interval (contiguous fraction) of resources such that neighboring nodes do not share resources. A k interval coloring is one where every node gets at least a $1/k$ fraction of the resources.

In particular, we focus on the case where the set of resources to be shared is time (i.e. computing a TDMA schedule). The discrete beeping model assumes all nodes agree on a period of length T, which is composed of Q slots of length μ where slot boundaries are synchronized. However, the lack of synchronized clocks implies the periods of different nodes are not aligned, and hence the first slot of a period for node u could be in the middle of the period for node v. Therefore, although nodes agree on the set of resources to be shared (Q time slots), they do not agree on an ordering of these resources. To sidestep this problem we will require interval coloring to output a tuple $\langle p_v, I_v \rangle$ for each node v, where p_v is the offset with respect to the period start of node v, and I_v is the interval length. These tuples should be such that for every pair of neighbors $\{u, v\} \in E$, the intervals $[\mathring{p_v} - I_v, \mathring{p_v}]$ and $[\mathring{p_u} - I_u, \mathring{p_u}]$ are disjoint for every period. Analogous to $\mathcal{O}(\Delta)$-vertex colorings, we are interested in $\mathcal{O}(\Delta)$-interval colorings, where each node gets assigned at least a $\Omega(1/\Delta)$ fraction of the resources.

Hardness of Interval Coloring. Discrete interval coloring is strongly related to vertex coloring. For each node v let $\langle p_v, I_v \rangle$ be the tuple output by an interval coloring at node v. The definition of interval coloring implies that for any two neighbors u and v it holds that $\mathring{p_v} \neq \mathring{p_u}$. Therefore, we can define a valid vertex coloring by assigning to each node v the color $c_v = \mathring{p_v} \pmod{Q}$. Observe that if $Q \in \Theta(\Delta)$, this is a $\mathcal{O}(\Delta)$-vertex coloring. Hence, even in executions where all nodes have either synchronized clocks or wakeup at the same time, a $\mathcal{O}(\Delta)$-interval coloring is at least as hard as $\mathcal{O}(\Delta)$-vertex coloring.

4 Continuous Interval Coloring

We essentially use the same model as Motskin et al. [17], and adhering to it we also assume each node v knows its own degree $d(v)$ and the maximum degree of its 1-hop neighbors $d^{\max}(v)$. Motskin et al. [17] described a randomized algorithm that solves continuous interval coloring and terminates with high probability in a logarithmic number of periods. In contrast, we present a randomized algorithm that solves the same

problem but terminates almost surely in a constant number of periods. While describing the algorithm we expose the flaws of this model that make such an algorithm possible.

Algorithm Description. Since nodes can emit an infinitely short beep at any point in time, then if two nodes choose to beep at random times in the interval $[0, T]$, their beeps will collide with probability zero (i.e. the probability that two samples from a continuous uniform distribution are equal is zero). We will exploit this property with the greedy algorithm BEEPFIRST, described in detail in Algorithm 1. Informally speaking, the BEEPFIRST algorithm searches for the first available time where a node can beep while respecting a buffer of size b_v around existing beeps. To ensure that no two nodes choose the same time to beep, the buffer size and starting time are randomized with a continuous variable.

More precisely, the algorithm has a parameter $\varepsilon \in (0, 1)$ which affects the size of the resulting intervals. In the initialization state, each node v sets its interval length to $I_v = (1-\varepsilon)T/2(d^{\max}(v)+1)$ and chooses $\varepsilon_v \in \mathfrak{U}[0, \varepsilon]$ to randomize its start time and set its buffer length to $b_v = (1 - \varepsilon_v)T/2(d(v) + 1)$.

In the searching state, nodes listen for one full time period T recording the phases at which beeps are heard. If a node hears no beeps in this first period it sets $p_v = 0$ and goes to the stable state. Otherwise nodes search for the first phase p_v such that (i) in the previous period no other node beeped in the interval $[p_v - b_v, p_v + b_v]$, and (ii) in this period no other node beeps on the interval $[p_v - b_v, p_v]$. Once such a phase is found, nodes beep to reserve it and listen for whatever remains of the period, switching to the stable state. Once a node becomes stable, it remains stable thereafter, beeping at the same phase every period.

Algorithm 1. BEEPFIRST running at node v

1: $\varepsilon_v \leftarrow \mathfrak{U}(0, \varepsilon)$ ▷ Initialize
2: $I_v \leftarrow (1-\varepsilon)\frac{T}{2(d^{\max}(v)+1)}$, $b_v \leftarrow (1-\varepsilon_v)\frac{T}{2(d(v)+1)}$
3: `listen`(ε_v) `(* randomized start time *)`
4: $S \leftarrow$ `listen`(T) ▷ Search
5: $p_v \leftarrow 0$
6: **while** $\exists$ beep in $S[p_v - b_v, p_v + b_v]$ **do**
7: $\quad t_v \leftarrow p_v$
8: $\quad p_v \leftarrow b_v +$ time of last beep in $S[p_v - b_v, p_v + b_v]$
9: $\quad S \leftarrow S \cup$ `listen`$(p_v - t_v)$
10: **end while**
11: `beep`, `listen`$(T - p_v)$ ▷ Stable
12: **loop**
13: $\quad$ `listen`(p_v), `beep`, `listen`$(T - p_v)$
14: **end loop**

For each node v in the searching state, the separation between beeps heard by v is at most $2b_v$, otherwise it would have exited the search state. Assume in a period node v hears at most one beep from each neighbor (the same result can be proved without this assumption with a slightly more technical argument). Therefore node v hears at most

$d(v)$ beeps in one period, which means that after time $d(v)2b_v < T$ in the searching state node v finds a proper phase to beep and enters the stable state.

Lemma 1. *The searching state of* BEEPFIRST *lasts less than one period.*

By construction node v will select $p_v = 0$ or $p_v = p_u + b_v$ where p_u is the phase of node u. However, recall that both the starting time and the buffer length are randomized using a continuous probability distribution. Therefore, with probability one, no two nodes will ever select the same phase. (The same argument is used by Motskin et al. [17] to prove that neighbors "pick the *exact* same start time with probability 0".) Which is captured by the following proposition.

Proposition 2. *Given a pair of nodes u and v (where $u \neq v$) at any point during the execution of* BEEPFIRST *almost surely $\mathring{p}_u \neq \mathring{p}_v$.*

From Proposition 2 it follows that given two neighboring nodes, one selects an earlier phase than the other. This fact can be leveraged to show that the intervals produced by BEEPFIRST do not overlap.

Lemma 3. *Let u and v be two neighboring nodes in a stable state of* BEEPFIRST*, then their intervals do not overlap ($\mathring{p}_u \notin [\mathring{p}_v - I_v, \mathring{p}_v + I_v]$).*

We can tie Lemmas 1 and 3 together in the following theorem.

Theorem 4. *The* BEEPFIRST *algorithm computes a $\mathcal{O}(\Delta)$-interval coloring almost surely in $O(1)$ time.*

Observe that if instead of setting the interval length in the initialization phase, we delayed it until the stable phase by setting it to the largest value such that $[p_v - I_v, p_v + I_v]$ does not contain any beeps, we would get a slightly stronger result which does not require knowledge of $d^{\max}(v)$. The BEEPFIRST algorithm hints at two flaws in this model (i) It assumes knowledge of $d(v)$ and $d^{\max}(v)$, where neither is trivial to compute. (ii) The algorithm's correctness relies on computation with arbitrary real numbers and sampling from continuous probability distributions.

5 Discrete Interval Coloring

We now turn our attention to a more realistic model where beeps occur at discrete times and have a minimum length, thus the probability distributions involved are discrete and finite. We present a Las Vegas randomized algorithm for $\mathcal{O}(\Delta)$-interval coloring that terminates with high probability in $\mathcal{O}(\log n)$ periods. This requires $Q \geq \Delta$ and in particular we assume $Q = \kappa\Delta$ where κ is a large enough constant ($\kappa \geq 3/\eta$ suffices, for η to be fixed later).

Algorithm Description. The JITTERANDJUMP algorithm relies on three key insights: (i) The number of beeps heard by a node is a good estimate of its degree. (ii) By adding a small random jitter to every beep, neighboring nodes which beep at the same slot can detect the collision with constant probability. (iii) If a node jumps into a random

slot which is surrounded by "enough" empty slots it finds a non-overlapping interval assignment with constant probability.

The detail pseudo-code is presented in Algorithm 2, in the following paragraphs we give an informal description. All nodes executing are initially uncolored, and they become colored when they believe to have found a non-overlapping interval. Except for the first period (where nodes listen without beeping), all nodes beep once per period. Therefore in a single period a node can hear at most two beeps per neighbor, and it follows that if $\tilde{d}_v$ is the number of beeps observed by node v during a period, then $1 \leq \tilde{d}_v \leq 2d(v)$.

To resolve collisions, if node v has decided to beep at the slot p_v, it chooses at random $jitter_v \in \mathfrak{U}[0..1]$, and beeps at $p_v + jitter_v$ instead. If a colored node detects a beep one slot before, or two slots after its own beep, it becomes uncolored.

Each node v sets the buffer length $b_v = \eta \frac{Q}{\tilde{d}_v + 1}$ to a fraction of the period proportional to its degree estimate, where η is a sufficiently small constant (we will show that any $\eta \leq 1/16$ suffices). Using the information collected in the previous period, node v computes a set of *free slots* F_v. A free slot $s \in F_v$ is one where no beep was heard in the $b_v + 2$ slots preceeding it, and the $b_v + 1$ following it. An uncolored node v selects a slot p_v to beep uniformly at random from the set of free slots F_v. If after beeping node v determines no other node is in the interval $[p_v - b_v, p_v]$ it becomes colored.

Algorithm 2. JITTERANDJUMP running at node v

```
1:  colored_v ← false
2:  S ← listen[Q]
3:  d̃_v ← max(|S|, 1)
4:  b_v ← η Q/(d̃_v + 1)
5:  loop
6:      if not colored_v then
7:          F_v ← {p | S ∪ {p_v} [p − b_v − 2, p + b_v + 1] = ∅}
8:          p_v ← 𝔘_{F_v}
9:      end if
10:     jitter_v ← 𝔘[0..1]
11:     S ← listen[p_v + jitter_v − 1] ∪ beep ∪ listen[Q − p_v − jitter_v]
12:     I_v ← max s s.t. S[p_v − s, p_v] = ∅
13:     d̃_v ← max(|S|, 1)
14:     b_v ← η Q/d̃_v
15:     if S[p_v − b_v, p_v + b_v] = ∅ then
16:         colored_v ← true
17:     else if S[p_v − 1, p_v + 2] ≠ ∅ then
18:         colored_v ← false
19:     end if
20: end loop
```

Two neighboring nodes are *colliding* if they beep at the same slot. Every period, each nodes selects independently at random a jitter which affects where they beep. It is possible to show that two collided nodes will detect the collision and become uncolored with constant probability (proof omitted).

Lemma 5. *If neighboring nodes u and v collide in* JITTERANDJUMP*, they become uncolored in the next period with probability at least $\frac{1}{2}$.*

By adjusting κ and η appropriately, it is possible to guarantee that the number of free slots observed by each node is a constant fraction of the number of slots.

Proposition 6. *If $\kappa \geq 4/\eta$ and $\eta \leq 1/3$ then $|F_v| \geq (1-3\eta)Q$ for every node v.*

We have already established that the degree estimate is an upper bound on the real degree; we also show that with constant probability it is a lower bound on the number of uncolored nodes. To do so we use the following result concerning the classical balls and bins problem (proof omitted).

Theorem 7. *When placing m balls randomly into n bins, if $n \geq m \geq 12$ then with probability more than $\frac{1}{2}$ the number of occupied bins is at least $\frac{m}{4}$.*

Using this result we can prove the following lemma.

Lemma 8. *With probability $\frac{1}{2}$ the number of beeps observed by a node is at least a quarter of the number of its uncolored neighbors.*

Proof. Fix node v and let $P \subseteq N(v)$ be its uncolored neighbors. We want to show $\mathbb{P}\left[\tilde{d}_v > |P|/4\right] \geq \frac{1}{2}$.

Each node $u \in P$ beeps at random in F_u and if $\kappa \geq 4/\eta$ then from Proposition 6 $|F_u| \geq (1-3\eta)Q = (1-3\eta)\kappa\Delta$. If we let $\eta \leq 1/16$ then $\kappa \geq 1/(1-3\eta)$ and thus $|F_u| \geq \Delta$.

On the other hand, the probability of collisions (and a lower degree estimate $\tilde{d}_v$) is increased if $\forall u, w \in P\ F_u = F_v$. In other words, if $|P| \leq \Delta$ beeps are randomly distributed in $|F_v| \geq \Delta$ slots, and assuming enough beeps theorem 7 implies that with probability $\frac{1}{2}$ the number of occupied slots is $|P|/4$. □

To argue termination we partition nodes into `good` and `bad` nodes. Informally, a `good` node is one which, modulo the jitter, continues to beep at the same slot in the rest of the execution.

Definition 1. Node v is `good` if it is colored and there does not exist a neighboring node $u \in N(v)$ with a phase p_u such that $|\mathring{p_u} - \mathring{p_v}| \leq 1$; otherwise v is `bad`.

By definition, once a node becomes `good` no neighboring node is colliding with it. Since nodes listen before beeping and always beep at slots which were previously unoccupied, it is not surprising that once a node becomes `good` it remains `good` thereafter (proof omitted).

Lemma 9. *Once a node is* `good`*, it remains* `good` *for the rest of the execution.*

We classify `bad` nodes further as colored and uncolored. First we consider the easier case of colored `bad` nodes.

Lemma 10. *A colored* `bad` *node becomes* `good` *or uncolored with probability $\geq \frac{1}{2}$.*

Proof. Fix a colored bad node v. Since it is bad and uncolored, then by definition a nonempty set of its neighbors $P \subseteq N(v)$ beep at the same slot as u.

If all nodes in P are uncolored, then they all jump to a random slot and node v becomes good. Otherwise there exists a colored node $u \in P$. However by Lemma 5 with probability $\frac{1}{2}$ in the next period both nodes detect the collision and become uncolored. □

Now we consider uncolored bad nodes.

Lemma 11. *An uncolored* bad *node becomes* good *with probability* $\geq \frac{1}{2}e^{-\frac{16\eta}{1-3\eta}}$.

Proof. Fix an uncolored bad node v. Let B_u be the event that node u choses to beep in the interval $[p_v - b_v, p_v + b_v]$. In other words, B_u is the event that node u interferes with the beep of node v. By definition $\mathbb{P}[B_u] \leq \frac{2b_v}{|F_u|}$, and from Proposition 6 $|F_u| \geq Q(1-3\eta)$ and thus $\mathbb{P}[B_u] \leq \frac{2b_v}{Q(1-3\eta)} \leq \frac{2\eta}{\tilde{d}_v(1-3\eta)}$.

Let G_v be the event that node v becomes good. Node v becomes good unless a nonempty subset of its (uncolored) neighbors pick a slot that interferes with its beep. Hence $\mathbb{P}[G_v] = \prod_{u\in P}\mathbb{P}[\neg B_u]$ where $P \subseteq N(v)$ are the uncolored neighbors of v.

Let P_v be the event that the number of beeps observed by v is at least one quarter of the number of its uncolored neighbors, that is $\tilde{d}_v \geq |P|/4$. We show that conditioned on P_v, node v becomes good with constant probability.

$$\mathbb{P}[G_v|P_v] = \prod_{u\in P}\mathbb{P}[\neg B_u|P_v] = \prod_{u\in P}(1-\mathbb{P}[B_u|P_v]) \geq \left(1-\frac{8\eta}{|P|(1-3\eta)}\right)^{|P|} \geq e^{-\frac{16\eta}{1-3\eta}}$$

Where the last inequality holds for sufficiently small $\eta \leq \frac{1}{16}$. Finally from Lemma 8 we have $\mathbb{P}[P_v] \geq \frac{1}{2}$, hence $\mathbb{P}[G_v] \geq \mathbb{P}[G_v|P_v]\,\mathbb{P}[P_v] \geq \frac{1}{2}e^{-\frac{16\eta}{1-3\eta}}$. □

From Lemmas 10 and 11, after two periods a bad node becomes good with constant probability. Therefore the probability that a node remains bad drops off exponentially with the number of periods. Using standard arguments one can show that a bad node becomes good with high probability after $\frac{6}{e^{-\frac{16\eta}{1-3\eta}}}\log n \in \mathcal{O}(\log n)$ rounds.

We now show that each node is assigned a "large" fraction of the slots.

Lemma 12. *Let v be a* good *node, then* $I_v \geq \eta\frac{Q}{2d^{\max}(v)+1}$.

Proof. Consider the period when v became colored. By construction in that period node v observed no beeps in the interval $[p_v - b_v, p_v]$, thus $I_v \geq \eta\frac{Q}{\tilde{d}_v+1}$ in that period.

Fix a node $u \in N(v)$. Node u will only select to beep in phases that respect a buffer of size $b_u + 2 = \eta\frac{Q}{\tilde{d}_u+1} + 2$ before the beep of node v. So independent of the jitter, node v will never observe a beep of u within within b_u of its phase. Finally, since $\forall u \in V$ it holds that $\tilde{d}_u \leq 2d^{\max}(v)$, and hence $I_v \geq \eta\frac{Q}{2d^{\max}(v)+1}$ in all subsequent periods. □

This leads to our main theorem.

Theorem 13. *The* JitterAndJump *algorithm computes a $\mathcal{O}(\Delta)$-interval coloring with high probability in $\mathcal{O}(\log n)$ periods.*

5.1 Dynamic Graphs

We turn our attention to dynamic graphs, where nodes and edges are added and removed throughout the execution. Adding nodes or edges is analogous to waking up, which is already handled gracefully by JITTERANDJUMP. However this is not the case for node or edge removals. In particular, once the algorithm has stabilized to an $\mathcal{O}(\Delta)$-interval coloring, the interval of each node is not guaranteed to increase, even if sufficiently many nodes leave and the new maximum degree becomes $\Delta' \ll \Delta$.

A natural solution would be to go back to an uncolored state when the degree estimate falls below a certain threshold. However, colliding nodes can cause the degree estimate to drop artificially, even when no nodes or edges are removed. Moreover in some cases, the colliding nodes are not aware of each other and can remain collided forever despite jittering. For example in a star graph, from the center's perspective the spokes may be colliding, but the spokes have no means of detecting the collision.

Algorithm description (modifications to JITTERANDJUMP*).* Regardless of the state, each node v picks a second phase p'_v at random from the free slots F_v. As before, node v will beep at $p_v + jitter_v$, but additionally also beep at p'_v. Let $S_v(i)$ be the set of slots where node v heard a beep in period i. We define $d^*_v(i) = \max_{j\in[i-r,i])} |S_v(j)|$ as the maximum number of beeps over a moving window of the last r periods. At period i we update the degree estimate by taking the maximum of the current beep count and $d^*_v(i)$ ($\tilde{d}_v = \max(\tilde{d}_v, d^*_v(i))$). Finally, if $d^*_v(i) < \frac{\tilde{d}_v}{16}$ we set $\tilde{d}_v = d^*_v(i)$ and uncolor node v.

Since nodes beep twice at every period then for every period i, $S_v(i) \leq 4d(v)$. In executions where the degree estimate doesn't decrease, the analysis of Section 5 still holds, albeit with slightly different constants. To prove correctness we need to show that with sufficiently high probability the degree estimate will decreases if and only if the degree drops by a large enough factor.

From proposition 6 the number of free slots is $|F_v| \geq (1-3\eta)Q = (1-3\eta)\kappa\Delta$, and since $\kappa \geq \frac{1}{1-3\eta}$ then $|F_v| \geq \Delta$. Given that a node v has $d(v)$ neighbors, and each neighbor beeps at least once per period in a random slot (at most twice), we are interested in the probability that the beeps observed account for a constant fraction of the neighbors. This is essentially the same scenario described by lemma 8 which can be viewed as an occupancy problem. Using theorem 7 we can show that with probability at least $\frac{1}{2}$ the number of beeps observed is at least $d(v)/4$.

Hence, at every period i we have $|S_v(i)| \geq d(v)/4$ with probability $\geq \frac{1}{2}$. Since the degree estimate is computed using the information of the last r periods, the degree estimate decreases only if in the last r periods the beep count observed was below $\tilde{d}_v/16$. However, unless the real degree has decreased by a constant factor, this happens with probability less than $\frac{1}{2}^r$. On the other hand, if the real degree decreases by a large enough factor, the degree count observed for the next r periods will be at most four times the real degree, which will cause the degree estimate to decrease with certainty after r periods.

Finally,y setting $r \in \mathcal{O}(\log 1/\varepsilon)$ the same argument used before can be used to prove the algorithm described computes an $\Omega(T/\Delta)$-interval coloring in $\mathcal{O}(\log 1/\varepsilon)$ periods with probability $1-\varepsilon$.

6 Lower Bound

We consider a stronger model, namely standard synchronous local broadcast with messages of constant size. During each slot a node sends a message of constant size and receives the set of messages sent by its neighbors. Assume every node v knows its own degree $d(v)$, the maximum degree Δ and the size of the network n, but does not have unique IDs. All nodes start the execution (wakeup) simultaneously.

The rest of this section is devoted to proving the following theorem.

Theorem 14. *Under the model described, in less than $\mathcal{O}(\log n)$ slots it is impossible to compute a $\mathcal{O}(\Delta)$-interval coloring or a $\mathcal{O}(\Delta)$-vertex coloring with high probability.*

Proof. Let $G_i = (B_i, E_i)$ be a graph on four vertices, with vertex set $B_i = \{a_i, b_i, c_i, d_i\}$ and edge set $E_i = \{(a_i, b_i), (b_i, c_i), (c_i, d_i), (a_i, c_i), (b_i, d_i)\}$. Define G as the cycle graph generated by pasting together $n/4$ copies of G_i, where $\forall i \in [0, \frac{n}{4} - 1]$ we connect component G_i with component $G_{i+1 \mod \frac{n}{4}}$ by adding the edge $(d_i, a_{(i+1 \mod \frac{n}{4})})$. G is a 4-regular graph of size n and inside every component G_i the vertices b_i and c_i have the same closed neighborhood.

Let s_u^k be the state of node u at slot k, and let m_u^k be the message sent by node u in slot k. Regardless of its state, a node can only choose to send a message amongst a set of constant size of possible messages, let c be the size of this set.

Consider a component B_i, and assume the states of b_i and c_i are identical at slot k. Since their closed neighborhood is identical, if they send the same message at slot k, they will receive the same set of messages and remain in identical states at slot $k+1$. Formally, if $s_{b_i}^k = s_{c_i}^k$ and $m_{b_i}^k = m_{c_i}^k$ then $s_{b_i}^{k+1} = s_{c_i}^{k+1}$.

Moreover, if b_i and c_i are in the same state at slot k, they choose what to send according to the same probability distribution, in particular let p_i (where $i \in [1, c]$) be the probability of sending the i^{th} message. By definition $\sum_{i=1}^{c} p_i = 1$, and thus by Cauchy-Schwarz we have $\sum_{i=1}^{c} p_i^2 \geq \frac{1}{c}$

We prove a lower bound on the probability that b_i and c_i remain in the same state in the next slot:

$$\mathbb{P}\left[s_{b_i}^{k+1} = s_{c_i}^{k+1} \mid s_{b_i}^k = s_{c_i}^k\right] \geq \mathbb{P}\left[m_{b_i}^k = m_{c_i}^k \mid s_{b_i}^k = s_{c_i}^k\right] = \sum_{i=1}^{c} p_i^2 \geq \frac{1}{c}$$

Therefore, if nodes b_i and c_i start at the same state ($s_{b_i}^0 = s_{c_i}^0$) the probability that they remain in the same state after ℓ slots is $\mathbb{P}\left[s_{b_i}^\ell = s_{c_i}^\ell \mid s_{b_i}^0 = s_{c_i}^0\right] \geq \frac{1}{c}^\ell$. If we let $\ell = \log_c \frac{n}{4}$ then $\mathbb{P}\left[s_{b_i}^\ell = s_{c_i}^\ell \mid s_{b_i}^0 = s_{c_i}^0\right] \geq \frac{4}{n}$, and thus $\mathbb{P}\left[s_{b_i}^\ell \neq s_{c_i}^\ell \mid s_{b_i}^0 = s_{c_i}^0\right] \leq 1 - \frac{4}{n}$.

Since there are no unique identifiers, initially all nodes have the same state ($\forall u, v \in V$, $s_u^0 = s_v^0$), and the probability that after ℓ slots every component B_i has $s_{b_i}^\ell \neq s_{c_i}^\ell$ is:

$$\mathbb{P}\left[\forall B_i, s_{b_i}^\ell \neq s_{c_i}^\ell\right] = \prod_{i=1}^{n/4} \mathbb{P}\left[s_{b_i}^\ell \neq s_{c_i}^\ell\right] \leq \left(1 - \frac{4}{n}\right)^{\frac{n}{4}} \leq \frac{1}{e}$$

Therefore there exists a pair of neighboring nodes that remain in the same state after ℓ slots with constant probability.

$$\mathbb{P}\left[\exists (u, v) \in E \text{ s.t. } s_u^\ell = s_v^\ell\right] \geq \mathbb{P}\left[\exists B_i \text{ s.t. } s_{b_i}^\ell = s_{c_i}^\ell\right] = 1 - \mathbb{P}\left[\forall B_i, s_{b_i}^\ell \neq s_{c_i}^\ell\right] \geq 1 - \frac{1}{e}$$

Moreover, since G is a 4-regular graph, it should ensure interval lengths of size $\Omega(Q/4) \in \Omega(Q)$. Finally, if two nodes in the same state select intervals of size $\Omega(Q)$ slots out of a total of Q slots, the probability that they select overlapping intervals is greater than a constant. Therefore with constant probability after $\Omega(\log n)$ slots there is a pair of neighboring nodes which do not have an $\mathcal{O}(\Delta)$-interval coloring.

Observe that if instead of solving interval coloring we were considering vertex coloring, the probability that two neighboring nodes select the same color out of Δ available colors is also a constant, and thus with constant probability a pair of neighboring nodes select the same color. Which concludes the proof. □

In light of the upper bound of $\mathcal{O}(\log n)$ periods presented in Section 5, the previous bound is asymptotically tight for constant degree graphs. Since each period has $Q \geq \Delta$ slots this implies a lower bound of $\Omega(\log n/\Delta)$ periods for general graphs. If we additionally assume each node beeps at most $\mathcal{O}(1)$ times per period, the same argument yields a lower bound of $\Omega(\log n/\log \Delta)$ periods for general graphs, since for each node the probability of beeping in the same slot as a neighbor is $1/\kappa\Delta$.

References

1. Awerbuch, B., Goldberg, A.V., Luby, M., Plotkin, S.A.: Network decomposition and locality in distributed computation. In: Proc. of 30th Symposium on Foundations of Computer Science (FOCS), pp. 364–369 (1989)
2. Balasundaram, B., Butenko, S.: Graph domination, coloring and cliques in telecommunications. In: Resende, M.G.C., Pardalos, P.M. (eds.) Handbook of Optimization in Telecommunications, pp. 865–890. Springer, Heidelberg (2006)
3. Barenboim, L., Elkin, M.: Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. In: Proc. of the 41st ACM Symposium on Theory of Computing, STOC (2009)
4. Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. In: Proc. 29th ACM Symposium on Principles of Distributed Computing, PODC (2010)
5. Degesys, J., Nagpal, R.: Towards desynchronization of multi-hop topologies. In: Proc. 2nd IEEE Conference Self-Adaptive and Self-Organizing Systems (SASO), pp. 129–138 (2008)
6. Degesys, J., Rose, I., Patel, A., Nagpal, R.: Desync: self-organizing desynchronization and TDMA on wireless sensor networks. In: Proc. 6th Conference on Information Processing in Sensor Networks (IPSN), p. 20 (2007)
7. Flury, R., Wattenhofer, R.: Slotted programming for sensor networks. In: Proc. 9th Conference on Information Processing in Sensor Networks, IPSN (2010)
8. Gandham, S., Dawande, M., Prakash, R.: Link scheduling in sensor networks: Distributed edge coloring revisited. In: Proc. of 24th IEEE Conference on Computer Communications (INFOCOM), pp. 2492–2501 (2005)
9. Goldberg, A.V., Plotkin, S.A., Shannon, G.E.: Parallel symmetry-breaking in sparse graphs. SIAM Journal on Discrete Mathematics 1(4), 434–446 (1988)
10. Herman, T., Tixeuil, S.: A distributed TDMA slot assignment algorithm for wireless sensor networks. In: Proc. of 1st Int. Workshop on Algorithmic Aspects of Wireless Sensor Networks, pp. 45–58 (2004)
11. Kothapalli, K., Onus, M., Scheideler, C., Schindelhauer, C.: Distributed coloring in $o(\sqrt{\log n})$ bit rounds. In: Proc. of 20th IEEE Parallel and Distributed Processing Symposium, IPDPS (2006)

12. Kuhn, F.: Local multicoloring algorithms: Computing a nearly-optimal TDMA schedule in constant time. In: Proc. of 26th Symp. on Theoretical Aspects of Computer Science, STACS (2009)
13. Kuhn, F.: Weak Graph Coloring: Distributed Algorithms and Applications. In: Proc. of 21st ACM Symposium on Parallelism in Algorithms and Architectures, SPAA (2009)
14. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing, 193–201 (1992)
15. Mecke, S.: MAC layer and coloring. In: Wagner, D., Wattenhofer, R. (eds.) Algorithms for Sensor and Ad Hoc Networks, pp. 63–80 (2007)
16. Moscibroda, T., Wattenhofer, R.: Coloring unstructured radio networks. In: Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 39–48 (2005)
17. Motskin, A., Roughgarden, T., Skraba, P., Guibas, L.: Lightweight coloring and desynchronization for networks. In: Proc. 28th IEEE Conference on Computer Communications, INFOCOM (2009)
18. Panconesi, A., Srinivasan, A.: On the complexity of distributed network decomposition. Journal of Algorithms 20(2), 581–592 (1995)
19. Ramanathan, S.: A unified framework and algorithm for channel assignment in wireless networks. Wireless Networks 5, 81–94 (1999)
20. Rhee, I., Warrier, A., Min, J., Xu, L.: DRAND: Distributed randomized TDMA scheduling for wireless ad-hoc networks. In: 7th ACM Symp. on Mobile Ad Hoc Networking and Computing (MOBIHOC), pp. 190–201 (2006)
21. Scheideler, C., Richa, A., Santi, P.: An o(log n) dominating set protocol for wireless ad-hoc networks under the physical interference model. In: Proc. 9th ACM Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC), pp. 91–100 (2008)
22. Schmid, S., Wattenhofer, R.: Algorithmic models for sensor networks. In: Proc. 14th Workshop on Parallel and Distributed Real-Time Systems, WPDRTS (2006)
23. Schneider, J., Wattenhofer, R.: A log-star distributed maximal independent set algorithm for growth-bounded graphs. In: Proc. of 27th ACM Symposium on Principles of Distributed Computing, PODC (2008)
24. Schneider, J., Wattenhofer, R.: Coloring unstructured wireless multi-hop networks. In: Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC), pp. 210–219 (2009)
25. Schneider, J., Wattenhofer, R.: A new technique for distributed symmetry breaking. In: Proc. 29th ACM Symposium on Principles of Distributed Computing, PODC (2010)
26. USC/ISI. Network Simulator 2 (NS2), http://www.isi.edu/nsnam/ns/
27. Zhang, X., Hong, J., Zhang, L., Shan, X., Li, V.O.K.: CP-TDMA: Coloring- and probability-based TDMA scheduling for wireless ad hoc networks. IEICE Transactions on Communication E91-B(1), 322–326 (2008)

Distributed Contention Resolution in Wireless Networks*

Thomas Kesselheim and Berthold Vöcking

Department of Computer Science, RWTH Aachen University, Germany
{thomask,voecking}@cs.rwth-aachen.de

Abstract. We present and analyze simple distributed contention resolution protocols for wireless networks. In our setting, one is given n pairs of senders and receivers located in a metric space. Each sender wants to transmit a signal to its receiver at a prespecified power level, e. g., all senders use the same, uniform power level as it is typically implemented in practice. Our analysis is based on the physical model in which the success of a transmission depends on the Signal-to-Interference-plus-Noise-Ratio (SINR). The objective is to minimize the number of time slots until all signals are successfully transmitted.

Our main technical contribution is the introduction of a measure called maximum average affectance enabling us to analyze random contention-resolution algorithms in which each packet is transmitted in each step with a fixed probability depending on the maximum average affectance. We prove that the schedule generated this way is only an $\mathcal{O}(\log^2 n)$ factor longer than the optimal one, provided that the prespecified power levels satisfy natural monontonicity properties. By modifying the algorithm, senders need not to know the maximum average affectance in advance but only static information about the network. In addition, we extend our approach to multi-hop communication achieving the same appoximation factor.

1 Introduction

In a wireless network, communication carried out at the same time is not physically separated. Therefore transmissions may collide due to too much interference. The Media Access Control (MAC) layer's task is to coordinate the communication such that simultaneous transmissions do not interfere but that the medium is sufficiently used to allow for optimal throughput. In this paper, we present and analyze distributed contention-resolutions protocols for this scheduling task giving worst-case guarantees.

The interference constraints are modelled by the *physical interference model* [11]. Between any two nodes of the network u and v a distance $d(u, v)$ is defined. The received signal strength decreases when this distance is increasing. More formally, if node u transmits a signal at power level p then it is received by v with

* This work has been supported by the UMIC Research Centre, RWTH Aachen University.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 163–178, 2010.

strength $p/d(u,v)^\alpha$, where the constant $\alpha > 0$ is the so-called *path-loss exponent*[1]. The node v can successfully decode this signal if the signal strength received from the intended sender is at least β times as large as the signals strengths by interfering transmissions made at the same time plus ambient noise. This is, the *Signal-to-Interference-plus-Noise Ratio (SINR)* is above some threshold $\beta \geq 0$, the so-called *gain*.

In the *interference scheduling problem*, we are given a set of n requests $\mathcal{R} \subseteq V \times V$, corresponding to pairs of nodes from a metric space. For each request $\ell \in \mathcal{R}$, we have to select a power level $p(\ell) > 0$ and a time slot $c(\ell) \in [k] := \{1, \ldots, k\}$ such that for each $\ell = (u, v) \in \mathcal{R}$ the SINR constraint

$$\frac{p(\ell)}{d(u,v)^\alpha} \geq \beta \left(\sum_{\substack{\ell'=(u',v')\in\mathcal{R} \\ c(\ell)=c(\ell')}} \frac{p(\ell')}{d(u',v)^\alpha} + N \right)$$

is fulfilled. The constant $N \geq 0$ expresses ambient noise that all transmissions have to cope with. The objective is to minimize the number of time slots k.

So, in fact, two choices are made: For each request ℓ, we have to select a power level $p(\ell)$ and a time slot in which the transmission should take place (to make the distinction clear the latter problem is also referred to as *coloring*). In this work, we focus on the coloring problem and assume the power assignment to be given such that all senders know a priori at which power to transmit. For example, powers might be given by hardware or depend on the distance between the sender and the receiver as described below.

Our objective is to calculate a schedule whose length is close to the length of the optimal schedule in the same instance. This measure was introduced by Moscibroda et al. as *scheduling complexity* $T(\mathcal{R})$ [17]. As we concentrate on the coloring problem, we compare lengths of the schedules we compute to the optimal schedule length for $\mathcal{R}$ that uses some fixed power assignment p, denoted by $T(\mathcal{R}, p)$.

In order to have an algorithm that is applicable in a realistic environment it has to work in a distributed fashion with as little information as possible. In fact, our algorithms only require static information on the network that can be spread at the time of deployment. Particularly, the number of network nodes, the clock synchronization and the power assignment can be seen as such static information. In contrast, no information about the current state of the network will be necessary. For example, communication requests arise after the deployment and an algorithm has to work without knowledge on which requests have to be served by the network and which of them were already successfully served.

In related work several schemes for assigning the powers have been used. The simplest way is to make all transmissions at the same power level, e. g., the maximum power supported by the hardware. These power assignments are called *uniform* [13,3].

[1] Typically it is assumed that $2 < \alpha < 5$. However, our analysis works for any $\alpha > 0$.

A more complex solution that still facilitates a distributed implementation at least in theory is to make the power only dependent of the distance between the respective sender and receiver (and therefore independent of the other nodes). In *linear power assignments* [9] the power for a transmission between a sender and a receiver whose distance is d is chosen proportional to d^α and thus proportional to the minimum transmission power needed to deal with ambient noise. *Square-root* (or mean) power assignments [8,12] choose the transmission power for distance d proportional to $\sqrt{d^\alpha}$. So the transmission power still grows for increasing distances but not as fast as in a linear power assignment.

For each of the power schemes mentioned above, there are specialized algorithms and several bounds on how fixing to this scheme influences the optimal schedule length. Most of these algorithms are centralized; so far, distributed algorithms with a provable performance guarantee are only known for linear power assignments [9] (cf. Section 1.2). Furthermore, most existing transceivers support only a relatively small, fixed number of possible power levels so that a practical implementation of both linear and square-root power assignments remain a challenge. As a consequence it is necessary to have more general algorithms which do not only work for a certain power scheme.

Our algorithms solve these issues as they work in a distributed fashion and take the power assignment as input. They do not require a certain power scheme but work for every power assignment satisfying the following natural conditions. First, it has to be *non-decreasing* and *sublinear* or *linear*. That means if $d(\ell) \leq d(\ell')$ for two requests $\ell, \ell' \in \mathcal{R}$ then

$$p(\ell) \leq p(\ell') \quad \text{and} \quad \frac{p(\ell)}{d(\ell)^\alpha} \geq \frac{p(\ell')}{d(\ell')^\alpha} \ . \tag{1}$$

So the transmission power of ℓ' has to be at least as large as the one for ℓ. At the same time, the received power at the receiver of ℓ' must not be larger than the one at the receiver of ℓ. This monotonicity condition is very natural and is fulfilled by all previously studied power assignments, particularly the ones mentioned above.

The second condition is that powers are chosen sufficiently large so that ambient noise plays a minor part compared to interference. In particular, we assume the power received at any receiver is at least some constant factor higher than the minimum power that is needed to deal with noise (βN). To simplify notation, we assume this constant factor to be 2. So for all requests $\ell \in \mathcal{R}$

$$\frac{p(\ell)}{d(\ell)^\alpha} \geq 2\beta N \ . \tag{2}$$

This ensures that we actually deal with conflicts due to too large interference and not due to too weak transmission power. Previous approaches also used this assumption but stated it rather implicitly. Indeed it may be violated by uniform and square-root power assignments when considering too large distances. But in this case the problem consists of rather dealing with noise than with interference.

1.1 Our Contribution

We introduce a new measure called *maximum average affectance* $\bar{A}(\mathcal{R}, p)$ that depends on the request set $\mathcal{R}$ and the power assignment p. This measure extends a so-called *measure of interference* for linear power assignments [9] in a non-trivial way towards general power assignments satisfying Conditions 1 and 2. It is the key for analyzing the performance of simple contention resolution protocols in wireless networks with prespecified power assignments and comparing it to the optimum that could be achieved.

For two requests $\ell = (u, v)$ and $\ell' = (u', v')$, and a power assignment p, we define the *affectance* of ℓ on ℓ' by

$$a_p(\ell, \ell') = \min\left\{1, \beta \frac{p(\ell)}{d(u, v')^\alpha} \Bigg/ \left(\frac{p(\ell')}{d(u', v')^\alpha} - \beta N\right)\right\} \ .$$

The notion of affectance was introduced by Halldórsson and Wattenhofer [13], which we extended to arbitrary power assignments and bounded by 1. When taking the noise out of consideration, it indicates which amount of interference ℓ induces at ℓ', normalized by the signal strength from the intended sender of ℓ. As a consequence the sum of affectance is at most 1 for a request set that may be assigned to same time slot.

To get the *maximum average affectance* $\bar{A}(\mathcal{R}, p)$, we take the maximum over all subsets of requests and consider the average affectance a link is exposed to from all other requests in this subset.

Definition 1. *The* maximum average affectance *of a request set* $\mathcal{R}$ *and a power assignment* p *is given by*

$$\bar{A}(\mathcal{R}, p) = \max_{M \subseteq \mathcal{R}} \operatorname*{avg}_{\ell' \in M} \sum_{\ell \in M} a_p(\ell, \ell') = \max_{M \subseteq \mathcal{R}} \frac{1}{|M|} \sum_{\ell' \in M} \sum_{\ell \in M} a_p(\ell, \ell') \ .$$

If $\mathcal{R}$ and p are clear from the context, we simply write $\bar{A}$. When replacing the average by the maximum in this definition, we would get a measure that is closely related to the measure of interference in [9].

The *maximum average affectance* enables us to derive lower bounds on the scheduling complexity for all power assignments satisfying Conditions 1 and 2. In particular, we prove $\bar{A}(\mathcal{R}, p)$ is at most a factor $\mathcal{O}(\log n)$ larger than the optimal schedule length $T(\mathcal{R}, p)$. This way it enables us to compare schedules we compute to the optimal schedule length $T(\mathcal{R}, p)$.

We use this measure to analyze random contention-resolution based algorithms. In this kind of algorithms each sender transmits with a certain probability q in each step until one of the transmissions has successfully been received. We first prove a stability result. If $q \leq 1/4\bar{A}$, all transmissions are successful within $\mathcal{O}(\log n/q)$ time slots whp[2]. Thus choosing $q = 1/4\bar{A}$, we generate a schedule of length $\mathcal{O}(\bar{A} \cdot \log n)$ whp, which is at most $\mathcal{O}(T(\mathcal{R}, p) \cdot \log^2 n)$.

[2] With high probability: with probability $1 - n^c$ for each constant c.

To make the algorithm applicable to a distributed setting, we present two modifications. These do not affect the schedule length vitally and we still get schedules of length $\mathcal{O}(\bar{A} \cdot \log n)$ whp. On the one hand, we extend it such that the network nodes do not have to know $\bar{A}$ anymore but adapt the transmission probability q on their own. On the other hand, we find a way to inform each sender if a transmission has successfully been received by transmitting acknowledgement packets. This is not a trivial task because these acknowledgement packets may also interfere.

Altogether, this is the first distributed algorithm to the interference scheduling problem with a guaranteed approximation ratio. The algorithm is distributed in the following sense. It can be run on all senders and receivers of a network such that during the execution no central entity is needed that spreads information about the current state of the network, e. g. which requests have to be scheduled. The nodes only need static information, namely the power assignment, a rough estimation on the total number of nodes and a synchronized clock.

As a further result, we adapt the ideas to a distributed multi-hop algorithm that allows packets to use intermediate relay nodes. For a fixed choice of paths and powers we get an $\mathcal{O}(\log^2 n)$ whp approximation for this problem as well. For most of the other approaches to scheduling such an adaptation is not possible. In particular, for uniform and square-root power assignments, no multi-hop scheduling algorithm with a provable performance guarantee was known up to now.

1.2 Related Work

Scheduling in wireless networks has been subject to reseach for more than four decades up to now. Right from the beginning random access protocols such as ALOHA [1] have been dealt with. Collision avoidance in models featuring binary conflict constraints has been deeply studied over the years (see, e. g. [4,6,7]). However, in the physical interference model the interference constraints are not binary but take all other network nodes into consideration in an additive way. Therefore, it brings about new problems and challenges. Depending on the choice of powers interesting phenomena such as nested pairs [8] can be observed. Furthermore, characterizing the optimal schedule length is much more involved.

The analysis of scheduling algorithms in the physical model that deals with arbitrarily (and not randomly) distributed network nodes has been initiated by Moscibroda and Wattenhofer [16]. However, they do not solve the interference scheduling problem but a similar one. Instead, they find and schedule a set of pairs of senders and receivers such that all network nodes are strongly connected. They prove scheduling these pairs is possible in $\mathcal{O}(\log^4 n)$ time steps (independent of the topology) with a certain (sublinear) power assignment. This approach was extended to arbitrary requests sets [17]. However, the lengths of the generated schedules were not compared to the optimal schedule length. In fact, they can be a factor of $\Omega(n)$ away from the optimal one.

There has also been much work on algorithms with an approximation guarantee compared to the optimal schedule length. Most of these algorithms are centralized.

Using uniform power assignments, Halldórsson and Wattenhofer [13] present an $\mathcal{O}(1)$ approximation in the plane compared to uniform power assignment. This problem was proved to be NP-hard by Goussevskaia et al. [10]. Andrews and Dinitz [2] in contrast compare to the optimal power assignment and present an $\mathcal{O}(\log \Delta \cdot \log n)$ approximation in the plane. Here, Δ is the ratio between the longest and the shortest distance between a sender and its receiver. They also prove the joint problem of power control and coloring to be NP-hard. Avin et al. [3] additionally show that the optimal schedule can at most be a factor $\mathcal{O}(\log P_{\max})$ shorter if the ratio between the maximum and the minimum power used is $P_{\max}$.

With special regard to linear power assignments, Fanghänel et al. [9] introduced a measure of interference I as a bound on the optimal schedule length. It holds $I = \mathcal{O}(\log \Delta \cdot \log n \cdot T(\mathcal{R}))$ in general and $I = \mathcal{O}(T(\mathcal{R}, p))$ if p is a linear power assignment. This bound is complemented by an algorithm using $\mathcal{O}(I + \log^2 n)$ time slots which results in an $\mathcal{O}(1)$ approximation in linear power assignments for sufficiently dense instances.

Chafekar et al. [5] also use linear power assignments to get an $\mathcal{O}(\log^2 \Delta \log^2 \Gamma \log n)$ approximation for the joint multi-hop scheduling and routing problem, where Γ is the maximum and the minimum transmission power used by the optimum. This result was improved by Fanghänel et al. [9] to $\mathcal{O}(\log \Delta \log^2 n)$. So far, these were the only algorithms for the multi-hop problem with a performance guarantee.

Square-root power assignments were introduced by Fanghänel et al. [8]. They proved that in the bidirectional model (featuring undirected requests) it yields schedules that are $\mathcal{O}(\log^{3.5+\alpha} n)$ times as long as the ones using the optimal power assignment. Halldórsson [12] improved this result to $\mathcal{O}(\log n)$ in fading metrics. He also showed that in the unidirectional model (as presented above) the resulting schedule is at most $\mathcal{O}(\log \log \Delta \cdot \log^2 n)$ away from the one using the optimal power assignment.

All of the mentioned approximation guarantees depend on Δ and indeed there are instances [12] with large Δ where the algorithm only computes a (trivial) $\Omega(n)$ approximation. Only very recently [14] an approach has been presented that achieves approximation guarantees poly-logarithmic in n. Nevertheless, optimizing transmission powers in a de-centralized way still remains a challenge. So distance-based power assignments seem to be a reasonable way for distributed environments at least in theory.

2 Distributed Single-Hop Scheduling Algorithms

Random contention-resolution algorithms are probably the most intuitive way to share limited resources among several agents in a distributed fashion. The idea is that each agent accesses the resource in any time slot with a certain

probability q until its first success. In case of a collision, none of the involved agents is successful in this round.

This idea is easily applicable in wireless networks by letting each sender transmit its packet in each time slot with probability q until the first success. Due to its simplicity, this and similar approaches are also relevant for practical applications. However, if the transmission probability q is chosen too small, time slots are not sufficiently used. In contrast, if it is chosen too large, conflicts are likely.

In this section, we present an analysis of a random contention-resolution algorithm for the interference scheduling problem. This analysis is based on the *maximum average affectance* $\bar{A}$. We start by analyzing a single time slot in which some senders transmit with probability q while the others remain silent. We prove that if q is chosen small enough, a $q/4$ fraction of the senders taking part succeed.

Lemma 1. *Given a request set $\mathcal{R}' \subseteq \mathcal{R}$. Consider a time slot in which each sender of the requests in $\mathcal{R}'$ transmits with probability $q \leq 1/4\bar{A}$, the others remain silent. Then at least $q/4 \cdot |\mathcal{R}'|$ transmissions are successful in expectation.*

Proof. For $\ell \in \mathcal{R}'$, let X_ℓ be the 0/1 random variable indicating if ℓ transmits, and X'_ℓ be the 0/1 random variable indicating if the transmission is successful.

Note that to have $X'_\ell = 1$, i. e. to make transmission ℓ successful, it suffices to have

$$X_\ell = 1 \qquad \text{and} \qquad \sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) X_{\ell'} < 1 \ .$$

By Markov inequality, we have

$$\mathbf{Pr}\left[\sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) X_{\ell'} \geq 1\right] \leq \mathbf{E}\left[\sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) X_{\ell'}\right] = \sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) q \ .$$

Let $M := \left\{\ell \in \mathcal{R}' \mid \sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) \leq 2\bar{A}\right\}$. By definition of $\bar{A}$, we know that $|M| \geq \frac{1}{2} \cdot |\mathcal{R}'|$. For a request $\ell \in M$, we have

$$\sum_{\ell' \in \mathcal{R}'} a_p(\ell', \ell) q \leq 2\bar{A}q \leq \frac{1}{2}$$

which implies that $\mathbf{E}[X'_\ell] = \mathbf{Pr}[X'_\ell = 1] = \mathbf{Pr}[X_\ell = 1] \cdot \mathbf{Pr}[X'_\ell = 1 \mid X_\ell = 1] \geq q/2$. In case $\ell \notin M$, we simply use $\mathbf{E}[X'_\ell] \geq 0$.

This yields the expected total number of successful transmissions is

$$\mathbf{E}\left[\sum_{\ell \in \mathcal{R}} X'_\ell\right] = \sum_{\ell \in \mathcal{R}} \mathbf{E}[X'_\ell] \geq \frac{q}{2} \cdot |M| \geq \frac{q}{4} \cdot |\mathcal{R}'| \ .$$

We use this lemma to analyze Algorithm 1, which takes the transmission probability q as a parameter. Each sender transmits its packet with probability q until the first success.

Algorithm 1. Scheduling using transmission probability q.

while *success* $\neq$ *true* **do**
  transmit with probability q;

The measure $\bar{A}$ allows us to derive a relation between the transmission probability q and the time until the last transmission has successfully taken place. This can be seen as a stability result for a fixed value q. We find out a value such that if $\bar{A}$ is below it, collisions do not take place too often and all packets are successfully delivered fast.

Theorem 1. *If* $q \leq 1/4\bar{A}$, *Algorithm 1 needs* $\mathcal{O}(\log n/q)$ *time slots whp.*

Proof. Let be n_t the random variable indicating the number of requests that have not been successfully scheduled in the time slots $1, \ldots, t$.

By Lemma 1, we have $\mathbf{E}[n_{t+1} \mid n_t = k] \geq k - \frac{q}{4}k$ and so

$$\mathbf{E}[n_{t+1}] \leq \sum_{k=0}^{\infty} \mathbf{Pr}[n_t = k] \cdot \left(1 - \frac{q}{4}\right) k = \left(1 - \frac{q}{4}\right) \sum_{k=0}^{\infty} k \cdot \mathbf{Pr}[n_t = k] = \left(1 - \frac{q}{4}\right) \mathbf{E}[n_t] .$$

Using $n_0 = n$, this yields

$$\mathbf{E}[n_t] \leq \left(1 - \frac{q}{4}\right)^t n \ .$$

In particular, after $4c \ln n/q$ time slots for each constant c, the expected number of remaining requests is

$$\mathbf{E}\left[n_{4c \ln n/q}\right] \leq \left(1 - \frac{q}{4}\right)^{4c \ln n/q} n \leq \left(\frac{1}{e}\right)^{c \ln n} n = n^{1-c} \ .$$

The Markov inequality yields

$$\mathbf{Pr}\left[n_{4c \ln n/q} \neq 0\right] = \mathbf{Pr}\left[n_{4c \ln n/q} \geq 1\right] \leq \mathbf{E}\left[n_{4c \ln n/q}\right] \leq n^{1-c}$$

So we need $\mathcal{O}(\log n/q)$ time slots whp.

We achieve the best result when choosing $q = 1/4\bar{A}$, which yields a schedule of length $\mathcal{O}(\bar{A} \cdot \log n)$ whp. In Section 4 we will see this yields an $\mathcal{O}(\log^2 n)$ approximation of the optimal schedule. However, two major issues prevent this algorithm from being applied in distributed scenarios. On the one hand, a suitable transmission probability has to be chosen, which requires knowing $\bar{A}$. On the other hand, senders have to know when to stop transmitting. This cannot be determined from the position of the sender node but only from the receiver. Therefore, in a distributed setting, senders have to be informed someway. In the next sections, we will present solutions to cope with these two problems.

2.1 Determining the Optimal Transmission Probability

One major drawback of Algorithm 1 is that it needs to get the transmission probability as a parameter, which has to be chosen suitably to guarantee short schedules. If senders do not know the network or the request this is not possible. We solve this problem by applying the idea of an *exponential backoff* as follows. Algorithm 2 works the same way as Algorithm 1 but does not have the parameter q anymore. Instead, it starts with a high transmission probability and reduces it if the transmission has not been successful during a longer period. That causes eventually the transmission probability to be small enough that no collisions occur.

Algorithm 2. A Distributed Single-Hop Scheduling Algorithm

$k := 0$;
while *success* $\neq$ *true* **do**
  run Algorithm 1 for $8 \ln n/q$ time slots with parameter $q = \frac{1}{4 \cdot 2^k}$;
  $k := k + 1$;

Although the algorithm is much more complex, we get a guarantee that is not essentially worse than the one for Algorithm 1.

Theorem 2. *When all nodes apply Algorithm 2, scheduling takes* $\mathcal{O}(\bar{A} \cdot \log n)$ *time slots whp.*

Due to space contraints, we have to skip this analysis and some further proofs. They can be found in the full version.

3 Sending Acknowledgements

Taking the model again into consideration, it only states feasibility of one-way communication. This yields senders do not know if a transmission has successfully been received. Our solution to this problem is to use acknowledgement packets to inform a sender that its transmission was successfully received by the intended receiver. These acknowledgement packets also need one time slot to be transmitted. In the final algorithm, they will be transmitted in even time slots, whereas the actual data packets are transmitted in odd time slots.

We need to assign powers to the acknowledgement transmissions as well. For these transmissions, the original senders act as receivers and vice versa. Using the same power as for the other transmission does not work in general, because there are instances and power assignments in which the optimal schedule length increases by $\Omega(n)$ when exchanging senders and receivers.

3.1 Dual Instances

For a request $\ell = (u, v)$, we define the *dual request* ℓ^* by (v, u). Analogously, for a request set $\mathcal{R}$ the *dual request set* $\mathcal{R}^*$ is defined by $\mathcal{R}^* = \{\ell^* \mid \ell \in \mathcal{R}\}$. For a

request set $\mathcal{R}$ and a power assignment $p\colon \mathcal{R} \to \mathbb{R}_{>0}$, we define the *dual power assignment* $p^*\colon \mathcal{R}^* \to \mathbb{R}_{>0}$ by

$$p^*(\ell^*) = \frac{p(\ell')^2}{d(\ell')^\alpha} \cdot \frac{d(\ell)^\alpha}{p(\ell)} \qquad \text{where } \ell' = \arg\max_{\ell \in \mathcal{R}} d(\ell) \ .$$

Note that if p fulfills Conditions 1, then p^* also does and $(p^*)^* = p$. Note that the ℓ' factor is only necessary to ensure Condition 2 holds. It could also be chosen much larger. So when switching to a subset of $\mathcal{R}$, we do not have to use a different dual power assignment.

We can observe that in the dual power assignment p^* the affectance of a dual request ℓ^* on another dual request ℓ'^* is bounded by the affectance of ℓ' on ℓ in the power assignment p.

Observation 3. *Given two requests $\ell, \ell' \in \mathcal{R}$ and some power assignments p, we have $a_{p^*}(\ell^*, \ell'^*) \leq 2a_p(\ell', \ell)$.*

The calculations can be found in the full version. This observation directly yields the maximum average affectance for a dual request set under the dual power assignment differs by at most a factor of 2 from the original one.

Lemma 2. *For all request sets $\mathcal{R}$ and power assignments p, we have $\bar{A}(\mathcal{R}^*, p^*) \leq 2\bar{A}(\mathcal{R}, p)$.*

So, we found a power assignment for the dual request set whose maximum average affectance is not much higher than the one for the original request set. Therefore it is suitable for the acknowledgement transmissions. For the algorithm, we again assume that all transmission powers are a priori known to the senders. This is, the receivers know the dual power assignment.

3.2 Scheduling Algorithm

In order to implement acknowledgement transmissions, we let each receiver transmit a packet back to its sender immediately in the time slot after having received a packet. However, as these transmissions may still interfere each other, each one is only transmitted with probability $1/8$. Otherwise, no acknowledgement is transmitted yielding a retransmission. Each sender only stops transmissions after having successfully received an acknowledgement. Algorithm 3 extends Algorithm 1 by these ideas. We can still adapt the approach of Algorithm 2 that assigns different values for the parameter q.

Algorithm 3. An extended algorithm implementing acknowledgements

```
while success ≠ true do
    transmit with probability q (otherwise wait one time slot);
    wait for acknowledgment (one time slot);
    if acknowledgement has been received then
        success := true;
```

Theorem 4. *If $q \leq 1/4\bar{A}$, Algorithm 3 needs $\mathcal{O}(\log n/q)$ time slots whp.*

Due to space limits, the proof can only be found in the full version. As we see, we only lose a constant factor in the schedule length when using the acknowledgement packets as above. In order to adapt the approach of Algorithm 2, the only modification needed is that for each possible value of q the algorithm has now to be run for $256 \ln n/q$ steps. However, this also changes the resulting schedule length by only a constant factor, still ensuring $\mathcal{O}(\bar{A} \cdot \log n)$ whp.

In total, we get an algorithm that is fully distributed in the sense that nodes do not need any additional information about the current state of the network. The only assumption we need is all nodes have to know a rough estimation of the total number of requests n and which powers to use and they have a synchronized clock.

4 Comparison to the Optimal Schedule

To this point, we have presented several algorithms, each with a performance bound of $\mathcal{O}(\bar{A} \cdot \log n)$ whp. However, it still remains to show that $\bar{A}$ is not far from the optimal schedule length in order to compare the performance of this algorithm to the optimal schedule. In contrast, for all similar measures this is not guaranteed – they can differ by a factor of $\Omega(n)$ from the optimal schedule length. As a matter of fact, this would also happen if we used the *maximum* instead of the *average*. In particular, we will prove that $\bar{A} = \mathcal{O}(T(\mathcal{R}, p) \cdot \log n)$. This will prove that the schedules calculated by our algorithms are at most a factor of $\mathcal{O}(\log^2 n)$ whp away from the optimal schedule.

As a first step towards this result, we consider a set of requests $\mathcal{R}$ that can be scheduled in a single time slot. Informally spoken, we add a request ℓ that is shorter than all requests in $\mathcal{R}$ but does not have to be scheduled in the same time slot as $\mathcal{R}$. We derive a bound on how much affectance this request ℓ is exposed to.

Lemma 3. *Given a set $\mathcal{R}$ of requests that may be scheduled in a single time slot using some power assignment p fulfilling Conditions 1 and 2, and another request ℓ with $d(\ell) \leq d(\ell')$ for all $\ell' \in \mathcal{R}$, then we have*

$$\sum_{\ell' \in \mathcal{R}} a_p(\ell', \ell) = \mathcal{O}(1) \ .$$

This lemma can be seen as a generalization of Theorem 1 in [9]. For the proof see the full version. By decomposing the set $\mathcal{R}$ corresponding to the schedule this directly yields the following generalization where $\mathcal{R}$ may be scheduled in T time slots.

Lemma 4. *Given a set $\mathcal{R}$ of links that may be scheduled in T slots using some power assignment p fulfilling Conditions 1 and 2, and another request ℓ with $d(\ell) \leq d(\ell')$ for all $\ell' \in \mathcal{R}$, then we have*

$$\sum_{\ell' \in \mathcal{R}} a_p(\ell', \ell) = \mathcal{O}(T) \ .$$

For the same situation as above, we now bound the sum of affectance that ℓ causes at all requests in $\mathcal{R}$.

Lemma 5. *Given a set $\mathcal{R}$ of links that may be scheduled in T time slots using power assignment p and another link ℓ with $d(\ell) \leq d(\ell')$ for all $\ell' \in \mathcal{R}$, then we have*

$$\sum_{\ell' \in \mathcal{R}} a_p(\ell, \ell') = \mathcal{O}(T \cdot \log n) \ .$$

Proof. To prove this lemma, we make use of the results on dual instances we presented in Section 3.1. For the the dual instance $\mathcal{R}^*$ and the dual power assignment p^* we showed

$$\sum_{\ell' \in \mathcal{R}} a_p(\ell, \ell') \leq 2 \sum_{\ell'^* \in \mathcal{R}^*} a_{p^*}(\ell'^*, \ell^*) \ .$$

Furthermore p^* fulfills Conditions 1 and 2. This allows us to apply Lemma 4 and get

$$\sum_{\ell'^* \in \mathcal{R}^*} a_{p^*}(\ell'^*, \ell^*) = \mathcal{O}(T(\mathcal{R}^*, p^*)) \ .$$

So it only remains to show that $T(\mathcal{R}^*, p^*) = \mathcal{O}(T(\mathcal{R}, p) \cdot \log n)$. Let $\mathcal{R}_1, \ldots, \mathcal{R}_T$ be the decomposition of $\mathcal{R}$ made by the schedule. We know that each of the sets $\mathcal{R}_t$ fulfills the SINR constraint using the power assignment p. This implies that $\bar{A}(\mathcal{R}_t, p) \leq 1$ for all $t \in [T]$.

By Lemma 2, we have $\bar{A}(\mathcal{R}_t^*, p^*) \leq 2$. Thus, using Algorithm 1 there is a schedule of length $\mathcal{O}(\log n)$ for each request set $\mathcal{R}_t^*$ using power assignment p^*. A concatenation of these schedules gives us a schedule of length $\mathcal{O}(T \cdot \log n)$ for the entire set $\mathcal{R}^*$.

The combination of both above lemmas allows us to compare $\bar{A}(\mathcal{R}, p)$ to the optimal schedule length $T(\mathcal{R}, p)$.

Theorem 5. *Given a set $\mathcal{R}$ of links that may be scheduled in T time slots using some power assignment p fulfilling Conditions 1 and 2. Then $T = \Omega(\bar{A}(\mathcal{R}, p)/\log n)$.*

Proof. We show that $\bar{A}(\mathcal{R}, p) = \mathcal{O}(T \cdot \log n)$. Let be $\mathcal{R}' \subseteq \mathcal{R}$ and $\mathcal{R}' = \{\ell_1, \ldots, \ell_{\bar{n}}\}$ with $d(\ell_1) \leq d(\ell_2) \leq \ldots \leq d(\ell_{\bar{n}})$. Observe that in this notation Lemmas 4 and 5 yield

$$\sum_{\substack{j \in [\bar{n}] \\ j > i}} a_p(\ell_i, \ell_j) = \mathcal{O}(T) \qquad \text{and} \qquad \sum_{\substack{j \in [\bar{n}] \\ j > i}} a_p(\ell_j, \ell_i) = \mathcal{O}(T \cdot \log n) \ .$$

So we get

$$\frac{1}{|\mathcal{R}'|} \sum_{\ell \in \mathcal{R}'} \sum_{\ell' \in \mathcal{R}'} a_p(\ell, \ell') = \frac{1}{|\mathcal{R}'|} \sum_{i \in [\bar{n}]} \sum_{j \in [\bar{n}]} a_p(\ell_i, \ell_j) = \frac{1}{|\mathcal{R}'|} \sum_{i \in [\bar{n}]} \sum_{\substack{j \in [\bar{n}] \\ j > i}} a_p(\ell_i, \ell_j) + a_p(\ell_j, \ell_i) \, .$$

This yields the claim.

So, in total, this guarantees that the schedules calculated by our algorithms are at most a factor of $\mathcal{O}(\log^2 n)$ longer than the optimal one that uses the same power assignment. Interestingly, this optimal schedule is not required to use acknowledgement packets and therefore not be computable in a distributed way.

5 Multi-Hop Scheduling

In the multi-hop variant of the interference scheduling problem, packets are routed via intermediate networks nodes until reaching their final destination. As we do for the powers, we assume also the paths to be fixed. This is, each packet has a predefined path, which is for example given by routing tables in the network nodes.

Our algorithm applies the technique of random delays that has successfully been applied to scheduling in wired [15] and also in wireless [5,9] networks. That is, time is divided into time frames such that each packet attempts to cross one hop in each time frame after having waiting the initial delay at its origin node.

However, in a distributed environment, determining the maximum delay is more involved as we assume that all nodes only know static information on the network. They neither know which packets have to be scheduled in general nor which is the future path some packet will take. Our deals with this problem as follows. It works in phases. In phase k each packet is assigned a delay independently uniformly at random that is at most 2^k. The phase consists of $\mathcal{O}(2^k \log^2 n)$ time slots that are grouped to 2^{k+1} time frames each of length $2^{12} \cdot 18 \cdot \lceil \ln^2 n \rceil$.

During each of these phases Algorithm 3 is executed, where in each step each node works as a receiver if it does not decide to transmit in this step. In each time frame, each packet attempts to cross one hop. If a packet fails to cross a hop in the respective time slot, it is not considered anymore during this phase but deferred to the next one starting at the node where the failure occurred.

Theorem 6. *If T is the optimal schedule length, using the multi-hop algorithm results in a schedule length of $\mathcal{O}(T \cdot \log^2 n)$ whp.*

The formal definition of the algorithm and its analysis can be found in the full version.

6 Adaptation to Different Scenarios

6.1 Devices without Power Control

As uniform power assignments obviously fulfill Condition 1, and also Condition 2 if the power is large enough, the basic scheduling results are immediately applicable even if the senders do not support transmitting at smaller power levels than the maximum. However, to transmit the acknowledgement packets we used the dual power assignment p^*, which is not uniform in general. So, in order to apply this algorithm, devices have to be able to control their transmission power.

Nevertheless, if devices are not able of power control we can still use the key techniques. Suppose all transmissions have to be made at the same power level, say $\hat{p}$. If $\beta > 1$, we can prove the following result, which is similar to Observation 3.

Observation 7. *If ℓ and ℓ' can be transmitted in the same time slot using power $\hat{p}$ and $\beta > 1$, then $a_{\hat{p}}(\ell^*, \ell'^*) \leq \left(\frac{\beta+1}{\beta-1}\right)^{\alpha} \cdot a_{\hat{p}}(\ell, \ell')$.*

This observation yields $\bar{A}(\mathcal{R}^*, \hat{p}) = \mathcal{O}(1)$ if $\mathcal{R}$ can be scheduled in a single time slot. So, we can send the acknowledgements in a similar manner as in Algorithm 3 using again the transmission power $\hat{p}$ rather than the dual power assignment. These changes do not affect the core of the analysis but only the constant factors involved. However, this result only holds for the case that all transmissions use the same transmission power $\hat{p}$.

6.2 Bidirectional Model

In the bidirectional model [8], requests are undirected because both involved nodes act as a sender and a receiver at the same time. In order to estimate the interference between two requests ℓ and ℓ' the smallest of the distances is relevant. For this model, we can replace the affectance definition by

$$a_p(\ell, \ell') = \min\left\{1, \beta \frac{p(\ell)}{\min\{d(u,u')^\alpha, d(u,v')^\alpha, d(v,u')^\alpha, d(v,v')^\alpha\}} \Big/ \left(\frac{p(\ell')}{d(u',v')^\alpha} - \beta N\right)\right\}.$$

With this definition, the above results can be transferred to the bidirectional model. Particularly, we get distributed algorithms with an approximation factor of $\mathcal{O}(\log^2 n)$ whp.

7 Discussion and Open Problems

While previous algorithms are mostly centralized, the algorithms and analyses we presented seem to be much closer to realistic scenarios. However, we made use of three assumptions, namely the power assignment is given, each node has a rough estimation of the total number of nodes and the clocks are synchronized. These are suitable assumptions as this knowledge can be spread when deploying the network. On the contrary, the information that arises over time, e. g. which transmissions have to be made is not necessary.

Nevertheless, it is an interesting question which performance can still be achieved without this knowledge. Unfortunately, we cannot get rid of any of these assumptions in a non-trivial way. For example, for the clock synchronization the standard ALOHA trick [18] does not work. However, concerning the number of nodes and the clock synchronization there are various results in other scenarios that could possibly be transferred.

For the power assignment problem the best solution for distributed settings up to now is to take distance-based power schemes such as the square-root power assignment. With our algorithms this yields an $\mathcal{O}(\log\log\Delta \cdot \log^3 n)$ approximation compared to the optimal schedule using the optimal power assignment. Very recently [14] an approach has been presented calculate a power assignment achieving an approximation ratio that is independent of Δ. However this algorithm only works in a centralized way. So there is still much space for future research in distributed protocols for these problems.

Acknowledgements

We like to thank Alexander Fanghänel for valuable discussions and comments.

References

1. Abramson, N.: The aloha system: another alternative for computer communications. In: AFIPS '70 (Fall): Proceedings of the Fall Joint Computer Conference, November 17-19, 1970, pp. 281–285 (1970)
2. Andrews, M., Dinitz, M.: Maximizing capacity in arbitrary wireless networks in the sinr model: Complexity and game theory. In: Proceedings of the 28th Conference of the IEEE Communications Society, INFOCOM (2009)
3. Avin, C., Lotker, Z., Pignolet, Y.A.: On the power of uniform power: Capacity of wireless networks with bounded resources. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 373–384. Springer, Heidelberg (2009)
4. 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. Journal of Computer and System Sciences 45(1), 104–126 (1992)
5. Chafekar, D., Anil Kumar, V.S., Marathe, M.V., Parthasarathy, S., Srinivasan, A.: Cross-layer latency minimization in wireless networks with SINR constraints. In: Proceedings of the 8thACM International Symposium Mobile Ad-Hoc Networking and Computing (MOBIHOC), pp. 110–119 (2007)
6. Clementi, A.E.F., Crescenzi, P., Monti, A., Penna, P., Silvestri, R.: On computing ad-hoc selective families. In: Proceedings of the 5th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM-APPROX), pp. 211–222 (2001)
7. Demaine, E.D., Hajiaghayi, M.T., Feige, U., Salavatipour, M.R.: Combination can be hard: approximability of the unique coverage problem. In: Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 162–171 (2006)
8. Fanghänel, A., Kesselheim, T., Räcke, H., Vöcking, B.: Oblivious interference scheduling. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 220–229 (2009)
9. Fanghänel, A., Kesselheim, T., Vöcking, B.: Improved algorithms for latency minimization in wireless networks. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 447–458. Springer, Heidelberg (2009)
10. Goussevskaia, O., Oswald, Y.A., Wattenhofer, R.: Complexity in geometric SINR. In: Proceedings of the 8th ACM International Symposium Mobile Ad-Hoc Networking and Computing (MOBIHOC), New York, NY, USA, pp. 100–109 (2007)

11. Gupta, P., Kumar, P.R.: The capacity of wireless networks. IEEE Transactions on Information Theory 46, 388–404 (2000)
12. Halldórsson, M.M.: Wireless scheduling with power control. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 361–372. Springer, Heidelberg (2009)
13. Halldórsson, M.M., Wattenhofer, R.: Wireless communication is in APX. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. Part I, LNCS, vol. 5555, pp. 525–536. Springer, Heidelberg (2009)
14. Kesselheim, T.: A constant-factor approximation for wireless capacity maximization with power control in the SINR model. CoRR abs/1007.1611 (2010)
15. Leighton, F.T., Maggs, B.M., Rao, S.B.: Packet routing and job-shop scheduling in O(congestion+dilation) steps. Combinatorica (1994)
16. Moscibroda, T., Wattenhofer, R.: The complexity of connectivity in wireless networks. In: Proceedings of the 25th Conference of the IEEE Communications Society (INFOCOM), pp. 1–13 (2006)
17. Moscibroda, T., Wattenhofer, R., Zollinger, A.: Topology control meets SINR: The scheduling complexity of arbitrary topologies. In: Proceedings of the 7th ACM International Symposium Mobile Ad-Hoc Networking and Computing (MOBIHOC), pp. 310–321 (2006)
18. Roberts, L.G.: Aloha packet system with and without slots and capture. SIGCOMM Comput. Commun. Rev. 5(2), 28–42 (1975)

A Jamming-Resistant MAC Protocol for Multi-Hop Wireless Networks*

Andrea Richa[1], Christian Scheideler[2], Stefan Schmid[3], and Jin Zhang[1]

[1] Computer Science and Engineering, SCIDSE, Arizona State University
Tempe, AZ 85287, USA
{aricha,jzhang82}@asu.edu
[2] Department of Computer Science, University of Paderborn, D-33102 Paderborn, Germany
scheideler@upb.de
[3] Deutsche Telekom Laboratories, TU Berlin, D-10587 Berlin, Germany
stefan@net.t-labs.tu-berlin.de

Abstract. This paper presents a simple local medium access control protocol, called JADE, for multi-hop wireless networks with a single channel that is provably robust against adaptive adversarial jamming. The wireless network is modeled as a unit disk graph on a set of nodes distributed arbitrarily in the plane. In addition to these nodes, there are adversarial jammers that know the protocol and its entire history and that are allowed to jam the wireless channel at any node for an arbitrary $(1-\epsilon)$-fraction of the time steps, where $0 < \epsilon < 1$ is an arbitrary constant. We assume that the nodes cannot distinguish between jammed transmissions and collisions of regular messages. Nevertheless, we show that JADE achieves an asymptotically optimal throughput if there is a sufficiently dense distribution of nodes.

1 Introduction

The problem of coordinating the access to a shared medium is a central challenge in wireless networks. In order to solve this problem, a proper medium access control (MAC) protocol is needed. Ideally, such a protocol should not only be able to use the wireless medium as effectively as possible, but it should also be robust against attacks. Unfortunately, most of the MAC protocols today can be easily attacked. A particularly critical class of attacks are *jamming attacks* (i.e., denial-of-service attacks on the broadcast medium). Jamming attacks are typically easy to implement as the attacker does not need any special hardware. Attacks of this kind usually aim at the physical layer and are realized by means of a high transmission power signal that corrupts a communication link or an area, but they may also occur at the MAC layer, where an adversary may either corrupt control packets or reserve the channel for the maximum allowable number of

* A full version of this article including all proofs appears on arXiv.org (identifier 1007.1189).

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 179–193, 2010.

slots so that other nodes experience low throughput by not being able to access the channel. In this paper we focus on jamming attacks at the physical layer, that is, the interference caused by the jammer will not allow the nodes to receive messages. The fundamental question that we are investigating is: *Is there a MAC protocol such that for any physical-layer jamming strategy, the protocol will still be able to achieve an asymptotically optimal throughput for the non-jammed time steps?* Such a protocol would *force* the jammer to jam all the time in order to prevent any successful message transmissions. Finding such a MAC protocol is not a trivial problem. In fact, the widely used IEEE 802.11 MAC protocol already fails to deliver any messages for very simple oblivious jammers that jam only a small fraction of the time steps [3]. On the positive side, Awerbuch et al. [2] have demonstrated that there are MAC protocols which are provably robust against even massive adaptive jamming, but their results only hold for single-hop wireless networks with a single jammer, that is, all nodes experience the same jamming sequence.

In this paper, we significantly extend the results in [2]. We present a MAC protocol called JADE (a short form of "jamming defense") that can achieve a constant fraction of the best possible throughput for a large class of jamming strategies in a large class of multi-hop networks where transmissions and interference can be modeled using unit-disk graphs. These jamming strategies include jamming patterns that can be completely different from node to node. It turns out that while JADE differs only slightly from the MAC protocol of [2], the proof techniques needed for the multi-hop setting significantly differ from the techniques in [2].

1.1 Model

We consider the problem of designing a robust MAC protocol for multi-hop wireless networks with a single wireless channel. The wireless network is modeled as a *unit disk graph* (UDG) $G = (V, E)$ where V represents a set of $n = |V|$ honest and reliable nodes and two nodes $u, v \in V$ are within each other's transmission range, i.e., $\{u, v\} \in E$, if and only if their (normalized) distance is at most 1. We assume that time proceeds in synchronous time steps called *rounds*. In each round, a node may either transmit a message or sense the channel, but it cannot do both. A node which is sensing the channel may either (*i*) sense an *idle* channel (if no other node in its transmission range is transmitting at that round and its channel is not jammed), (*ii*) sense a *busy* channel (if two or more nodes in its transmission range transmit at that round or its channel is jammed), or (*iii*) *receive* a packet (if exactly one node in its transmission range transmits at that round and its channel is not jammed).

In addition to these nodes there is an adversary (who may control any number of jamming devices). We allow the adversary to know the protocol and its entire history and to use this knowledge in order to jam the wireless channel at will at any round (i.e, the adversary is *adaptive*). However, like in [2], the adversary has to make a jamming decision *before* it knows the actions of the nodes at the current round. The adversary can jam the nodes individually at will, as long

as for every node v, at most a $(1-\epsilon)$-fraction of its rounds is jammed, where $\epsilon > 0$ can be an arbitrarily small constant. That is, v has the chance to receive a message in at least an ϵ-fraction of the rounds. More formally, an adversary is called $(T, 1-\epsilon)$-*bounded* for some $T \in \mathbb{N}$ and $0 < \epsilon < 1$, if for any time window of size $w \geq T$ and at any node v, the adversary can jam at most $(1-\epsilon)w$ of the w rounds at v.

Given a node v and a time interval I, we define $f_v(I)$ as the number of time steps in I that are non-jammed at v and $s_v(I)$ as the number of time steps in I in which v successfully receives a message. A MAC protocol is called c-*competitive* against some $(T, 1-\epsilon)$-bounded adversary if, for any time interval I with $|I| \geq K$ for a sufficiently large K (that may depend on T and n), $\sum_{v \in V} s_v(I) \geq c \cdot \sum_{v \in V} f_v(I)$. In other words, a c-competitive MAC protocol can achieve at least a c-fraction of the best possible throughput.

Our goal is to design a *symmetric local-control* MAC protocol (i.e., there is no central authority controlling the nodes, and all the nodes are executing the same protocol) that has a constant-competitive throughput against any $(T, 1-\epsilon)$-bounded adversary in any multi-hop network that can be modeled as a UDG. In order to obtain a more refined picture of the competitiveness of our protocol, we will also investigate so-called k-uniform adversaries. An adversary is k-*uniform* if the node set V can be partitioned into k subsets so that the jamming sequence is the same within each subset. In other words, we require that at all times, the nodes in a subset are either all jammed or all non-jammed. Thus, a 1-uniform jammer jams either everybody or nobody in a round whereas an n-uniform jammer can jam the nodes individually at will.

In this paper, we will say that a claim holds *with high probability (w.h.p.)* iff it holds with probability at least $1 - 1/n^c$ for any constant $c \geq 1$; it holds *with moderate probability (w.m.p.)* iff it holds with probability at least $1 - 1/(\log n)^c$ for any constant $c \geq 1$.

1.2 Related Work

Due to the topic's importance, wireless network jamming has been extensively studied in the applied research fields [1,5,6,22,26,27,28,30,31,37,38,39,40], both from the attacker's perspective [6,26,27,40] as well as from the defender's perspective [1,5,6,27,28,30,38,40]—also in multi-hop settings (e.g. [21,32,42,43,44]).

Traditionally, jamming defense mechanisms operate on the physical layer [28,30,36]. Mechanisms have been designed to *avoid* jamming as well as *detect* jamming. Spread spectrum technology has been shown to be very effective to avoid jamming as with widely spread signals, it becomes harder to detect the start of a packet quickly enough in order to jam it. Unfortunately, protocols such as IEEE 802.11b use relatively narrow spreading [20], and some other IEEE 802.11 variants spread signals by even smaller factors [5]. Therefore, a jammer that simultaneously blocks a small number of frequencies renders spread spectrum techniques useless in this case. As jamming strategies can come in many different flavors, detecting jamming activities by simple methods based on

signal strength, carrier sensing, or packet delivery ratios has turned out to be quite difficult [27].

Recent work has also studied *MAC layer strategies* against jamming, including coding strategies [6], channel surfing and spatial retreat [1,41], or mechanisms to hide messages from a jammer, evade its search, and reduce the impact of corrupted messages [38]. Unfortunately, these methods do not help against an adaptive jammer with *full* information about the history of the protocol, like the one considered in our work.

In the theory community, work on MAC protocols has mostly focused on efficiency. Many of these protocols are random backoff or tournament-based protocols [4,7,17,18,25,34] that do not take jamming activity into account and, in fact, are not robust against it (see [2] for more details). The same also holds for many MAC protocols that have been designed in the context of broadcasting [8] and clustering [24]. Also some work on jamming is known (e.g., [9] for a short overview). There are two basic approaches in the literature. The first assumes randomly corrupted messages (e.g. [33]), which is much easier to handle than adaptive adversarial jamming [3]. The second line of work either bounds the number of messages that the adversary can transmit or disrupt with a limited energy budget (e.g. [16,23]) or bounds the number of channels the adversary can jam (e.g. [10,11,12,13,14,15,29]).

The protocols in [16,23] can tackle adversarial jamming at both the MAC and network layers, where the adversary may not only be jamming the channel but also introducing malicious (fake) messages (possibly with address spoofing). However, they depend on the fact that the adversarial jamming budget is finite, so it is not clear whether the protocols would work under heavy continuous jamming. (The result in [16] seems to imply that a jamming rate of $1/2$ is the limit whereas the handshaking mechanisms in [23] seem to require an even lower jamming rate.)

In the multi-channel version of the problem introduced in the theory community by Dolev [13] and also studied in [10,11,12,13,14,15,29], a node can only access one channel at a time, which results in protocols with a fairly large runtime (which can be exponential for deterministic protocols [11,14] and at least quadratic in the number of jammed channels for randomized protocols [12,29] if the adversary can jam almost all channels at a time). Recent work [10] also focuses on the wireless synchronization problem which requires devices to be activated at different times on a congested single-hop radio network to synchronize their round numbering while an adversary can disrupt a certain number of frequencies per round. Gilbert et al. [15] study robust information exchange in single-hop networks.

Our work is motivated by the work in [3] and [2]. In [3] it is shown that an adaptive jammer can dramatically reduce the throughput of the standard MAC protocol used in IEEE 802.11 with only limited energy cost on the adversary side. Awerbuch et al. [2] initiated the study of throughput-competitive MAC protocols under continuously running, adaptive jammers, but they only consider single-hop wireless networks. We go one step further by considering *multi-hop*

networks where different nodes can have different channel states at a time, e.g., a transmission may be received only by a fraction of the nodes. It turns out that while the MAC protocol of [2] can be adopted to the multi-hop setting with a small modification, the proof techniques cannot. We are not aware of any other theoretical work on MAC protocols for multi-hop networks with provable performance against adaptive jamming.

1.3 Our Contributions

In this paper, we present a robust MAC protocol called JADE. JADE is a fairly simple protocol: it is based on a very small set of assumptions and rules and has a minimal storage overhead. In fact, in JADE every node just stores a constant number of parameters, among them a fixed parameter γ that should be chosen so that the following main theorem holds:

Theorem 1. *When running* JADE *for at least* $\Omega((T \log n)/\epsilon + (\log n)^4/(\gamma\epsilon)^2)$ *time steps,* JADE *has a constant competitive throughput for any* $(T, 1-\epsilon)$*-bounded adversary and any UDG w.h.p. as long as* $\gamma = O(1/(\log T + \log\log n))$ *and (a) the adversary is 1-uniform and the UDG is connected, or (b) there are at least* $2/\epsilon$ *nodes within the transmission range of every node.*

Note that in practice, $\log T$ and $\log\log n$ are rather small so that our condition on γ is not too restrictive. Also, a conservative estimate on $\log T$ and $\log\log n$ would leave room for a superpolynomial change in n and a polynomial change in T over time.

On the other hand, we can also show the following result demonstrating that Theorem 1 essentially captures all the scenarios (within our notation) under which JADE can have a constant competitive throughput.

Theorem 2. *If (a) the UDG is not connected, or (b) the adversary is allowed to be 2-uniform and there are nodes with* $o(1/\epsilon)$ *nodes within their transmission range, then there are cases in which* JADE *is not constant competitive for any constant* c *independent of* ϵ*.*

Certainly, no MAC protocol can guarantee a constant competitive throughput if the UDG is not connected. However, it is still open whether there are simple MAC protocols that are constant competitive under non-uniform jamming strategies even if there are $o(1/\epsilon)$ nodes within the transmission range of a node.

2 Description of JADE

This section first gives a short motivation for our algorithmic approach and then presents the JADE protocol in detail.

2.1 Intuition

The intuition behind our MAC protocol is simple: in each round, each node u tries to send a message with probability p_u with $p_u \leq \hat{p}$ for some small constant $0 < \hat{p} < 1$. Consider the unit disk $D(u)$ around node u consisting of

u's neighboring nodes as well as u.[1] Moreover, let $N(u) = D(u) \setminus \{u\}$ and $p = \sum_{v \in N(u)} p_v$. Suppose that u is sensing the channel. Let q_0 be the probability that the channel is idle at u and let q_1 be the probability that exactly one node in $N(u)$ is sending a message. It holds that $q_0 = \prod_{v \in N(u)}(1 - p_v)$ and $q_1 = \sum_{v \in N(u)} p_v \prod_{w \in N(u) \setminus \{v\}}(1 - p_w)$. Hence,

$$q_1 \leq \sum_{v \in N(u)} p_v \frac{1}{1-\hat{p}} \prod_{w \in N(u)} (1-p_w) = \frac{q_0 \cdot p}{1-\hat{p}}, \quad q_1 \geq \sum_{v \in N(u)} p_v \prod_{w \in N(u)} (1-p_w) = q_0 \cdot p.$$

Thus we have the following lemma, which has also been derived in [2] for the single-hop case.

Lemma 1. $q_0 \cdot p \leq q_1 \leq \frac{q_0}{1-\hat{p}} \cdot p$.

By Lemma 1, if a node v observes that the number of rounds in which the channel is idle is essentially equal to the number of rounds in which exactly one message is sent, then $p = \sum_{v \in N(v)} p_v$ is likely to be around 1 (if $\hat{p}$ is a sufficiently small constant), which would be ideal. Otherwise, the nodes know that they need to adapt their probabilities. Thus, if we had sufficiently many cases in which an idle channel or exactly one message transmission is observed (which is the case if the adversary does not heavily jam the channel and p is not too large), then one can adapt the probabilities p_v just based on these two events and ignore all cases in which the wireless channel is blocked, either because the adversary is jamming it or because at least two messages interfere with each other (see also [19] for a similar conclusion). Unfortunately, p can be very high for some reason, which requires a more sophisticated strategy for adjusting the access probabilities.

2.2 Protocol Description

In JADE, each node v maintains a probability value p_v, a threshold T_v and a counter c_v. The parameters $\hat{p}, \gamma > 0$ in the protocol are fixed and the same for every node. $\hat{p}$ may be set to any constant value so that $0 < \hat{p} \leq 1/24$, and γ should be small enough so that the condition in Theorem 1 is met.

Initially, every node v sets $T_v := 1$, $c_v := 1$ and $p_v := \hat{p}$. Afterwards, the JADE protocol works in synchronized rounds. In every round, each node v decides with probability p_v to send a message. If it decides not to send a message, it checks the following two conditions:

- If v senses an idle channel, then $p_v := \min\{(1+\gamma)p_v, \hat{p}\}$.
- If v successfully receives a message, then $p_v := (1+\gamma)^{-1} p_v$ and $T_v := \max\{T_v - 1, 1\}$.

[1] In this paper, disks (and later sectors) will refer both to 2-dimensional areas in the plane as well as to the set of nodes in the respective areas. The exact meaning will become clear in the specific context.

Afterwards, v sets $c_v := c_v + 1$. If $c_v > T_v$ then it does the following: v sets $c_v := 1$, and if there was no round among the past T_v rounds in which v sensed a successful message transmission *or an idle channel*, then $p_v := (1+\gamma)^{-1} p_v$ and $T_v := \min\{T_v + 1, 2^{1/(4\gamma)}\}$.

As we will see in the upcoming section, the concept of using a multiplicative-increase-multiplicative-decrease mechanism for p_v and an additive-increase-additive-decrease mechanism for T_v, as well as the slight modifications of the protocol in [2], marked in italic above, are crucial for JADE to work.

3 Analysis of JADE

In contrast the description of JADE, its stochastic analysis is rather involved as it requires to shed light onto the complex interplay of the nodes all following their randomized protocol in a highly dependent manner. We first prove Theorem 1 (Sections 3.1 and 3.2) and then prove Theorem 2 (Section 3.3). In order to show the theorems, we will frequently use the following variant of the Chernoff bounds [2,35].

Lemma 2. *Consider any set of binary random variables $X_1, \ldots, X_n$. Suppose that there are values $p_1, \ldots, p_n \in [0,1]$ with $\mathbb{E}[\prod_{i\in S} X_i] \le \prod_{i\in S} p_i$ for every set $S \subseteq \{1,\ldots,n\}$. Then it holds for $X = \sum_{i=1}^n X_i$ and $\mu = \sum_{i=1}^n p_i$ and any $\delta > 0$ that*

$$\mathbb{P}[X \ge (1+\delta)\mu] \le \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \le e^{-\frac{\delta^2\mu}{2(1+\delta/3)}}.$$

If, on the other hand, it holds that $\mathbb{E}[\prod_{i\in S} X_i] \ge \prod_{i\in S} p_i$ for every set $S \subseteq \{1,\ldots,n\}$, then it holds for any $0 < \delta < 1$ that

$$\mathbb{P}[X \le (1-\delta)\mu] \le \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^\mu \le e^{-\delta^2\mu/2}.$$

Throughout the section we assume that $\gamma = O(1/(\log T + \log\log n))$ is sufficiently small.

3.1 Proof of Theorem 1

We first look at a slightly weaker form of adversary. We say a round t is *open* for a node v if v and at least one other node in its neighborhood are non-jammed (which implies that v's neighborhood is non-empty). An adversary is *weakly $(T, 1-\epsilon)$-bounded* for some $T \in \mathbb{N}$ and $0 < \epsilon < 1$ if the adversary is $(T, 1-\epsilon)$-bounded and in addition to this, at least a constant fraction of the non-jammed rounds at each node are open in every time interval of size $w \ge T$.

Theorem 3. *When running* JADE *for $\Omega([T + (\log^3 n)/(\gamma^2\epsilon)] \cdot (\log n)/\epsilon)$ rounds it holds w.h.p. that* JADE *is constant competitive for any weakly $(T, 1-\epsilon)$-bounded adversary.*

Proof. First, we focus on a *time frame* F consisting of $\alpha \log n/\epsilon$ *subframes* of size $f = \alpha[T + (\log^3 n)/(\gamma^2\epsilon)]$ each, where f is a multiple of T and α is a sufficiently large constant. The proof needs the following three lemmas. The first one is identical to Claim 2.5 in [2]. It is true because only successful message transmissions reduce T_u.

Lemma 3. *If in a time interval I the number of rounds in which a node u successfully receives a message is at most r, then u increases T_u in at most $r + \sqrt{2|I|}$ rounds in I.*

The second lemma holds for arbitrary (not just weakly) $(T, 1-\epsilon)$-bounded adversaries and will be shown in Section 3.2.

Lemma 4. *For every node u, $\sum_{v\in D(u)} p_v = O(1)$ for at least a $(1-\epsilon\beta)$-fraction of the rounds in time frame F, w.h.p., where the constant $\beta > 0$ can be made arbitrarily small.*

The third lemma just follows from some simple geometric argument.

Lemma 5. *A disk of radius 2 can be cut into at most 20 regions so that the distance between any two points in a region is at most 1.*

Consider some fixed node u. Let $J \subseteq F$ be the set of all non-jammed *open* rounds at u in time frame F (which are a constant fraction of the non-jammed rounds at u because we have a weakly $(T, 1-\epsilon)$-bounded adversary). Let p be a constant satisfying Lemma 4 (i.e., $\sum_{w\in D(v)} p_w \leq p$). Define $DD(u)$ to be the disk of radius 2 around u (i.e., it has twice the radius of $D(u)$). Cut $DD(u)$ into 20 regions $R_1, \ldots, R_{20}$ satisfying Lemma 5, and let v_i be any node in region R_i (if such a node exists), where $v_i = u$ if $u \in R_i$. According to Lemma 4 it holds for each i that at least a $(1-\epsilon\beta'/20)$-fraction of the rounds in F satisfy $\sum_{w\in D(v_i)} p_w \leq p$ for any constant $\beta' > 0$, w.h.p. Thus, at least a $(1-\epsilon\beta'')$-fraction of the rounds in F satisfy $\sum_{w\in D(v_i)} p_w \leq p$ for *every* i for any constant $\beta'' > 0$, w.h.p. As $D(v) \subseteq DD(u)$ for all $v \in D(u)$ and u has at least $\epsilon|F|$ non-jammed rounds in F, we get the following lemma, which also holds for arbitrary $(T, 1-\epsilon)$-bounded adversaries:

Lemma 6. *At least a $(1-\beta)$-fraction of the rounds in J satisfy $\sum_{v\in D(u)} p_v \leq p$ and $\sum_{w\in D(v)} p_w = O(p)$ for all nodes $v \in D(u)$ for any constant $\beta > 0$, w.h.p.*

Let us call these rounds *good*. Since the probability that u senses the channel is at least $1-\hat{p}$ and the probability that the channel at u is idle for $\sum_{w\in D(u)} p_w \leq p$ is equal to $\prod_{v\in N(u)}(1-p_v) \geq \prod_{v\in N(u)} e^{-2p_v} \geq e^{-2p}$, u senses an idle channel for at least $(1-\hat{p})(1-\beta)|J|e^{-2p} \geq 2\beta|J|$ many rounds in J on expectation if β is sufficiently small. This also holds w.h.p. when using the Chernoff bounds under the condition that at least $(1-\beta)|J|$ rounds in F are good (which also holds w.h.p.). Let k be the number of times u receives a message in F. We distinguish between two cases.

Case 1: $k \geq \beta|J|/6$. Then JADE is constant competitive for u and we are done.

Case 2: $k < \beta|J|/6$. Then we know from Lemma 3 that p_u is decreased at most $\beta|J|/6 + \sqrt{2|F|}$ times in F due to $c_u > T_u$. In addition to this, p_u is decreased at most $\beta|J|/6$ times in F due to a received message. On the other hand, p_u is increased at least $2\beta|J|$ times in J (if possible) due to an idle channel w.h.p. Also, we know from the JADE protocol that at the beginning of F, $p_u = \hat{p}$. Hence, there must be at least $\beta(2 - 1/6 - 1/6)|J| - \sqrt{2|F|} \geq (3/2)\beta|J|$ rounds in J w.h.p. at which $p_u = \hat{p}$. As there are at least $(1 - \beta)|J|$ good rounds in J (w.h.p.), there are at least $\beta|J|/2$ good rounds in J w.h.p. in which $p_u = \hat{p}$. For these good rounds, u has a constant probability to transmit a message and every node $v \in D(u)$ has a constant probability of receiving it, so u successfully transmits $\Theta(|J|)$ messages to at least one of its non-jammed neighbors in F (on expectation and also w.h.p.).

If we charge 1/2 of each successfully transmitted message to the sender and 1/2 to the receiver, then a constant competitive throughput can be identified for every node in both cases above, so JADE is constant competitive in F.

It remains to show that Theorem 3 achieves constant competitiveness for any time interval exceeding $|F|$. First, note that all the proofs are valid as long as $\gamma \leq 1/[c(\log T + \log\log n)]$ for a constant $c \geq 2$, so we can increase T and thereby also $|F|$ as long as this inequality holds. So w.l.o.g. we may assume that $\gamma = 1/[2(\log T + \log\log n)]$. In this case, $2^{1/(4\gamma)} \leq \sqrt{|F|}$, so our rule of increasing T_v in JADE implies that $T_v \leq \sqrt{|F|}$ at any time, which is crucial for Lemma 4 to hold for a time frame starting at any time. This allows us to extend the competitive throughput result to any sequence of time frames, which completes the proof of Theorem 3. □

Now, let us consider the two cases of Theorem 1. Recall that we allow here any $(T, 1-\epsilon)$-bounded adversary and not just a weakly bounded.

Case 1: the adversary is 1-uniform and the UDG is connected. In this case, every node has a non-empty neighborhood and therefore *all* non-jammed rounds of the nodes are open. Hence, the conditions on a weakly $(T, 1-\epsilon)$-bounded adversary are satisfied. So Theorem 3 applies, which completes the proof of Theorem 1 a).

Case 2: $|D(v)| \geq 2/\epsilon$ for all $v \in V$. Consider some fixed time interval I with $|I|$ being a multiple of T. For every node $v \in D(u)$ let f_v be the number of non-jammed rounds at v in I and o_v be the number of open rounds at v in I. Let J be the set of rounds in I with at most one non-jammed node. Suppose that $|J| > (1-\epsilon/2)|I|$. Then every node in $D(u)$ must have more than $(\epsilon/2)|I|$ of its non-jammed rounds in J. As these non-jammed rounds must be serialized in J to satisfy our requirement on J, it holds that $|J| > \sum_{v\in D(u)}(\epsilon/2)|I| \geq (2/\epsilon)\cdot(\epsilon/2)|I| = |I|$. Since this is impossible, it must hold that $|J| \leq (1-\epsilon/2)|I|$.

Thus, $\sum_{v\in D(u)} o_v \geq (\sum_{v\in D(u)} f_v) - |J| \geq (1/2)\sum_{v\in D(u)} f_v$ because $\sum_{v\in D(u)} f_v \geq (2/\epsilon)\cdot\epsilon|I| = 2|I|$. Let $D'(u)$ be the set of nodes $v \in D(u)$

with $o_v \geq f_v/4$. That is, for each of these nodes, a constant fraction of the non-jammed time steps is open. Then $\sum_{v \in D(u) \setminus D'(u)} o_v < (1/4) \sum_{v \in D(u)} f_v$, so $\sum_{v \in D'(u)} o_v \geq (1/2) \sum_{v \in D(u)} o_v \geq (1/4) \sum_{v \in D(u)} f_v$.

Consider now a set $U \subseteq V$ of nodes so that $\bigcup_{u \in U} D(u) = V$ and for every $v \in V$ there are at most 6 nodes $u \in U$ with $v \in D(u)$ (U is easy to construct in a greedy fashion for arbitrary UDGs and also known as a *dominating set of constant density*). Let $V' = \bigcup_{u \in U} D'(u)$. Since $\sum_{v \in D'(u)} o_v \geq (1/4) \sum_{v \in D(u)} f_v$ for every node $u \in U$, it follows that $\sum_{v \in V'} o_v \geq (1/6) \sum_{u \in U} \sum_{v \in D'(u)} o_v \geq (1/24) \sum_{u \in U} \sum_{v \in D(u)} f_v \geq (1/24) \sum_{v \in V} f_v$. Using that together with Theorem 3, which implies that JADE is constant competitive w.r.t. the nodes in V', completes the proof of Theorem 1 b).

3.2 Proof of Lemma 4

In order to finish the proof of Theorem 1, it remains to prove Lemma 4. Consider any fixed node u. We partition u's unit disk $D(u)$ into six *sectors* of equal angles from u, $S_1, ..., S_6$. Note that all nodes within a sector S_i have distances of at most 1 from each other, so they can directly communicate with each other (in $D(u)$, distances can be up to 2). We will first explore properties of an arbitrary node in one sector, then consider the implications for a whole sector, and finally bound the cumulative sending probability in the entire unit disk.

Recall the definition of a time frame, a subframe and f in the proof of Theorem 3. Fix a sector S in $D(u)$ and consider some fixed time frame F. Let us refer to the sum of the probabilities of the neighboring nodes of a given node $v \in S$ by $\bar{p}_v := \sum_{w \in S \setminus \{v\}} p_w$. The following lemma shows that p_v will decrease dramatically if $\bar{p}_v$ is high throughout a certain time interval. It needs the fact that $max_v T_v \leq \sqrt{|F|}$ (not shown here).

Lemma 7. *Consider a node v in a unit disk $D(u)$. If $\bar{p}_v > 5 - \hat{p}$ during* all *rounds of a subframe I of F, then p_v will be at most $1/n^2$ at the end of I, w.h.p.*

We omit the proof here. Given this property of the individual probabilities, we can derive a bound for the cumulative probability of an entire sector S. In order to compute $p_S = \sum_{v \in S} p_v$, we introduce three thresholds, a low one, $\rho_{green} = 5$, one in the middle, $\rho_{yellow} = 5e$, and a high one, $\rho_{red} = 5e^2$. The following three lemmas provide some important insights about these probabilities. The proof of the second one is omitted here.

Lemma 8. *For any subframe I in F and any initial value of p_S in I there is at least one round in I with $p_S \leq \rho_{green}$ w.h.p.*

Proof. We prove the lemma by contradiction. Suppose that throughout the entire interval I, $p_S > \rho_{green}$. Then it holds for every node $v \in S$ that $\bar{p}_v > \rho_{green} - \hat{p}$ throughout I. In this case, however, we know from Lemma 7, that p_v will decrease to at most $1/n^2$ at the end of I w.h.p. Hence, all nodes $v \in S$ would decrease p_v to at most $1/n^2$ at the end of I w.h.p., which results in $p_S \leq 1/n$. This contradicts our assumption, so w.h.p. there must be a round t in I at which $p_S \leq \rho_{green}$. □

Lemma 9. *For any time interval I in F of size f and any sector S it holds that if $p_S \leq \rho_{green}$ at the beginning of I, then $p_S \leq \rho_{yellow}$ throughout I, w.m.p. Similarly, if $p_S \leq \rho_{yellow}$ at the beginning of I, then $p_S \leq \rho_{red}$ throughout I, w.m.p.*

Lemma 10. *For any subframe I in F it holds that if there has been at least one round during the past subframe where $p_S \leq \rho_{green}$, then throughout I, $p_S \leq \rho_{red}$ w.m.p.*

Proof. Suppose that there has been at least one round during the past subframe where $p_S \leq \rho_{green}$. Then we know from Lemma 9 that w.m.p. $p_S \leq \rho_{yellow}$ at the beginning of I. But if $p_S \leq \rho_{yellow}$ at the beginning of I, we also know from Lemma 9 that w.m.p. $p_S \leq \rho_{red}$ throughout I, which proves the lemma. □

Now, define a subframe I to be *good* if $p_S \leq \rho_{red}$ throughout I, and otherwise I is called *bad*. With the help of Lemma 8 and Lemma 10 we can prove the following lemma.

Lemma 11. *For any sector S, at most $\epsilon\beta/6$ of the subframes I in F are bad w.h.p., where the constant $\beta > 0$ can be made arbitrarily small depending on the constant α in f.*

From Lemma 11 it follows that apart from an $\epsilon\beta$-fraction of the subframes, all subframes I in F satisfy $\sum_{v \in D(u)} p_v \in O(1)$ throughout I, which completes the proof of Lemma 4.

3.3 Limitations of the JADE Protocol

One may ask whether a stronger throughput result than Theorem 1 can be shown. Ideally, we would like to use the following model. A MAC protocol is called *strongly c-competitive* against some $(T, 1-\epsilon)$-bounded adversary if, for any sufficiently large time interval and any node v, the number of rounds in which v successfully receives a message is at least a c-fraction of the total number of non-jammed rounds at v. In other words, a strongly c-competitive MAC protocol can achieve at least a c-fraction of the best possible throughput for every individual node. Unfortunately, such a protocol seems to be difficult to design. In fact, JADE is not strongly c-competitive for any constant $c > 0$, even if the node density is sufficiently high.

Theorem 4. *In general, JADE is not strongly c-competitive for a constant $c > 0$ if the adversary is allowed to be 2-uniform and $\epsilon \leq 1/3$.*

Proof. Suppose that (at some corner of the UDG) we have a set U of at least $1/\hat{p}$ nodes located closely to each other that are all within the transmission range of a node v. Initially, we assume that $\sum_{u \in U} p_u \geq 1$, $p_v = \hat{p}$ and $T_x = 1$ for all nodes $x \in U \cup \{v\}$. The time is partitioned into time intervals of size T. In each such time interval, called *T-interval*, the $(T, 1-\epsilon)$-bounded adversary jams all but the first ϵT rounds at U and all but the last ϵT rounds at v. It follows

directly from Section 2.3 of [2] that if $T = \Omega((\log^3 n)/(\gamma^2\epsilon))$, then for every node $u \in U$, $T_u \leq \alpha\sqrt{T \log n/\epsilon}$ w.h.p. for some sufficiently large constant α. Thus, $T_u \leq \gamma T/(\beta \log n)$ w.h.p. for any constant $\beta > 0$ if T is sufficiently large. Hence, between the last non-jammed round at U and the first non-jammed round at v in a T-interval, the values T_u are increased (and the values p_u are decreased) at least $\beta(\log n)/(6\gamma)$ times. Thus, at the first non-jammed round at v, it holds for every $u \in U$ that

$$p_u \leq \hat{p} \cdot (1+\gamma)^{-\beta(\log n)/(6\gamma)} \leq \hat{p} \cdot e^{-(\beta/6)\log n} \leq 1/n^{\beta/6}$$

and, therefore, $\sum_{u \in U} p_u = O(1/n^2)$ if $\beta \geq 18$. This cumulative probability will stay that low during all of v's non-jammed rounds as during these rounds the nodes in U are jammed. Hence, the probability that v receives any message during its non-jammed rounds of a T-interval is $O(1/n^2)$, so JADE is not c-competitive for v for any constant $c > 0$. □

Also, in our original model, JADE is not constant competitive if the node density is too low.

Theorem 5. *In general,* JADE *is not c-competitive for a constant c independent of ϵ if there are nodes u with $|D(u)| = o(1/\epsilon)$ and the adversary is allowed to be 2-uniform.*

Proof. Suppose that we have a set U of $k = o(1/\epsilon)$ nodes located closely to each other that are all within the transmission range of a node v. Let $T = \Omega((\log^3 n)/(\gamma^2\epsilon))$. In each T-interval, the adversary never jams v but jams all but the first ϵT rounds at U. Then Section 2.3 of [2] implies that for every node $u \in U$, $T_u \leq \gamma T/(\beta \log n)$ w.h.p. for any constant $\beta > 0$ if T is sufficiently large. The nodes in U continuously increase their T_u-values and thereby reduce their p_u values during their jammed time steps. Hence, the nodes in $U \cup \{v\}$ will receive at most $\epsilon T \cdot |U| + (\epsilon T + O(T/\log n)) = \epsilon T \cdot o(1/\epsilon) + (\epsilon + o(1))T = (\epsilon + o(1))T$ messages in each T-interval on expectation whereas the sum of non-jammed rounds over all nodes is more than T. □

This implies Theorem 2. Hence, Theorem 1 is essentially the best one can show for JADE (within our notation).

3.4 Simulations

We briefly report on some simulation results that complement the theoretical insights. We assume that initially, $p_v = \hat{p} = 1/24$ for all nodes v. The nodes are distributed over a square plane of 4×4 units, and are connected in a unit disk graph manner (multi-hop). In each round, a node is jammed independently with probability $(1 - \epsilon)$. We run the simulation for a sufficiently large number of time steps indicated by the Theorem 1, where $\epsilon = 0.3$, $T = 200$, and $\gamma = 0.1$. Figure 1 (*left*) shows the throughput competitiveness of JADE as a function of the network size for a scenario with a uniform node distribution and a scenario

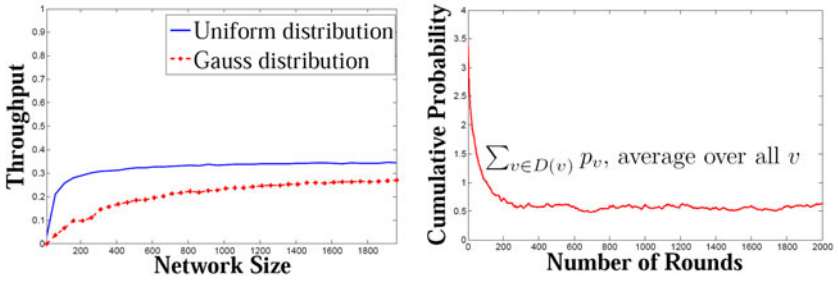


Fig. 1. *Left:* Throughput as a function of network size. *Right:* Convergence behavior for multi-hop networks (uniform distribution). For the plot, we used $n = 500$.

with a normal/Gaussian distribution. In both cases, the throughput is larger when the density is higher (20% to 40%), which corresponds to our formal insight that a constant competitive throughput is possible only if the node density exceeds a certain threshold. Moreover, we found that a constant throughput and a constant cumulative sending probability (per unit disk) is reached fast. See Figure 1 (*right*).

4 Conclusion

This paper has presented the first jamming-resistant MAC protocol with provably good performance in multi-hop networks exposed to an adaptive adversary. While we have focused on unit disk graphs, we believe that our stochastic analysis is also useful for more realistic wireless network models. Moreover, although our analysis is involved, our protocol is rather simple.

References

1. Alnifie, G., Simon, R.: A multi-channel defense against jamming attacks in wireless sensor networks. In: Proc. of Q2SWinet '07, pp. 95–104 (2007)
2. Awerbuch, B., Richa, A., Scheideler, C.: A jamming-resistant MAC protocol for single-hop wireless networks. In: Proc. of PODC '08 (2008)
3. Bayraktaroglu, E., King, C., Liu, X., Noubir, G., Rajaraman, R., Thapa, B.: On the performance of IEEE 802.11 under jamming. In: Proc. of IEEE Infocom '08, pp. 1265–1273 (2008)
4. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: Proc. of SPAA '05 (2005)
5. Brown, T., James, J., Sethi, A.: Jamming and sensing of encrypted wireless ad hoc networks. In: Proc. of MobiHoc '06, pp. 120–130 (2006)
6. Chiang, J., Hu, Y.-C.: Cross-layer jamming detection and mitigation in wireless broadcast networks. In: Proc. of MobiCom '07, pp. 346–349 (2007)
7. Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Adversarial queuing on the multiple-access channel. In: Proc. of PODC '06 (2006)

8. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. Journal of Algorithms 60(2), 115 (2006)
9. Dolev, S., Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C., Kuhn, F., Lynch, N.: Reliable distributed computing on unreliable radio channels. In: Proc. 2009 MobiHoc S3 Workshop (2009)
10. Dolev, S., Gilbert, S., Guerraoui, R., Kuhn, F., Newport, C.C.: The wireless synchronization problem. In: Proc. 28th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 190–199 (2009)
11. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a multi-channel radio network: An oblivious approach to coping with malicious interference. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 208–222. Springer, Heidelberg (2007)
12. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Secure communication over radio channels. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC), pp. 105–114 (2008)
13. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.C.: Gossiping in a multi-channel radio network. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 208–222. Springer, Heidelberg (2007)
14. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-resilient information exchange. In: Proc. of the 28th Conference on Computer Communications, IEEE Infocom 2009 (2009)
15. Gilbert, S., Guerraoui, R., Kowalski, D.R., Newport, C.C.: Interference-resilient information exchange. In: Proc. 28th IEEE International Conference on Computer Communications (INFOCOM), pp. 2249–2257 (2009)
16. Gilbert, S., Guerraoui, R., Newport, C.: Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 215–229. Springer, Heidelberg (2006)
17. Goldberg, L.A., Mackenzie, P.D., Paterson, M., Srinivasan, A.: Contention resolution with constant expected delay. J. ACM 47(6) (2000)
18. Hastad, J., Leighton, T., Rogoff, B.: Analysis of backoff protocols for mulitiple accesschannels. SIAM Journal on Computing 25(4) (1996)
19. Heusse, M., Rousseau, F., Guillier, R., Duda, A.: Idle sense: An optimal access method for high throughput and fairness in rate diverse wireless lans. In: Proc. SIGCOMM (2005)
20. IEEE: Medium access control (MAC) and physical specifications. In: IEEE P802.11/D10 (1999)
21. Jain, K., Padhye, J., Padmanabhan, V.N., Qiu, L.: Impact of interference on multihop wireless network performance. In: Proc. 9th Annual International Conference on Mobile Computing and Networking (MobiCom), pp. 66–80 (2003)
22. Jiang, S., Xue, Y.: Providing survivability against jamming attack via joint dynamic routing and channel assigment. In: Proc. 7th Workshop on Design of Reliable Communication Networks, DRCN (2009)
23. Koo, C.Y., Bhandari, V., Katz, J., Vaidya, N.H.: Reliable broadcast in radio networks: The bounded collision case. In: Proc. of PODC '06 (2006)
24. Kuhn, F., Moscibroda, T., Wattenhofer, R.: Radio network clustering from scratch. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, Springer, Heidelberg (2004)
25. Kwak, B.-J., Song, N.-O., Miller, L.E.: Performance analysis of exponential backoff. IEEE/ACM Transactions on Networking 13(2), 343–355 (2005)

26. Law, Y., van Hoesel, L., Doumen, J., Hartel, P., Havinga, P.: Energy-efficient link-layer jamming attacks against wireless sensor network mac protocols. In: Proc. of SASN '05, pp. 76–88 (2005)
27. Li, M., Koutsopoulos, I., Poovendran, R.: Optimal jamming attacks and network defense policies in wireless sensor networks. In: Proc. of Infocom '07, pp. 1307–1315 (2007)
28. Liu, X., Noubir, G., Sundaram, R., Tan, S.: Spread: Foiling smart jammers using multi-layer agility. In: Proc. of Infocom '07, pp. 2536–2540 (2007)
29. Meier, D., Pignolet, Y.A., Schmid, S., Wattenhofer, R.: Speed dating despite jammers. In: Krishnamachari, B., Suri, S., Heinzelman, W., Mitra, U. (eds.) DCOSS 2009. LNCS, vol. 5516, Springer, Heidelberg (2009)
30. Navda, V., Bohra, A., Ganguly, S., Rubenstein, D.: Using channel hopping to increase 802.11 resilience to jamming attacks. In: Proc. of Infocom '07, pp. 2526–2530 (2007)
31. Negi, R., Perrig, A.: Jamming analysis of MAC protocols. Technical report, Carnegie Mellon University (2003)
32. Noubir, G.: On connectivity in ad hoc networks under jamming using directional antennas and mobility. In: Langendoerfer, P., Liu, M., Matta, I., Tsaoussidis, V. (eds.) WWIC 2004. LNCS, vol. 2957, pp. 186–200. Springer, Heidelberg (2004)
33. Pelc, A., Peleg, D.: Feasibility and complexity of broadcasting with random transmission failures. In: Proc. of PODC '05 (2005)
34. Raghavan, P., Upfal, E.: Stochastic contention resolution with short delays. SIAM Journal on Computing 28(2), 709–719 (1999)
35. Schmidt, J., Siegel, A., Srinivasan, A.: Chernoff-Hoeffding bounds for applications with limited independence. SIAM Journal on Discrete Mathematics 8(2), 223–250 (1995)
36. Simon, M.K., Omura, J.K., Schultz, R.A., Levin, B.K.: Spread Spectrum Communications Handbook. McGraw-Hill, New York (2001)
37. Thuente, D., Acharya, M.: Intelligent jamming in wireless networks with applications to 802.11b and other networks. In: Proc. of MILCOM '06 (2006)
38. Wood, A., Stankovic, J., Zhou, G.: DEEJAM: Defeating energy-efficient jamming in IEEE 802.15.4-based wireless networks. In: Proc. of SECON '07 (2007)
39. Xu, W., Ma, K., Trappe, W., Zhang, Y.: Jamming sensor networks: attack and defense strategies. IEEE Network 20(3), 41–47 (2006)
40. Xu, W., Trappe, W., Zhang, Y., Wood, T.: The feasibility of launching and detecting jamming attacks in wireless networks. In: Proc. of MobiHoc '05, pp. 46–57 (2005)
41. Xu, W., Wood, T., Zhang, Y.: Channel surfing and spatial retreats: defenses against wireless denial of service. In: Proc. of Workshop on Wireless Security (2004)
42. Ye, S., Wang, Y., Tseng, Y.: A jamming-based MAC protocol for wireless multihop ad hoc networks. In: Proc. IEEE 58th Vehicular Technology Conference (2003)
43. Ye, S.-R., Wang, Y.-C., Tseng, Y.-C.: A jamming-based MAC protocol to improve the performance of wireless multihop ad-hoc networks. Wirel. Commun. Mob. Comput. 4(1), 75–84 (2004)
44. Zander, J.: Jamming in slotted ALOHA multihop packed radio networks. IEEE Transactions on Networking 39(10), 1525–1531 (1991)

Brief Announcement: Simple Gradecast Based Algorithms

Michael Ben-Or, Danny Dolev, and Ezra N. Hoch

The Hebrew University of Jerusalem, Israel
{benor,dolev,ezraho}@cs.huji.ac.il

Summary: Gradecast is a simple three-round algorithm presented by Feldman and Micali [4]. The current work presents two very simple algorithms that utilize Gradecast to achieve *Byzantine* agreement and to solve the Approximate agreement problem [2].

An optimal approximate agreement algorithm was presented by Fekete [3] (see also [5]), which supports up to $\frac{1}{4}n$ *Byzantine* nodes and has message complexity of $O(n^k)$, where n is the number of nodes and k is the number of rounds. Our solution to the approximate agreement problem is optimal, simple and reduces the message complexity to $O(k \cdot n^3)$, while supporting $\frac{1}{3}n$ *Byzantine* nodes.

In the *Byzantine* consensus problem each node p has an input value v_p, and all non-faulty nodes are required to reach the same output value v (*"agreement"*), s.t. if all non-faulty nodes have the same input value v' then the output value is v', *i.e.,* $v = v'$ (*"validity"*). Approximate agreement aims at reaching an agreement on a value from the Real domain, s.t. the output values of non-faulty nodes are at most ϵ apart; and are within the range of non-faulty nodes' inputs.

In both problems it is interesting to compare the round-complexity when there are $f < t$ failures. That is, what if in a specific run there are only $f < t$ failures? Can the *Byzantine* consensus / approximate agreement problems be solved faster? The answer is "yes" on both accounts. The property of terminating in accordance to the actual number of failures f is termed "early-stopping". Both algorithms presented in this paper have the early-stopping property.

The solutions presented herein all use Gradecast as a building block. Gradecast is a 3 round distributed algorithm that ensures some properties that are similar to those of broadcast. Specifically, in Gradecast there is a sender node p that sends a value v to all other nodes. Each node q's output is a pair $\langle v_q, c_q \rangle$, where v_q is the value q thinks that p has sent and c_q is q's confidence in this value. The Gradecast properties ensure that:

1. if p is non-faulty then $v_q = v$ and $c_q = 2$, for every non-faulty q;
2. for every non-faulty nodes q, q': if $c_q > 0$ and $c_{q'} > 0$ then $v_q = v_{q'}$;
3. $|c_q - c_{q'}| \leq 1$ for every non-faulty nodes q, q'.

Both algorithms use Gradecast to detect faulty nodes and ignore them in future rounds. By using Gradecast we ensure that either a *Byzantine* node z discloses its faultiness, or all non-faulty nodes see the same message from z. A very simple iterative algorithm schema solves *Byzantine* consensus and approximate agreement. These solutions are simple, optimal in their resiliency ($t < \frac{1}{3}n$), stop-early

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 194–197, 2010.

Algorithm ByzConsensus

```
    /* Initialization */
 1: set BAD := ∅;

    /* Main loop */
 2: for r := 1 to t + 1 do:
 3:     gradecast v, while ignoring messages
            from nodes in BAD;

        /* Notations */
 4:     let ⟨q, v, c⟩ represent that q
            gradecasted v with confidence c;
 5:     let maj be the value received most
            among values with confidence ≥ 1;

 6:     let #maj be the number of
            maj occurrences with confidence 2;

        /* Updates */
 7:     set v := maj;
 8:     set BAD := BAD ∪
            {q | received ⟨q, ∗, c⟩ with c ≤ 1};
 9:     if #maj ≥ n − t then break loop;
10: end for

11: if executed for < t + 1 iterations then
        participate in one more iteration;
12: return v;
```

Algorithm ApproxAgree(ϵ)

```
    /* Initialization */
 1: set BAD := ∅;

    /* Main loop */
 2: while true do:
 3:     gradecast v, while ignoring messages
            from nodes in BAD;

        /* Notations */
 4:     let ⟨q, v, c⟩ represent that q
            gradecasted v with confidence c;
 5:     let values be the multiset of received
            values with confidence ≥ 1 and add
            "0" until values contains n items;
 6:     let values′ be the multiset of received
            values with confidence 2;

        /* Updates */
 7:     set v := AVG(values);
 8:     set BAD := BAD ∪
            {q | received ⟨q, ∗, c⟩ with c ≤ 1};
 9:     if there are n − t items in values′ that
            are at most ϵ apart, then break loop;
10: end while

11: participate in one more iteration;
12: return v;
```

Fig. 1. ByzConsensus and ApproxAgree: efficient *Byzantine* consensus and approximate agreement algorithms

and are optimal in their running time (up to a constant factor induced by using Gradecast in each iteration). *I.e.,* The first algorithm solves the *Byzantine* consensus problem within $3 \cdot \min\{f+2, t+1\}$ rounds.

The second algorithm solves the approximate agreement problem, converging to $\frac{1}{n}$ within $O(\frac{\log n}{\log \log n})$ rounds. The message complexity is $O(k \cdot n^3)$ per k rounds, while tolerating $\frac{1}{3}n$ Byzantine failures, as opposed to $O(n^k)$ of the previous best known results. Moreover, the solution dynamically adapts to the number of failures at each round.

Results: Figure 1 presents both ByzConsensus which solves the *Byzantine* consensus problem, and ApproxAgree which solves approximate agreement.

The idea behind the algorithms is to use gradecast as a means of requiring the *Byzantine* nodes to "lie" at the expense of being expelled from the algorithm. That is, at each iteration a node p will gradecast its own value, and then consider the values it received: a) any node that gradecasted a value with confidence ≤ 1 will be marked as faulty (by being added to BAD), and will be ignored for the rest of the algorithm; b) any value with confidence ≥ 1 will be considered, and p will update its own value according to the values with confidence ≥ 1.

This mechanism ensures that for a faulty node z, if different non-faulty nodes consider different values for z's gradecast (for example, one considers z gradecasted "0" with confidence 1, and the other considers z gradecast's confidence to

be 0) then all non-faulty nodes will know z to be faulty, and will remove it from the algorithm. In other words, a *Byzantine* node can give contradicting values to non-faulty nodes at most once.

Theorem: ByzConsensus solves the *Byzantine* consensus problem.
Proof idea: If no *Byzantine* node "lies" in some round, then all non-faulty nodes see the same set of values and update their own value exactly the same. Thus, in the following round they will all terminate. There can be at most f rounds in which a *Byzantine* node "lies".

Approximate agreement
Given a constant ϵ the approximate agreement problem requires that: 1) *"agreement"*: $|o_p - o_q| \leq \epsilon$ for any two non-faulty nodes p, q; 2) *"validity"*: $o_p \in [L, H]$ for every non-faulty node p; where $o_p \in \Re$ and L (H resp.) is the lowest (highest resp.) input values of non-faulty nodes.

The function AVG used in Figure 1 removes the t lowest and t highest values and computes the average of the remaining values. A main property of AVG is that for any $x \leq t$: if M is a multi-set of $n - x$ values, and M_1 and M_2 contain M and additional x items (*i.e.,* M_1, M_2 differ by at most x values) then $|AVG(M_1) - AVG(M_2)| \leq (H(M) - L(M)) \cdot \frac{x}{n-2t}$.

Theorem: ApproxAgree(ϵ) solves the approximate agreement problem, and for $\epsilon = \frac{H-L}{n}$ it converges within at most $O\left(\frac{\log n}{\log \log n}\right)$ rounds.
Proof idea: Similar to ByzConsensus if a *Byzantine* node "lies" in some round it is excluded from all future rounds. Assume there are NEW_i *Byzantine* nodes that "lie" at round i. From the properties of AVG after k rounds the non-faulty nodes' values are converged to within $(H-L) \cdot \prod_{i=1}^{k} \frac{NEW_i}{n-2t}$. Thus, the *Byzantine* nodes aim to maximize $\prod_{i=1}^{k} NEW_i$ under the constraint that $\sum_{i=1}^{k} NEW_i \leq t$. Therefore, the worst case *Byzantine* behavior will assign $NEW_i = \frac{t}{k}$. Consequently, after k rounds, the non-faulty nodes' values are converged to within $\frac{H-L}{k^k}(\frac{t}{n-2t})^k$. For $\epsilon = \frac{H-L}{n}$, the above discussion proves that within $O\left(\frac{\log n}{\log \log n}\right)$ rounds, all non-faulty nodes' values are within ϵ if each other.

Additional results
In the full paper [1], in addition to the full proofs of the above results, there is also an additional algorithm. The algorithm, which is based on similar ideas solves ℓ sequential *Byzantine* consensuses within $O(t + \ell)$ rounds. The algorithm overcomes the very limiting requirement of previous results - the assumption about synchronized starts of the consensus instances (a requirement that cannot be obtained when instances stop early).

Acknowledgements. Michael Ben-Or is the incumbent of the Jean and Helena Alfassa Chair in Computer Science, and he was supported in part by the Israeli Science Foundation (ISF) research grant. Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. Danny Dolev was supported in part by the Israeli Science Foundation (ISF) Grant number 0397373.

References

1. Ben-Or, M., Dolev, D., Hoch, E.N.: Simple gradecast based algorithms. CoRR, abs/1007.1049 (2010)
2. Dolev, D., Lynch, N.A., Stark, E., Weihl, W.E., Pinter, S.: Reaching approximate agreement in the presence of faults. J. of the ACM 33, 499–516 (1986)
3. Fekete, A.D.: Asymptotically optimal algorithms for approximate agreement. In: PODC '86, pp. 73–87. ACM, New York (1986)
4. Feldman, P., Micali, S.: Optimal algorithms for byzantine agreement. In: STOC '88, pp. 148–161. ACM, New York (1988)
5. Zamsky, A.: Phd thesis (October 1998)

Brief Announcement: Decentralized Network Bandwidth Prediction

Sukhyun Song, Pete Keleher, Bobby Bhattacharjee, and Alan Sussman

UMIACS and Department of Computer Science, University of Maryland, U.S.A.

Motivation. Distributed applications can often benefit from knowledge of the bandwidth between hosts, without performing measurements between all host pairs. For example, if a peer-to-peer (P2P) computational grid system predicts pairwise bandwidth between all nodes in the system, that information could increase overall system performance by finding high-bandwidth nodes to store large scientific input or output datasets. Another possible beneficiary is a P2P online game, which can provide users a seamless gaming experience by selecting a coordinator node that has high-bandwidth connections to the players in a game region.

Ramasubramanian et. al [4] claim, from both theoretical and empirical evidence, that a metric space for Internet bandwidth almost satisfies the four-point condition (4PC) [2]. Based on this finding, they have proposed a tree-embedding model for bandwidth prediction, where bandwidth measurements are embedded as distances in an edge-weighted tree, called a *prediction tree*. The model shows that $d_T(u, v) = d(u, v)$ holds, where $d_T(u, v)$ is the distance between nodes u and v in an edge-weighted tree T, and $d(u, v)$ is the distance in a metric space defined by the formula $d(u, v) = C - BW(u, v)$, such that C is a large constant that keeps $d(u, v)$ from being negative, and $BW(u, v)$ is the bandwidth between hosts u and v.

Inspired by the tree-embedding model, our study proposes a decentralized bandwidth prediction system. We briefly describe the key ideas for the node join algorithm in such a decentralized system, using a random measurement starting point, and also describe different techniques used to increase prediction accuracy.

Algorithms. The overall design goal is to construct a prediction tree in a distributed fashion. We first show a basic centralized algorithm, then describe how it can be extended to a decentralized version.

A prediction tree starts with the first two joining nodes connected by an edge weighted by their distance. The tree grows by iteratively adding nodes as follows. For a newly joining node x, a *base node* z is chosen as a random host in the system. The algorithm selects another node y, called an *end node*, that maximizes the *Gromov product* $(x|y)_z$. The Gromov product of x and y at z, denoted $(x|y)_z$, is defined as $(x|y)_z = \frac{1}{2}(d(z, x) + d(z, y) - d(x, y))$. An inner node t_x is created and located in the graph where $d_T(z, t_x) = (x|y)_z$. The algorithm then adds x to the prediction tree by creating an edge (t_x, x) with weight $(y|z)_x$.

A naive way to find an end node y is to measure bandwidth to **all** N hosts in the system. To develop a more efficient prediction system, we reduce the number of measurements necessary for each newly joining node by constructing an *anchor*

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 198–200, 2010.

tree. An anchor tree is a rooted unweighted tree where each node represents a host in the system. Each time a node x is added to a prediction tree, x is also added to the corresponding anchor tree by becoming a child of x's *anchor node*. x's anchor node is defined as a node that was previously added to the prediction tree along with the edge that x's inner node t_x is located on. When joining, x moves up and down on the anchor tree starting from a random base node z until finding a Gromov product maximizer y. At each hop, x measures bandwidth to the currently visited node and all its neighbors to decide where it moves next. In this way, the number of measurements can be reduced to less than N, but how much less depends on the shape of the anchor tree and the positions of z and y. The number of measurements will be $O(1)$ in the best case and N in the worst case.

There have been several studies related to constructing an edge-weighted tree for a metric space that satisfies the 4PC. Since the algorithms proposed by Abraham et. al and Buneman [1,2] use a non-incremental recursive algorithm that builds the tree from scratch, unlike our incremental iterative algorithm, those algorithms cannot be used in practice when nodes dynamically join a distributed system. Compared to the algorithm with a fixed base node described by Ramasubramanian et. al [4], our algorithm starts the measurements at a random base node, which provides an indispensable underpinning for decentralization.

We last describe how the algorithm is decentralized. The key insight is that the participating nodes build an overlay network that directly matches the structure of the anchor tree. By assigning a *distance label* to each node when it joins the system, we can construct a prediction tree in a distributed fashion. Node x's distance label contains all anchor nodes on the path from the root node to x in the anchor tree. The distance label also maintains the corresponding distance values between anchor nodes on the prediction tree. Since a distance label is equivalent to a partial prediction tree, the bandwidth between two nodes can be computed via a simple computation using their distance labels. In other words, a distance label plays a similar role to the network coordinates in a network latency prediction system, such as Vivaldi [3].

Heuristics for High Accuracy. Even though the algorithm guarantees perfect accuracy for a metric space that satisfies the 4PC, directly applying it to the real world Internet could result in poor prediction accuracy, since real networks do not always satisfy the 4PC. This leads us to introduce heuristics to tolerate an imperfect metric space that does not perfectly satisfy the 4PC.

The first technique is *error minimization*. When the 4PC does not hold in a metric space, it is no longer an effective approach to find an end node that maximizes the Gromov product. So we modify the algorithm to choose the next hop in the anchor tree that minimizes the relative prediction error. At each hop, x computes its temporary position in a prediction tree for each candidate end node, For each candidate and the corresponding temporary position of x, we estimate a relative error by comparing the bandwidth data embedded in the prediction tree to the real measured bandwidth data. Then x chooses the next hop that is associated with x's temporary position that minimizes the relative error.

The second heuristic is to use a *rational transform function*. Since $d_T(u,v) > d(u,v)$ might occur in a real world network, the algorithm can predict a negative bandwidth value, which will decrease overall prediction accuracy. By estimating bandwidth not by $C - d_T(u,v)$, but instead by $\frac{C}{d_T(u,v)}$, the predicted bandwidth will always be positive even when $d_T(u,v)$ is highly overestimated.

The last technique is *deep search* for an end node choice. The base algorithm considers only direct neighbors as candidate nodes to be an end node. We can modify that to take advantage of additional candidates, including indirect neighbors. The larger pool of candidates will result in higher prediction accuracy.

Experimental Results. We have simulated our algorithms using a dataset that contains bandwidth measurements between hundreds of PlanetLab nodes, based on the same dataset used to validate the base algorithm by Ramasubramanian et. al [4]. Preliminary results demonstrate that our approach shows high prediction accuracy, with more than 80% of the node pair predictions having a relative error less than 0.5. Also, each node joining the system causes a small amount of measurement traffic.

References

1. Abraham, I., Balakrishnan, M., Kuhn, F., Malkhi, D., Ramasubramanian, V., Talwar, K.: Reconstructing approximate tree metrics. In: PODC '07: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 43–52. ACM, New York (2007)
2. Buneman, P.: A Note on the Metric Properties of Trees. J. Combinatorial Theory (B) 17, 48–50 (1974)
3. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. In: SIGCOMM '04: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 15–26. ACM, New York (2004)
4. Ramasubramanian, V., Malkhi, D., Kuhn, F., Balakrishnan, M., Gupta, A., Akella, A.: On the treeness of internet latency and bandwidth. In: SIGMETRICS '09: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, pp. 61–72. ACM, New York (2009)

Brief Announcement: Synchronous Las Vegas URMT Iff Asynchronous Monte Carlo URMT

Abhinav Mehta, Shashank Agrawal, and Kannan Srinathan

Center for Security, Theory and Algorithmic Research (C-STAR),
International Institute of Information Technology, Hyderabad, 500032, India.
{abhinav_mehta@research.,shashank.agrawal@research.,srinathan@}iiit.ac.in

Introduction: In the unconditionally reliable message transmission (URMT) problem, two non-faulty nodes, the sender **S** and the receiver **R** are part of a communication network modelled as a digraph over a set of nodes influenced by an unbounded active adversary that may corrupt some subset of these nodes. **S** has a message that he wishes to send to **R**; the challenge is to design a protocol such that **R** correctly obtains **S**'s message with arbitrarily high probability, irrespective of what the adversary (maliciously) does to disrupt the protocol. Analogous to randomized sequential algorithms, one may distinguish between two variants of URMT, namely, *Monte Carlo* and *Las Vegas*. In the former variant **R** outputs the sender's message with high probability and may produce an incorrect output with small probability; in the latter, **R** outputs the sender's message with high probability and with small probability may abort the protocol but in no case does the receiver terminates with an incorrect output.

In this work, we focus on studying the (im)possibility of *Monte Carlo* URMT protocols over asynchronous networks (U_{AMC}) and *Las Vegas* URMT protocols over synchronous networks (U_{SLV}). Though not seemingly related, interestingly, we show that the network connectivity requirements for both the aforementioned cases are same (and are strictly greater than that of *Monte Carlo* protocols over synchronous networks, which has been studied in [4]).

Model and Definitions: We model the underlying network by a directed graph, $\mathcal{N} = (V, \mathcal{E})$, where V is the set of nodes and $\mathcal{E} \subseteq V \times V$ is the set of directed edges in the network. We assume that the edges are *secure*, i.e., if $(u, v) \in \mathcal{E}$ then u can send any message to v securely and reliably. We further assume that the topology of the network is known to every node. Fault in the network is modelled by a non-threshold adversary structure $\mathbb{A}$ [1], which is a set of the subsets of the node set, i.e., $\mathbb{A} \subseteq \mathcal{P}(V \setminus \{\mathbf{S}, \mathbf{R}\})$, one of which may be Byzantinely corrupt during an execution. We assume that $\mathbb{A}$ is a maximal basis [1].

Let the message space be a large finite field $\langle \mathbb{F}, +, \cdot \rangle$. All the computations are done in this field. In the following, all probabilities are taken over the choice of the message **S** intends to send, the random inputs of all honest players and the random inputs of the adversary.

Definition $(\mathbb{A}, \delta)$-U_{AMC}: Let $\delta < \frac{1}{2}$. We say that a protocol in an asynchronous network $\mathcal{N}$ is $(\mathbb{A}, \delta)$-U_{AMC} if for all valid Byzantine corruptions of any

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 201–203, 2010.

$B \in \mathbb{A}$, the probability that $\mathbf{R}$ outputs $\mathbf{m}$ given that $\mathbf{S}$ has sent $\mathbf{m}$, is at least $(1-\delta)$. Otherwise $\mathbf{R}$ outputs $\mathbf{m}' \neq \mathbf{m}$ or does not terminate.

DEFINITION $(\mathbb{A}, \delta)$-U$_{SLV}$: Let $\delta < \frac{1}{2}$. We say that a protocol in a synchronous network $\mathcal{N}$ is $(\mathbb{A}, \delta)$-U$_{SLV}$ if for all valid Byzantine corruptions of any $B \in \mathbb{A}$, the probability that $\mathbf{R}$ outputs $\mathbf{m}$ given that $\mathbf{S}$ has sent $\mathbf{m}$, is at least $(1-\delta)$. Otherwise, $\mathbf{R}$ outputs a special symbol $\perp$ $(\notin \mathbb{F})$.

For $m, K_1, K_2 \in \mathbb{F}$, we define an authentication function $\chi(m; K_1, K_2) = (m, m \cdot K_1 + K_2)$. We refer to K_1, K_2 as *keys*. Given that the adversary does not know the keys, it may tamper with an authenticated message without being detected with atmost $1/|\mathbb{F}|$ probability [3].

Our Results: We have the following results, detailed proofs appear in [2].

THEOREM 1: In a directed network $\mathcal{N}$, $(\mathbb{A}, \delta)$-U$_{SLV}$ (resp. $(\mathbb{A}, \delta)$-U$_{AMC}$) protocol is possible if and only if for every adversary structure $B \subseteq \mathbb{A}$ such that $|B| = 2$, (B, δ)-U$_{SLV}$ (resp. (B, δ)-U$_{AMC}$) protocol is possible.

THEOREM 2: Let $B = \{B_1, B_2\}$. In a directed network $\mathcal{N}$, (B, δ)-U$_{SLV}$ (or (B, δ)-U$_{AMC}$ protocol) is possible if and only if for each $\alpha \in \{1, 2\}$, there exists a weak path q_α avoiding nodes in $B_1 \cup B_2$ such that every node u along the path q_α has a strong path to $\mathbf{R}$ avoiding all nodes in $B_{\overline{\alpha}}$ [1] . (Paths q_1, q_2 need not be distinct.)

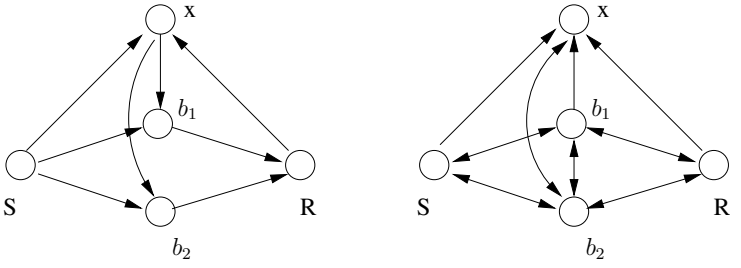


Fig. 1. (a) Network $\mathcal{N}_1$, (b) Network $\mathcal{N}_2$. Adversary structure is $B = \{\{b_1\}, \{b_2\}\}$

Proof Sketch: With the help of simple graphs, we give a sketch of our sufficiency and necessity proofs for Theorem 2. Consider the directed network $\mathcal{N}_1$ shown in Figure 1(a). Let m be the message $\mathbf{S}$ intends to send. We describe a $(B, 1/|\mathbb{F}|)$-U$_{SLV}$ protocol Π:

- Round 1: $\mathbf{S}$ sends m and $\mathbf{R}$ sends a random pair of keys K_1, K_2 to node x.
- Round 2: Node x sends $\chi(m; K_1, K_2)$ to b_1 and b_2.
- Round 3: Nodes b_1 and b_2 forward to $\mathbf{R}$ whatever is received from node x at the end of round 2.
- Round 4: For each $i \in \{1, 2\}$, let $\mathbf{R}$ receive (y_i, z_i) from b_i. $\mathbf{R}$ applies the following decision rule to recover the message: If $\exists\ j \in \{1, 2\}$ s.t. $z_j \neq y_j \cdot K_1 + K_2$, output $y_{\overline{j}}$ (since b_j is corrupt). Else if $y_1 \neq y_2$, output $\perp$ (this happens with probability atmost $1/|\mathbb{F}|$). Else, output y_1 (since in this case, $y_1 = y_2 = m$).

[1] We denote $\overline{1} = 2$ and vice-versa.

Along the lines of protocol Π, we can construct a $(B, 1/|\mathbb{F}|)$-U$_{AMC}$ protocol Π' for network $\mathcal{N}_1$. As the network is asynchronous, computation does not proceed in rounds, rather nodes wait for messages to arrive. However, obviously, $\mathbf{R}$ cannot wait for both (y_1, z_1) and (y_2, z_2) to arrive before taking some action. So, we modify $\mathbf{R}$'s decision rule as follows: Wait until for some $i \in \{1, 2\}$, (y_i, z_i) is received from b_i. If $z_i = y_i \cdot K_1 + K_2$, output y_i (probability that $y_i \neq m$ is atmost $1/|\mathbb{F}|$). Else, wait until $(y_{\overline{i}}, z_{\overline{i}})$ is received from $b_{\overline{i}}$ and output $y_{\overline{i}}$ (since b_i is corrupt, $\mathbf{R}$ is bound to receive $y_{\overline{i}}$, and $y_{\overline{i}} = m$).

We now see why the edge (x, b_1) is critical. Consider the network $\mathcal{N}_2$ shown in Figure 1(b) in which the edge (x, b_1) is missing but all the other edges present in $\mathcal{N}_1$ are also present in $\mathcal{N}_2$ (along with a few more edges). Here, we briefly describe adversary strategies under which no protocol for URMT can exist. We direct the reader to [2] for a formal proof of why the strategies succeed. We assume that the adversary knows the message $\mathbf{S}$ intends to send. Let $m_1, m_2 \in \mathbb{F}$ be two distinct messages. When $\mathbf{S}$ intends to send m_i, adversary corrupts node b_i.

Case of (B, δ)-U_{SLV}: When adversary corrupts b_1, it does not exchange any messages with $\mathbf{S}$, b_2 and x. Also, it simulates a local copy of $\mathbf{S}$ with input m_2. When adversary corrupts b_2, it does not exchange any messages with nodes $\mathbf{S}$, x and b_1. Also, it simulates a local copy of $\mathbf{S}$ with input m_1 and a local copy of x (say x_2). Furthermore, to ensure indistinguishability, adversary guesses the messages sent along the edge $(\mathbf{R}, x)$ and feeds it as input to x_2.

Case of (B, δ)-U_{AMC}: On corrupting b_2, adversary fail-stops it. For any valid protocol, when $\mathbf{S}$ chooses to send m_1, there must exist a finite time instant T before which $\mathbf{R}$ halts with output m_1 with at least $1 - \delta$ probability. When adversary corrupts b_1, it does not exchange any messages with nodes $\mathbf{S}$, x and b_2. Also, it simulates a local copy of $\mathbf{S}$ with input m_2. Furthermore, as the adversary has power to schedule messages, it delays all messages along the directed edge $(b_2, \mathbf{R})$ till time T.

References

1. Hirt, M., Maurer, U.: Player Simulation and General Adversary Structures in Perfect Multi-party Computation. Journal of Cryptology 13(1), 31–60 (2000)
2. Mehta, A., Agrawal, S., Srinathan, K.: Interplay between (im)perfectness, synchrony and connectivity: The Case of Probabilistic Reliable Communication. Cryptology ePrint Archive, Report 2010/392 (2010), http://eprint.iacr.org/
3. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: STOC '89: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, pp. 73–85. ACM, New York (1989)
4. Srinathan, K., Rangan, C.P.: Possibility and complexity of probabilistic reliable communications in directed networks. In: Proceedings of 25th ACM Symposium on Principles of Distributed Computing, PODC'06 (2006)

Foundations of Speculative Distributed Computing
(Invited Lecture Extended Abstract)

Rachid Guerraoui

EPFL, Switzerland

The Pitch

Speculation is frequent in distributed computing. It is even the norm in distributed algorithms that are designed for practical purposes. Yet, speculation is difficult and it has, so far, led to algorithms that are intricate and hard to reason about, let aside prove and test. The reason is simple: speculation involves different execution paths that are usually intermingled in the same algorithm, adding to the difficulties of concurrency and communication, inherent to general-purpose distributed computing.

This talk suggests the possibility of a principled approach to speculation and, indirectly, of a well-founded approach to the design and implementation of practical distributed algorithms.

Speculation in a Nutshell

Speculation is the idea that the design of an algorithm follows an *if-then-else* control structure, obeying the celebrated political motto: *hope for peace but plan for war*.

In short, under certain good conditions, i.e., the *speculation*, a specific path of the algorithm, i.e., the *speculative path*, is executed. When the conditions are not met, i.e. in the *bad case*, a *backup path* is executed instead. The rationale behind this approach is twofold: *(i)* the probability for the speculation to hold is high and, *(ii)* the complexity of the speculative is significantly lower than that of the backup path.

Examples of speculative algorithms are numerous. They include cache coherence protocols, multi-thread assignment techniques, Ethernet congestion control mechanisms, transactional memory systems, as well as mutual exclusion, lock-free, replication and agreement algorithms.

The Many Faces of Speculation

Speculative algorithms can be classified according to at least three categories.

One form of speculation, arguably the most common, is the *direct* one with exactly one speculative and one backup paths. This is for instance the case with *fast* mutual exclusion algorithms where, in the absence of contention, the number of steps needed to enter a critical section is significantly lower than the one needed in the presence of contention.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 204–205, 2010.

A more sophisticated form of speculation is the *nested* one where the backup path might itself be composed of a speculative path and another, nested, backup path. An example of such a form of speculation is replication algorithms that behave in an efficient manner, say with constant complexity if the system is synchronous and prone to failure, and finally into a backup path with potentially undefined complexity if the system is not synchronous. One could also consider different types of contention and devise mutual exclusion algorithms that degrades gracefully according to the type of contention encountered, e.g. total contention, interval contention, step contention, etc. Early deciding algorithms, where the complexity depends on the actual number of failures that occur in an execution can also be viewed as nested speculative algorithms.

An even more sophisticated form of speculation is the *recursive* one where the distributed algorithm might keep switching back and forth between the speculative and the backup paths. This is typically the case with eventually synchronous agreement algorithms, which keep looping over a speculative path that terminates whenever the system becomes synchronous. Interestingly, speculation is used here to ensure termination, i.e. to turn the complexity from infinite to finite.

Speculative Principles

Like any distributed algorithm, a speculative algorithm A is supposed to solve a problem P assuming a set of executions (or traces) M, commonly called a model. The very characteristic of a speculative algorithm is that, under M', a strict subset of M, the complexity of A is supposed to be lower than its complexity in M.

Part of the challenge underlying speculation is that, while A is supposed to perform efficiently under M', A is still expected to solve P under M. Hence the design of A requires to *detect* whether the speculation turns out to be true, and if it does not, to *safely* fall back to the alternative path.

Setting the grounds for a theory of speculation goes through addressing many challenging questions. Given a problem P and an algorithm A' solving P in a model M', is it possible to automatically transform A' into the speculative part of a more general algorithm A that solves P within a strict superset of M? Is there a refined speculation of P of which solutions can be viewed as speculative, composable, parts of the same speculative algorithm solving P. Are there, for any sub-model M' of a more general model M, a general *detector* that establishes whether a given execution path of M belongs M'.

Ideally, a theory of speculation would lead to devise the distributed counterpart of the *if-then-else* control structure where *(a)* the speculation, *(b)* the speculative algorithm and *(c)*, its backup, would be, first class, separate citizens of a distributed computation.

Anonymous Asynchronous Systems: The Case of Failure Detectors

François Bonnet and Michel Raynal

IRISA, Université de Rennes 1, France
fbonnet@irisa.fr, raynal@irisa.fr

Abstract. Trivially, agreement problems such as consensus, that cannot be solved in non-anonymous asynchronous systems prone to process failures, cannot be solved either if the system is anonymous. The paper investigates failure detectors that allow processes to circumvent this impossibility. It has several contributions. It first presents four failure detectors (denoted AP, $\overline{AP}$, $A\Omega$ and $A\Sigma$) and show that they are the "identity-free" counterparts of the two perfect failure detectors, eventual leader failure detectors and quorum failure detectors, respectively. $A\Sigma$ is new and showing that $A\Sigma$ and Σ have the same computability power in a non-anonymous system is not trivial. The paper also shows that the notion of failure detector reduction is related to the computation model. Then, the paper presents and proves correct an uniform anonymous consensus algorithm based on the failure detector pair $(A\Omega, A\Sigma)$ ("uniform" means that not only processes have no identity, but no process is aware of the total number of processes). This new algorithm is not a "straightforward extension" of an algorithm designed for non-anonymous systems. To benefit from $A\Sigma$, it uses a novel message exchange pattern where each phase of every round is made up of subrounds in which appropriate control information is exchanged. Finally, the paper discusses the notions of failure detector hierarchy, weakest failure detector for anonymous consensus, and the implementation of identity-free failure detectors in anonymous systems.

1 Introduction

Anonymous systems. One of the main issue faced by distributed computing lies in mastering uncertainty created by the adversaries that are asynchrony and failures. As a simple example, the net effect of these adversaries makes impossible for a process to know if another process has crashed or is only very slow. Recently, new facets of uncertainty (e.g., dynamicity, mobility) have appeared and made distributed computing even more challenging.

Among the many adversaries that distributed computing has to cope with, *anonymity* is particularly important. It occurs when the computing entities (processes, agents, sensors, etc.) have no name, and consequently cannot distinguish the ones from the others. It is worth noticing that, from a practical point of view, anonymity is a first class property as soon as one is interested in guaranteeing privacy.

One of the very first works (to our knowledge) that addressed anonymous systems is the work of D. Angluin [1]. In that paper, considering message passing systems,

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 206–220, 2010.

Angluin was mainly interested in computability issues, namely answering the question "which functions can be computed in presence of asynchrony and anonymity?"

Enriching a system with a failure detector. Failure detectors [8] are one of the most popular approaches to circumvent impossibility results in non-anonymous failure-prone asynchronous systems. Roughly speaking, a failure detector is a device that provides each process with failure-related information. According to the quality of this information, several types of failure detectors can be defined. As an example, let us consider the consensus problem. This problem, that cannot be solved in a pure asynchronous message-passing system prone to even a single process crash [13], is defined as follows. Each process proposes a value, and every process that does not crash has to decide a value (termination), such that a decided value is a proposed value (validity), and no two processes decide different values (agreement). It has been shown that the *eventual leader* failure detector denoted Ω is the weakest failure detector that allows consensus to be solved in message-passing asynchronous systems where a majority of processes never crash [7]. It has also been shown that the pair (Σ, Ω), where Σ is a *quorum* failure detector [10], is the weakest failure detector to solve consensus in non-anonymous systems when any number of processes may crash [10,11]. (These failure detectors are precisely defined later in the paper.)

Failure detectors and anonymous systems. The local output of a failure detector Ω is a process identity. Similarly, the local output of a failure detector P (perfect failure detector) [8] or Σ [10] is a set of process identities. While these failure detectors can be added to an anonymous distributed system, their outputs cannot be directly used by the anonymous processes for the simple reason that there is no "process identity" notion inside the system. This means that (the output of) Ω is useless in an anonymous system. As far as the output of P or Σ is concerned an anonymous process can only exploit the cardinality of the identity set that is currently output. As we will see, this cardinal value can be exploited if the failure detector is P, while it cannot if it is Σ.

Differently from Ω, P or Σ, failure detectors have been proposed that, while used in non-anonymous systems, neither output process identities nor associate values with process identities. We call them *identity-free* failure detectors. As an example, the failure detector $\mathcal{L}$, that outputs a Boolean value at every process has been introduced in [12], where it is shown to be the weakest failure detector for the $(n-1)$-set agreement problem in n-process asynchronous message-passing systems prone to any number of crashes. This failure detector has been generalized in [2].

A failure detector, denoted here AP, that outputs an approximation of the number of crashed processes has been proposed in [17,18]. This failure detector has been used in [3] to solve consensus in anonymous systems prone to any number $t < n$ of process crashes. It has also been shown in [3] that, in an anonymous system enriched with such a failure detector, $2t + 1$ is a lower bound on the number of rounds for consensus (in a non-anonymous system enriched with a perfect failure detector, this lower bound is $t + 1$).

Roadmap. The paper is made up of 6 sections. The anonymous distributed computation model is presented in Section 2. Section 3 presents the four identity-free failure detectors AP, $\overline{AP}$, $A\Omega$ and $A\Sigma$. Section 4 addresses failure detector reductions and shows

that $A\Sigma$ and Σ are equivalent in non-anonymous systems. This section also discusses the notion of failure detector hierarchy and the implementability of anonymous failure detectors. Section 5 presents a uniform $(A\Sigma, A\Omega)$ -based anonymous consensus algorithm. Finally, Section 6 discusses the "weakest failure detector" issue for anonymous consensus.

Note. Due to page limitation, the proofs of theorems appear in the corresponding technical report [6].

2 Anonymous Asynchronous Message-Passing Systems

Process model. The system is made up of a fixed number n of processes, denoted $p_1, \ldots, p_n$. $\Pi = \{1, \ldots, n\}$ denotes the set of process identities (also called indexes).

Processes are anonymous in the sense that no process knows the existence of indexes and all processes execute the same algorithm. This means that indexes are can only be used from an external observer point of view: they do not belong to the system as perceived by processes. The processes are asynchronous in the sense that there is no assumption on their respective speeds.

The underlying time model is the set of positive integers (denoted $\mathbb{N}$). Time instants are denoted τ, τ', etc. Similarly to indexes, this time notion is not accessible to the processes. It is only used from an external observer point of view to state or prove properties. More generally, but for process identities, the computation model is the same as in [8].

Failure model. A process executes correctly its algorithm until it possibly crashes. A crash is a premature stop; after it has crashed, a process executes no step. A process that does not crash in a run is *correct* in that run. Otherwise, it is *faulty* in that run. Until it crashes (if ever it does), a process is *alive*. Given a run, $Correct$ denotes the set of processes that are correct in that run.

An environment is a set of failure patterns, where a *failure pattern* [8] is a function $F : \mathbb{N} \to 2^{\Pi}$ such that $F(\tau)$ denotes the set of processes that have crashed by time τ. We consider here failure patterns in which all (but one) processes may crash in a run. This set of failure patterns is called *wait-free environment*.

Communication. The processes communicate by exchanging messages through reliable channels. These channels are asynchronous, which means that there is no assumption on message transit delays, except that they are positive and finite (every message eventually arrives).

The processes are provided with a broadcast() communication primitive that allows the invoking process to send the same message to all the processes (including itself). The broadcast() primitive is not reliable in the sense that, if a process p_i crashes while broadcasting a message, that message can be received by an arbitrary subset of processes. When it receives a message, a process cannot determine which process is its sender.

Notation. The previous computation model is denoted $\mathcal{AAS}[\emptyset]$. $\mathcal{AAS}$ is an acronym for Anonymous Asynchronous System; $\emptyset$ means that there is no additional assumption.

$\mathcal{AS}[\emptyset]$ is used to denote the non-anonymous counterpart of $\mathcal{AAS}[\emptyset]$, i.e., an *A*synchronous message-passing *S*ystem prone to any number of crash failures and where each process has a distinct identity and knows all process identities.

3 Failure Detectors

3.1 Definition of Failure Detectors

The following definitions, based on Π and the set $\mathbb{N}$ of time instants, are from [8]. A *failure detector history with range* R describes the behavior of a failure detector during a run. It is a function $H : \Pi \times \mathbb{N} \to R$ where $H(i, \tau)$ describes the value of the failure detector at p_i at time τ. A *failure detector* D *with range* R is a function that maps each failure pattern F to a set of failure detector histories with range R: $D(F)$ is the set of failure detector histories that D can exhibit when the failure pattern is F.

Let A and B be two failure detectors. In the following $\mathcal{AAS}[A, B]$ denotes the system $\mathcal{AAS}[\emptyset]$ enriched with failure detector A and failure detector B. This means that any process can additionally read the local variables provided by these failure detectors. $\mathcal{AAS}[A]$ denotes a system where only the failure detector A can be accessed.

3.2 A Few Classical Failure Detectors

This section briefly recall informally the definition of three well-known failure detectors, namely, P, Ω, and Σ (their formal definitions can be found in [7,8,10,11]) and also the defintion of $\overline{P}$ (a simple variant of P). While they have been designed for non-anonymous systems, nothing prevent us from enriching an anonymous system with any of them (but, as shown below, it is possible that such an "enrichment" does not add computational power to the anonymous system).

The perfect failure detector P. A *perfect* failure detector [8] provides each process p_i with a set denoted $suspected_i$ that contains process indexes and is such that it (1) never contains the index of a process before it crashes and (2) eventually contains the indexes of all faulty processes.

When considering $\mathcal{AAS}[P]$, it is easy to see that, while a process p_i cannot use the identities that are currently in $suspected_i$ (those are meaningless in an anonymous system), it can instead use the integer $|suspected_i|$ that provides it with a lower bound on the number of crashed processes.

The perfect failure detector $\overline{P}$. This failure detector provides each process p_i with a set denoted $alive_i$ that contains process indexes and is such that it (1) contains at least the indexes of the processes that are currently alive and (2) eventually contains only the indexes of correct processes. Intuitively, the output of $\overline{P}$ corresponds to the complement set of the output of P with respect to Π the set of indexes.

The eventual leader failure detector Ω. An *eventual leader* failure detector Ω [7] provides each process p_i with a variable $leader_i$ that contains a process index such that eventually (1) the variables $leader_i$ of the non-faulty processes contain forever the same index and (2) this index is the one of a non-faulty process.

As identities are outside an anonymous system, it is easy to see that $\mathcal{AAS}[\Omega]$ is not more powerful than $\mathcal{AAS}[\emptyset]$ (due to anonymity, no process can exploit the process identity it is provided by Ω).

The quorum failure detector Σ. The notion of quorum has been introduced in [14] (and explicitly used to solve consensus in [19]). A *quorum* failure detector Σ [10] provides each process with a set $sigma_i$ of process indexes (such a set is called quorum) such that (1) eventually, any quorum contains only correct processes and (2) any two quorum values do intersect (whatever the time instants at which these quorum values have been output).

It is shown in [10] that Σ is the weakest failure detector to implement a register in an asynchronous message-passing system prone to any number of crashes. A simple proof of this result appears in [5].

3.3 Identity-Free Failure Detectors

As seen in the Introduction, some failure detectors do not output process identities (or values associated with process identities) but Boolean values, integers, etc. whose "meaning" is on the entire system. As already indicated, we call them identity-free failure detectors. This section recalls the definition of two of them (AP and $A\Omega$) and introduces a new one ($A\Sigma$) that is the identity-free counterpart of Σ. As we will see later, "counterpart" means that while $A\Sigma$ is meaningful in an anonymous system, Σ is not, but they have the same computational power in a non-anonymous system.

The identity-free perfect failure detector AP. Such a failure detector (a variant of a failure detector introduced in [17,18]) provides each process p_i with an integer $ancp_i$ ($a$pproximate $n$umber of $c$rashed $p$rocesses) that (1) is never greater than the number of crashed processes and (2) is eventually equal to the number of faulty processes (see [3] for a formal definition). Intuitively AP satisfies the same properties as P except that, instead of returning a set of indexes, it simply returns the cardinal of this set.

The identity-free perfect failure detector $\overline{AP}$. Such a failure detector provides each process p_i with an integer $anap_i$ ($a$pproximate $n$umber of $a$live $p$rocesses) that (1) is never smaller than the number of alive processes and (2) is eventually equal to the number of correct processes. Intuitively $\overline{AP}$ satisfies the same properties as $\overline{P}$ except that, instead of returning a set of indexes, it simply returns the cardinal of this set.

The identity-free eventual leader failure detector $A\Omega$. Such a failure detector [15] provides each process p_i with a boolean a_leader_i such that eventually (1) there is one non-faulty process (say p_ℓ) whose Boolean variable remains forever true and (2) the Boolean variables of the other non-faulty processes remain forever false. Intuitively $A\Omega$ satisfies the same properties as Ω.

The identity-free quorum failure detectors $A\Sigma$. Such a failure detector provides each process with a variable a_sigma_i that contains pairs. Each pair is composed of a label x and an integer y. Without loss of generality, the set of labels is assumed to be a subset

of $\mathbb{N}$. The intuition is the following. If $(x, y) \in a\_sigma_i$, $A\Sigma$ has informed process p_i of (1) the existence of label x and (2) the fact that y processes are assumed to know the label x. As, we will see, a quorum is a set of processes that know the same label.

Formally, the behavior of the local variables $\{a\_sigma_i\}_{1 \leq i \leq n}$ is defined by the following properties. The first two properties (validity and monotonicity) are well-formedness properties, while the last two properties (safety an liveness) are behavioral properties.

Definition 1. $S(x) = \{i \mid \exists \tau \in \mathbb{N} : \ (x, -) \in a\_sigma_i^\tau\}$.

Formal definition. The formal definition of $A\Sigma$ is as follows.

- Validity. $\forall\, i \in \Pi$: $\forall\, \tau \in \mathbb{N}$: $a\_sigma_i^\tau = \{(x_1, y_1), \ldots, (x_p, y_p)\}$ where $\forall\, 1 \leq a, b \leq p : (x_a, y_b \in \mathbb{N}) \ \wedge \ \big((a \neq b) \Rightarrow (x_a \neq x_b)\big)$.
- Monotonicity. $\forall i \in \Pi$: $\forall \tau \in \mathbb{N}$: $\big(((x, y) \in a\_sigma_i^\tau) \Rightarrow \big(\forall \tau' \geq \tau : \exists y' \leq y : (x, y') \in a\_sigma_i^{\tau'}\big)$.
- Liveness. $\forall i \in Correct : \exists (x, y)$: $\exists \tau : \forall\, \tau' \geq \tau$: $\big((x, y) \in a\_sigma_i^{\tau'}\big) \wedge \big(|S(x) \cap Correct| \geq y\big)$.
- Safety. $\forall\, i_1, i_2 \in \Pi$: $\forall\, \tau_1, \tau_2 \in \mathbb{N}$: $\forall\, (x_1, y_1) \in a\_sigma_{i_1}^{\tau_1}$: $\forall\, (x_2, y_2) \in a\_sigma_{i_2}^{\tau_2}$: $\forall\, T_1 \subseteq S(x_1)$: $\forall\, T_2 \subseteq S(x_2)$: $\quad \big((|T_1| = y_1) \ \wedge \ (|T_2| = y_2)\big) \Rightarrow (T_1 \cap T_2 \neq \emptyset)$.

Interpretation. The validity property expresses the fact that, at any time, a_sigma_i is a non-empty set of pairs (x, y) where x is a label and y a number of processes associated with this label (those are processes assumed to know the label x). For any process p_i, at any time τ and any label x, x can appear at most once in $a\_sigma_i^\tau$, but any number of distinct labels can appear in $a\_sigma_i^\tau$.

The monotonicity property states that the number y of processes associated with a label x, as known by p_i, can only decrease. This requirement is not necessary but makes things simpler. Not considering this monotonicity property will not change our results but would make them more difficult to understand and proofs more technical. Hence, this property has to be seen as a "comfort" property, and not as a "computability" property.

$S(x)$ is the set of all processes that know the label x. While a process p_i knows it belongs to $S(x)$, it does not know the value of $S(x)$. Moreover $S(x)$ can not be used by algorithms; it is only used to define $A\Sigma$.

The next property is called liveness because it is used to prove liveness of $A\Sigma$-based algorithms (and similarly for the safety property). It captures the fact that, after some time, a quorum contains only correct processes, thereby preventing a correct process from blocking forever if it uses that quorum. To that end, this property states that, for any correct process p_i, there is eventually a label x such that its associated number y of processes remains always smaller or equal to the number of correct processes in $S(x)$. (The underlying intuition is that any correct process will eventually know a label that is associated with a set of correct processes only.)

The safety property is a little bit more involved. It captures the intersection property associated with quorums. Let x_1 and x_2 be two labels known by p_{i_1} and p_{i_2} respectively,

T_1 any subset of $S(x_1)$, T_2 any subset of $S(x_2)$ (let us remember that $S(x)$ is the set of all the processes that know label x). The safety property states the following: if $|T_1| = y_1$ and $|T_2| = y_2$, where $(x_1, y_1) \in a\_sigma_{i_1}$ and $(x_2, y_2) \in a\_sigma_{i_2}$), then $T_1 \cap T_2 \neq \emptyset$. Let us remember that y_1 is the number of processes associated with label x_1 as know by p_{i_1} (and similarly for y_2). The intuition is that the y_1 processes that know label x_1 and y_2 processes that know label x_2 do intersect.

4 Reductions between Failure Detectors

Definitions. The following definitions are a straightforward generalization of definitions given in [8]. They add the notion of "system model". Given two failure detectors $D1$ and $D2$, and a system model $\mathcal{C}$ ($\mathcal{AAS}$ or $\mathcal{AS}$), $D1$ is *weaker* than $D2$ in $\mathcal{C}$ (denoted $D1 \preceq_{\mathcal{C}} D2$) if there is an algorithm that emulates the output of the failure detector $D1$ in $\mathcal{C}[D2]$. If reductions exist in both direction, (i.e., $D1 \preceq_{\mathcal{C}} D2$ and $D2 \preceq_{\mathcal{C}} D1$), $D1$ and $D2$ are *equivalent* in $\mathcal{C}$ (denoted $D1 \simeq_{\mathcal{C}} D2$). Finally the notation $D1 \prec_{\mathcal{C}} D2$ means that $D1$ is *strictly weaker* than $D2$ (i.e., $D1 \preceq_{\mathcal{C}} D2$ and $D2 \not\preceq_{\mathcal{C}} D1$). Similarly, if $D1$ is (strictly) weaker than $D2$ in $\mathcal{C}$, $D2$ is said to be (strictly) *stronger* than $D1$ in $\mathcal{C}$. If $D1 \not\preceq_{\mathcal{C}} D2$ and $D2 \not\preceq_{\mathcal{C}} D1$, $D1$ and $D2$ are said to be not comparable in $\mathcal{C}$.

It is important to notice that the existence of reductions between failure detectors depends on the system model. Given any two failure detectors $D1$ and $D2$, as $\mathcal{AAS}$ is $\mathcal{AS}$ without the notion of process identities, we have $(D1 \preceq_{\mathcal{AAS}} D2) \Rightarrow (D1 \preceq_{\mathcal{AS}} D2)$, but we do not have $(D1 \preceq_{\mathcal{AS}} D2) \Rightarrow (D1 \preceq_{\mathcal{AAS}} D2)$.

As simple examples (see below) we have (1) $X \simeq_{\mathcal{AS}} AX$ for $X = P, \Sigma$ and Ω; (2) $AP \prec_{\mathcal{AAS}} P$; (3) $(A\Omega \preceq_{\mathcal{AS}} \Omega) \wedge (A\Omega \not\preceq_{\mathcal{AAS}} \Omega)$.

4.1 Simple Reductions

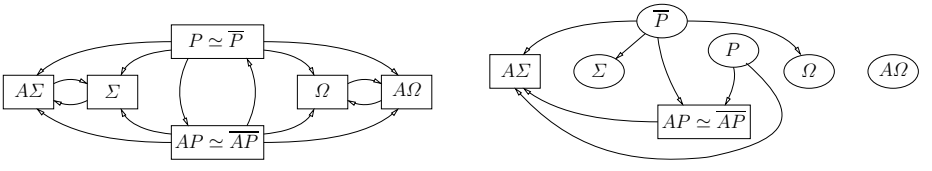
In $\mathcal{AS}$. Directly from definitions, one can easily see that $P \simeq_{\mathcal{AS}} \overline{P}$ and $AP \simeq_{\mathcal{AS}} \overline{AP}$: for the first equivalence it is sufficient to compute the complement set of the output set with respect to Π the set of all indexes, whereas for the latter it is sufficient to substract the output integer to n, the total number of processes.

Due to page limitation, the proofs that $P \simeq_{\mathcal{AS}} AP$ and $\Omega \simeq_{\mathcal{AS}} A\Omega$ are given in [6]. Moreover it is well-known that $\Omega \prec_{\mathcal{AS}} P$ [7] and $\Sigma \prec_{\mathcal{AS}} P$ [10]. The simple reduction $A\Sigma \preceq_{\mathcal{AS}} \Sigma$ appears in [6] while the next section gives an algorithm for $\Sigma \preceq_{\mathcal{AS}} A\Sigma$ (whose proof appears also in [6]). It follows that, in non-anonymous systems, the three classical failure detectors defined in Section 3.2 and their identity-free counterparts defined in Section 3.3 have the same computational power.

In $\mathcal{AAS}$. Some reductions between failure detectors that exist in a non-anonymous system are no longer possible in an anonymous system. More precisely we have the following in $\mathcal{AAS}$.

- P and $\overline{P}$ are not more equivalent. There is no reduction from one to the other since there is no way for processes, in the anonymous model, to discover indexes of alive (resp. faulty) processes when their failure detectors provide them only with indexes of faulty (resp. alive) processes.

- The equivalence between AP and $\overline{AP}$ still exist (reductions in the non-anonymous model do not use indexes and then remain valid in the anonymous model).
- P is no more stronger than Σ and Ω. Indeed there is no way for processes, in the anonymous model, to discover indexes of correct processes when their failure detectors provide them only with indexes of faulty processes. P is still stronger than AP and $A\Sigma$; the reductions of the non-anonymous model remain valid since there is no use of indexes.
- $\overline{P}$ is strictly stronger than Σ (take the output of $\overline{P}$), $\overline{AP}$ (take the cardinal of the output of $\overline{P}$), and Ω (take the smallest identity output by $\overline{P}$).
- P (resp. $\overline{P}$) and $A\Omega$ cannot be compared. Indeed, as the system is anonymous, there is no way for the processes to break asymmetry and elect a leader.
- AP is strictly stronger than $A\Sigma$. Assuming n is known, the reduction consists in permanently outputting the pair $(0, n - ancp_i)$ ($ancp_i$ is p_i's local output of AP). As $A\Sigma \prec_{\mathcal{AAS}} AP$, $AP \prec_{\mathcal{AAS}} P$, $AP \simeq_{\mathcal{AAS}} \overline{AP}$, $\overline{AP} \prec_{\mathcal{AAS}} \overline{P}$, we also have $A\Sigma \prec_{\mathcal{AAS}} P$ and $A\Sigma \prec_{\mathcal{AAS}} \overline{P}$.
- Due to the absence of identities, AP (resp. $\overline{AP}$) cannot be compared with Σ and Ω. Moreover, AP cannot be compared with $A\Omega$ (Proof in [6]).
- Ω and $A\Omega$ cannot be compared. On the one side, there exists no algorithm that can build Ω in $\mathcal{AAS}[A\Omega]$ since it is not possible to associate identities with processes. On the other side, there exists no algorithm that builds $A\Omega$ in $\mathcal{AAS}[\Omega]$ since the processes being anonymous, none of them can discover it is the eventual leader. Σ and $A\Sigma$ cannot be compared for the same reasons.



(a) in $\mathcal{AS}$ (b) in $\mathcal{AAS}$

Fig. 1. Hierarchy of failure detectors

Hierarchy of failure detectors. Figure 1 summarizes the reductions that exist between the previous failure detectors in non-anonymous systems (left) and anonymous systems (right). An arrow from $D1$ to $D2$ means that $D2$ is weaker than $D1$ in the corresponding system model.

Implementability. Failure detectors are introduced to capture the additional power required to solve a problem that is otherwise unsolvable in the considered system. While a (non-trivial) failure detector cannot be implemented in a pure asynchronous, it is interesting to investigate if it can be implemented in a synchronous system. When such an implementation does exist, the failure detector is *realistic* [9].

Considering Figure 1, a square indicates that the associated failure detector can be implemented in the corresponding synchronous system, while an ellipses denotes it cannot. As P can be easily implemented in a non-anonymous synchronous system, by reduction all the proposed failure detectors are realistic in $\mathcal{AS}$. As far anonymous synchronous systems are concerned we have the following.

- As there is no notion of process identity, P, $\overline{P}$, Σ and Ω cannot be implemented in an anonymous synchronous system.
- AP, $\overline{AP}$ and $A\Sigma$ can be implemented in an anonymous synchronous system. At every round r, any alive process broadcasts a heartbeat message and counts the number h_r of heartbeats received during that round. This number defines the current output of $\overline{AP}$, the integer $n - h_r$ defines the current output of AP and the pair $(0, h_r)$ defines the current output of $A\Sigma$.
- $A\Omega$ cannot be implemented in an anonymous synchronous system. Indeed even if the system is synchronous there is no deterministic solution for processes to break the symmetry between them.

The important point is that $A\Omega$ is not a realistic failure detector as far as the system is anonymous. Despite this fact, $A\Omega$ remains important from a theoretical point of view as it is relevant in the search for the "weakest failure detector" for anonymous consensus (see Section 6).

4.2 Building Σ in $\mathcal{AS}[A\Sigma]$

The construction. The algorithm that builds Σ in $\mathcal{AS}[A\Sigma]$ is described in Figure 2. It relies on two main data structures at each process p_i.

- $alive_i$ is a queue, always containing all the process indexes, that is managed as follows. When p_i receives a message from p_j, it reorders j and places it at the head of that queue. In that way, the processes that are alive (i.e., those that send messages) appear at the head of $alive_i$, while the processes that have crashed are progressively moved at its tail.
- $queue_i$ is an array of queues; $queue_i[x]$ contains the identities of the processes that, from p_i's point of view, know the quorum whose name is x. The quorum names x are obtained from the local output a_sigma_i (underlying failure detector $A\Sigma$). $queue_i[x][j]$ denotes the jth element of $queue_i[x]$.

According to these data structures, the behavior of a process p_i is made up of three tasks $T1$, $T2$ and $T3$.

- $T1$ is an infinite loop in which p_i repeatedly broadcasts a message ALIVE$(i, labels_i)$ that contains the names of the quorums it knows (i.e., those that appear in a_sigma_i).
- $T2$ is the matching task of $T1$. When p_i receives ALIVE$(j, labels)$, it first updates $alive_i$ accordingly (line 6). Then, for each quorum name it knows (line 7), updates its current view of the processes that know x (i.e., the processes p_j that have $(x, -)$ in their a_sigma_j, lines 8-9)).
- $T3$ is the core of the construction. It is an infinite loop whose aim is to define the current value of $sigma_i$ (the local output of Σ). Process p_i first computes a set $candidates$ that contains the pairs $(x, y) \in a\_sigma_i$ such that $|queue_i[x]| \geq y$ (line 12). Those are the pairs such that p_i has received ALIVE$(-, \{\cdots, x, \cdots\})$ from at least y distinct processes (i.e., y processes know the label x). If the set $candidates$ is empty, p_i cannot compute a non-trivial value for $sigma_i$. It consequently sets $sigma_i$ to Π (line 14). Otherwise, p_i computes a non-trivial value for

$sigma_i$ from the set $candidates$ (lines 15-18). To that end, $\mathsf{rank}(\ell)$ is defined as the position of the identity ℓ in the queue $alive_i$ (line 16).
The aim is to assign to $sigma_i$ the y processes that are at the head of $queue_i[x]$ (line 18), where the corresponding pair $(x, y) \in candidates$ is determined as follows. Using an array-like notation, the identities in the prefix $queue_i[x][1..y]$ "globally appear in $alive_i$ before" the identities in the other prefixes $queue_i[x'][1..y']$. "Globally appear before" means that there is an identity in $queue_i[x'][1..y']$ whose rank in $alive_i$ is after the rank of any identity in $queue_i[x][1..y]$. (This is formally expressed by lines 15-17.) Let us notice that several prefixes $queue_i[x][1..y]$ can globally appear as being the "first" in $alive_i$. If it is the case, any of them can be selected.
To fix the idea, consider the following example. $alive_i = [7, 1, 3, 9, 4, 8, 2, 5, 6]$, $a\_sigma_i = \{(5, 4), (7, 3), (2, 5)\}$, $queue_i[5] = [1, 3, 4, 2, 5]$, $queue_i[7] = [1, 8, 5]$, $queue_i[2] = [1, 5]$. Considering only $queue_i[5]$, $queue_i[7]$ and $queue_i[2]$, we have $candidates = \{(5, 4), (7, 3)\}$. As $queue_i[5][4] = 2$, and $queue_i[7][3] = 5$, we have $r\_min = \mathsf{rank}(queue_i[5][4]) = \mathsf{rank}(2) = 7 < \mathsf{rank}(queue_i[7][3]) = \mathsf{rank}(5) = 8$. Hence, $(x, y) = (5, 4)$ defines the queue prefix whose identities are "first" in $alive_i$. Consequently $sigma_i$ is set to $queue_i[5][1..4] = \{1, 3, 4, 2\}$.

Init: $alive_i \leftarrow$ all process indexes in arbitrary order;
for each x **do** $queue_i[x] \leftarrow$ empty queue **end for**.

(1) $T1$: **repeat forever**
(2) $labels_i \leftarrow \{x \mid (x, -) \in a\_sigma_i\}$;
(3) broadcast ALIVE$(i, labels_i)$
(4) **end repeat**.

(5) $T2$: **when** ALIVE$(j, labels)$ **is received**:
(6) remove j from $alive_i$; enqueue j at the head of $alive_i$;
(7) **for each** $x \in labels$ **such that** $\big((x, -) \in a\_sigma_i\big)$ **do**
(8) **if** $(j \in queue_i[x])$ **then** remove j from $queue_i[x]$ **end if**;
(9) enqueue j at the head of $queue_i[x]$
(10) **end for**.

(11) $T3$: **repeat forever**
(12) **let** $candidates = \{\ (x, y) \mid (x, y) \in a\_sigma_i \ \wedge\ |queue_i[x]| \geq y\ \}$;
(13) **if** $(candidates = \emptyset)$
(14) **then** $sigma_i \leftarrow \{1, \ldots, n\}$
(15) **else let** $r\_min = \min_{(x,y)\in candidates}\big(\mathsf{rank}(queue_i[x][y])\big)$
(16) **where** $\mathsf{rank}(\ell)$ = rank of ℓ in the queue $alive_i$;
(17) **let** $(x, y) \in candidates$ **such that** $\mathsf{rank}(queue_i[x][y]) = r\_min$;
(18) $sigma_i \leftarrow$ the first y elements of $queue_i[x]$
(19) **end if**
(20) **end repeat**.

Fig. 2. Building Σ in $\mathcal{AS}[A\Sigma]$ (code for p_i)

Theorem 1. *The algorithm described in Figure 2 builds Σ in $\mathcal{AS}[A\Sigma]$.*

5 Consensus Algorithm in $\mathcal{AAS}[A\Sigma, A\Omega]$

Why $(A\Sigma, A\Omega)$?. As indicated in the Introduction, the fact that (1) (Σ, Ω) is the weakest failure detector to solve consensus in a non-anonymous system, (2) (Σ, Ω) are useless in anonymous systems, and (3) $(A\Sigma, A\Omega)$ is the identity-free counterpart of (Σ, Ω) was one of our motivations for designing an $(A\Sigma, A\Omega)$-based uniform anonymous consensus algorithm.

Description of the uniform $(A\Sigma, A\Omega)$-based algorithm. This algorithm borrows its "three-phase per round" structure from the non-anonymous consensus algorithm presented in [20]. Differently from that algorithm, the message exchange pattern used inside the second and third phases is based on an entirely new principle. Moreover, due to the novelty of $A\Sigma$, the way the safety and liveness properties are ensured are also new.

The algorithm is described in Figure 3. It is round-based: a process executes a sequence of asynchronous rounds until it decides. A process p_i invokes the operation propose(v_i) (where v_i is the value it proposes), and decides when it executes the statement return(v) (line 25 or 38, where v is the value it decides). As in other non-anonymous consensus algorithms, when a process decides it stops participating in the consensus algorithm. Consequently, before deciding a process p_i broadcasts a message DECIDE(v) in order to prevent the other processes from blocking forever (waiting for a message that p_i will never send).

The three main local variables associated with a round are r_i (the local round number), and a pair of estimates of the decision value $est1_i$ and $est2_i$. The variable $est1_i$ contains p_i's current estimate of the decision value when a new round starts while $est2_i$, whose value is computed during the second phase of every round, contains either a new estimate of the decision value or a default value $\perp$.

The behavior of a process p_i during a round r is made up of three phases, denoted phase 1, 2 and 3 which are as follows. The first phase of a round is the only one where $A\Omega$ is used, while $A\Sigma$ is used only in the second and third phases of a round.

- In the first phase of a round, a process p_i that considers it is leader broadcasts a message PHASE1(r_i, v). If a_leader_i is false, p_i waits for a message PHASE1(r, v), adopts v as its new estimate and forwards PHASE1(r_i, v) to all (to prevent other processes from blocking forever in that phase of round r).
- Similarly to [19], the aim of the second phase of a round r is to assign a value to the variables $est2_i$ in such a way that the following round property denoted $P(r)$ is always satisfied (where $est2_i[r]$ denotes the value assigned to $est2_i$ at line 12 of round r):

$$P(r) \equiv \big[(est2_i[r] \neq \perp) \wedge (est2_j[r] \neq \perp)\big] \Rightarrow \big(est2_i[r] = est2_j[r]\big).$$

To attain this goal, a classical non-anonymous algorithm directs a process to wait for messages from processes defining a quorum [19]. In an anonymous system, this is no longer possible as the notion of process name is outside $A\Sigma$. A process p_i

```
operation propose (v_i):
(1)  est1_i ← v_i; r_i ← 0;
(2)  while true do
(3)    begin asynchronous round
(4)    r_i ← r_i + 1;
       % Phase 1: assign a value to est1_i with the help of AΩ %
(5)    wait until ((a_leader_i) ∨ (PHASE1(r_i, v) received));
(6)    if (PHASE1(r_i, v) received) then est1_i ← v end if;
(7)    broadcast PHASE1(r_i, est1_i);
       % Phase 2: assign a value v or ⊥ to est2_i %
(8)    sr_i ← 1; labels_i ← {x | (x, −) ∈ a_sigma_i}; broadcast PHASE2(r_i, sr_i, labels_i, est1_i);
(9)    repeat
(10)     if (PHASE3(r_i, −, −, est2) received) then est2_i ← est2; exit repeat loop end if;
(11)     if (∃ (x, y) ∈ a_sigma_i ∧ ∃ sr ∈ ℕ
                such that y msgs PHASE2(r_i, sr, labels_j, −) received with x ∈ labels_j for each msg)
(12)       then if (all y msgs contain the same estimate v) then est2_i ← v else est2_i ← ⊥ end if;
(13)            exit repeat loop
(14)       else if (labels_i ≠ {x | (x, −) ∈ a_sigma_i}) ∨ (PHASE2(r_i, sr, −, −) received with sr > sr_i)
(15)            then sr_i ← sr_i + 1; labels_i ← {x | (x, −) ∈ a_sigma_i};
(16)                 broadcast PHASE2(r_i, sr_i, labels_i, est1_i)
(17)            end if
(18)     end if
(19)   end repeat;
       % Phase 3: try to decide a value from the est2 values %
(20)   sr_i ← 1; labels_i ← {x | (x, −) ∈ a_sigma_i}; broadcast PHASE3(r_i, sr_i, labels_i, est2_i);
(21)   repeat
(22)     if (PHASE1(r_i + 1, −) received) then exit repeat loop end if;
(23)     if (∃ (x, y) ∈ a_sigma_i ∧ ∃ sr ∈ ℕ
                such that y msgs PHASE3(r_i, sr, labels_j, −) received with x ∈ labels_j for each msg)
(24)       then let rec_i = the set of estimates est2 contained in the y previous messages;
(25)            case (rec_i = {v})     then broadcast DECIDE(v); return(v)
(26)                 (rec_i = {v, ⊥}) then est1_i ← v
(27)                 (rec_i = {⊥})    then skip
(28)            end case;
(29)            exit repeat loop
(30)       else if (labels_i ≠ { x | (x, −) ∈ a_sigma_i }) ∨ (PHASE3(r_i, sr, −, −) received with sr > sr_i)
(31)            then sr_i ← sr_i + 1; labels_i ← {x | (x, −) ∈ a_sigma_i};
(32)                 broadcast PHASE3(r_i, sr_i, labels_i, est2_i)
(33)            end if
(34)     end if
(35)   end repeat
(36)  end asynchronous round
(37) end while.

(38) when DECIDE(v) is received: broadcast DECIDE(v); return(v).
```

Fig. 3. A Consensus algorithm for $\mathcal{AAS}[A\Sigma, A\Omega]$ (code for p_i)

can use only the pairs $(x, y) \in a\_sigma_i$, which supply no "immediately usable" information on which processes have sent messages. This issue is solved as follows. During each round, the messages broadcast by processes carry appropriate label-based information, and processes can be required to re-broadcast messages related to the very same round when this information does change.

Hence, a process p_i first broadcasts a message PHASE2$(r_i, sr_i, labels_i, est1_i)$ where sr_i is a sub-round number (initialized to 1), and $labels_i$ is the set of labels it knows (line 8). As we are about to see, this information (the pair sr_i and $labels_i$) allows p_i to wait for message from an appropriate quorum of processes.

Process p_i then enters a waiting loop (lines 9-19), that (as we will see in the proof) it eventually exits at line 10 or 13 after having assigned a value to $est2_i$. The exit

at line 10 is to prevent p_i from blocking forever in phase 2 when processes have already progressed to phase 3 of the current round r_i.
As far the exit of the repeat loop at line 13 is concerned, p_i exits when it has received "enough" (namely y) messages PHASE2$(r_i, sr, labels_j, -)$ (these messages carry a round number equal to r_i and the same sub-round number sr -which can be different from sr_i-) such that (a) $\exists (x, y) \in a\_sigma_i$ and (b) $x \in labels_j$ for every of the y received messages (line 11). When this occurs, if the y messages carry the same estimate value v, p_i assigns that value v to $est2_i$, otherwise, it assigns it the default value $\perp$ (line 12). In both cases, p_i exits the loop and starts the third phase.
If the test of line line 11 is not satisfied, p_i checks (line 14) if it has new information from its failure detector (predicate $labels_i \neq \{x \mid (x, -) \in a\_sigma_i\}$), or has received a message PHASE2$(r_i, sr, -, -)$ such that $sr > sr_i$ (which means that this message refers to a sub-round of r_i more advanced than sr_i). If this test is satisfied, while remaining at the same round, p_i increase sr_i and broadcasts the message PHASE2$(r_i, sr_i, labels_i, est1_i)$ which refreshes the values of sr_i and $labels_i$ it had sent previously. Otherwise, p_i continues waiting for messages.

– The aim of the third phase is to allow a process to decide when it discovers that a quorum of processes have the same non-$\perp$ value v in their estimates $est2_i$.
The message exchange pattern used in this phase (where the notion of sub-round is used) is exactly the same as in the one used in the second phase where the value of $est2_i$ replaces the value of $est1_i$.
The only thing that changes with respect to the second phase is the processing done when the predicate of line 23 is satisfied (let us notice that this predicate is the same as the one of line 11 when the message tag PHASE2 is replaced by the tag PHASE3). If p_i has received the same value v from all the processes that compose the last quorum defined from the predicate of line 23, it decides v (line 25). If it has received a value v and $\perp$, it adopts v as its new estimate $est1_i$ (line 26). Finally, if it has received only $\perp$, it keeps its previous estimate $est1_i$ (line 27). The proof shows that the combination of property $P()$ (established by the second phase), and the quorums defined by the predicates of lines 11 and 23, ensures that no two processes can decide differently.

Theorem 2. *The algorithm described in Figure 3 solves the consensus problem in* $\mathcal{AAS}[A\Sigma, A\Omega]$.

6 On the Weakest FD for Anonymous Consensus

Failure detector-based consensus algorithms. The previous section has presented an $(A\Sigma, A\Omega)$-based anonymous consensus algorithm. A (non-uniform) AP-based consensus algorithm has been presented in [3]. A natural question is then: "which of $(A\Sigma, A\Omega)$ and AP is the weakest to solve consensus in anonymous systems ?" Unfortunately $(A\Sigma, A\Omega)$ and AP cannot be compared in $\mathcal{AAS}$ (proof similar to the ones that appear in Section 4.1).

Notion of weakest failure detector for a given problem. [7] Given a problem $\mathcal{P}$ and a failure detector D, D is the *weakest failure detector* for $\mathcal{P}$ in $\mathcal{AX}[\emptyset]$ (where $\mathcal{AX}$ stands for $\mathcal{AS}$ or $\mathcal{AAS}$) if (a) there is an algorithm that solves $\mathcal{P}$ in $\mathcal{AX}[D]$, and (b) for any failure detector D' such that $\mathcal{P}$ can be solved in $\mathcal{AX}[D']$, we have $D \preceq D'$. It is shown in [16] that, in $\mathcal{AS}[\emptyset]$, any problem has a weakest failure detector.

New failure detectors. Given two failure detectors $D1$ and $D2$, let us define a new failure detector $D1 \oplus D2$ as follows. During an arbitrary but finite period of time, $D1 \oplus D2$ outputs $\perp$ at every process, and then behaves either as $D1$ or as $D2$ at all processes.

Let us observe that, if $D1$ and $D2$ cannot be compared, $D1$ (resp.,$D2$) is strictly stronger that $D1 \oplus D2$. This is because $D1 \oplus D2$ can trivially be built in $\mathcal{AX}[D1]$ (resp., $\mathcal{AX}[D2]$), while $D1$ (resp.,$D2$) cannot be built in $\mathcal{AX}[D1 \oplus D2]$.

Weakest failure detector for consensus in $\mathcal{AAS}$. Whereas (Σ, Ω) is the weakest failure detector for consensus in non-anonymous systems, $(A\Sigma, A\Omega)$ is not the weakest failure detector for anonymous consensus, due to the absence of reduction between AP and $A\Omega$. Hence, let us introduce the new failure detector $(A\Sigma, A\Omega) \oplus AP$ which is strictly weaker than both $(A\Sigma, A\Omega)$ and AP. Interestingly, there is a simple algorithm that solves anonymous consensus in $\mathcal{AAS}[(A\Sigma, A\Omega) \oplus AP]$. This algorithm is as follows. Each process p_i waits until the output of $\mathcal{AAS}[(A\Sigma, A\Omega) \oplus AP]$ is different from $\perp$. Then, according to the actual output of the failure detector (that is non-deterministic), it executes either the $(A\Sigma, A\Omega)$-based algorithm presented in Section 5 or the AP-based algorithm described in [3].

A conjecture. We conjecture that $(A\Sigma, A\Omega) \oplus AP$ is the weakest failure detector for solving anonymous consensus. This conjecture is motivated by the observation that in a non-anonymous system we have $\big((\Sigma, \Omega) \oplus P\big) \simeq_{\mathcal{AS}} (\Sigma, \Omega)$ where (Σ, Ω) is the weakest failure detector for consensus. If correct, this conjecture implies that $(A\Sigma, A\Omega) \oplus AP$ is *the* weakest failure detector for consensus since it will then be true in both anonymous and non-anonymous systems.

References

1. Angluin, D.: Local and Global Properties in Networks of Processes. In: Proc. 12th Symposium on Theory of Computing (STOC'80), pp. 82–93. ACM Press, New York (1980)
2. Biely, M., Robinson, P., Schmid, U.: Weak Synchrony Models and Failure Detectors for Message-passing (k)Set Agreement. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 285–299. Springer, Heidelberg (2009)
3. Bonnet, F., Raynal, M.: The Price of Anonymity: Optimal Consensus despite Asynchrony, Crash and Anonymity. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 341–355. Springer, Heidelberg (2009)
4. Bonnet, F., Raynal, M.: Looking for the Weakest Failure Detector for k-Set Agreement in Message-passing Systems: Is Π_k the End of the Road? In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 149–164. Springer, Heidelberg (2009)

5. Bonnet, F., Raynal, M.: A Simple Proof of the Necessity of the Failure Detector Σ to Implement an Atomic Register in Asynchronous Message-passing Systems. Information Processing Letters 110(4), 153–157 (2010)
6. Bonnet, F., Raynal, M.: Anonymous Asynchronous Systems: The Case of Failure Detectors. Tech. Report PI 1945, IRISA, Rennes (January 2010)
7. Chandra, T., Hadzilacos, V., Toueg, S.: The Weakest Failure Detector for Solving Consensus. Journal of the ACM 43(4), 685–722 (1996)
8. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: A Realistic Look at Failure Detectors. In: Proc. Int'l Conference International on Dependable Systems and Networks (DSN'02), pp. 345–353. IEEE Computer Press, Los Alamitos (2002)
10. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R.: Tight Failure Detectors Bounds on Atomic Objects. Journal of the ACM 57(4) (2010)
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: Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04), pp. 338–346. ACM Press, New York (2004)
12. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Tielmann, A.: The Weakest Failure Detector for Message Passing Set-Agreement. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 109–120. Springer, Heidelberg (2008)
13. 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)
14. Gifford, D.K.: Weighted Voting for Replicated Data. In: Proc. 7th ACM Symposium on Operating System Principles (SOSP'79), pp. 150–172. ACM Press, New York (1979)
15. Guerraoui, R., Raynal, M.: The Alpha of Indulgent Consensus. The Computer Journal 50(1), 53–67 (2007)
16. Jayanti, P., Toueg, S.: Every Problem has a Weakest Failure Detector. In: Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08), pp. 75–84 (2008)
17. Mostefaoui, A., Rajsbaum, S., Raynal, M., Travers, C.: On the Computability Power and the Robustness of Set Agreement-oriented Failure Detector Classes. Distributed Computing 21(3), 201–222 (2008)
18. Mostefaoui, A., Rajsbaum, S., Raynal, M., Travers, C.: The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. SIAM Journal of Computing 38(4), 1974–1601 (2008)
19. Mostefaoui, A., Raynal, M.: Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 49–63. Springer, Heidelberg (1999)
20. Mostefaoui, A., Raynal, M.: Leader-Based Consensus. PPL 11(1), 95–107 (2001)

The Computational Structure of Progress Conditions

Gadi Taubenfeld

The Interdisciplinary Center, P.O. Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il
http://www.faculty.idc.ac.il/gadi/

Abstract. Understanding the effect of different progress conditions on the computability of distributed systems is an important and exciting research direction. For a system with n processes, we define exponentially many new progress conditions and explore their properties and strength. We cover all the known, symmetric and asymmetric, progress conditions and many new interesting conditions. Together with our technical results, the new definitions provide a deeper understanding of synchronization and concurrency.

Keywords: Progress conditions, wait-freedom, obstruction-freedom, S-freedom, consensus, synchronization, contention, cooperation, universality, hierarchy.

1 Introduction

We define exponentially many new progress conditions and explore their properties and relative strength. Our results regarding the computational structure of the new and known, symmetric and asymmetric, progress conditions provide a deeper understanding of synchronization and concurrency. Most of the known progress conditions can be classified as either cooperation-based conditions or contention-based conditions. Cooperation arises when several processes need to coordinate their actions in order to achieve a common goal. Contention arises when several processes compete for exclusive use of shared resources, such as communication bandwidth, data items or files.

Fault-freedom, the weakest cooperation-based condition, guarantees that every process will complete its pending operations provided that all the processes participate and there are no failures. *Obstruction-freedom*, the weakest contention-based condition, guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" (i.e., do not take any steps) long enough [11]. *Wait-freedom*, the strongest both contention-based and cooperation-based progress condition, guarantees that every process will always be able to complete its pending operations in a finite number of its own steps, regardless of the behavior of the other processes [10]. While a consensus object can be implemented using only atomic registers under either fault-freedom or obstruction-freedom, it can not be implemented using registers under wait-freedom.

We start by proving two general impossibility results for symmetric progress conditions, which have many interesting implications. For example, we show that objects that satisfy cooperation-based progress conditions can be implemented from objects that satisfy the corresponding contention-based conditions, but not vice versa. We establish a formal connection between symmetric and asymmetric progress conditions,

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 221–235, 2010.

which enables us to apply the general impossibility results for proving new results also for asymmetric progress conditions. For the special case where only atomic registers are used, we give a complete characterization under which symmetric progress conditions consensus is solvable, and prove impossibility results for the asymmetric case. Finally, we prove a general universality result.

1.1 Exponentially Many Symmetric Progress Conditions

From now on we assume that the number of processes is n and $n \geq 2$. A process is *active* when it has pending operations, otherwise it is *passive*. For a set of processes P, let $|P|$ denotes the size of P. For a given point in a computation, $active.P$ is the *number* of active processes in P. We use S to denote a non-empty set such that $S \subseteq \{1, ..., n\}$.

> **Definition.** For any non-empty set S, the progress condition *S-freedom* guarantees that for every set of processes P, if at some point in a computation $active.P = |P|$ and $|P| \in S$, then every process in P will be able to eventually complete its pending operations, provided that (1) all the processes not in P do not take steps for long enough; and (2) none of the processes in P fails (which means that each of the processes in P will continue to take steps until it becomes passive).

Let A be an algorithm for n processes that satisfies S-freedom for some set S. Furthermore, assume that for some number k, $k \in S$ and $k - 1 \notin S$. Assume that for a set of processes P at some point in a computation of A, $active.P = |P| = k$, and that all the processes not in P do not take any more steps. Since not all the processes in P become passive at the same time eventually some process in P will become passive and once this happens $active.P = k - 1$. It is important to notice, that although at this point $active.P = k-1 \notin S$, the definition of *S-freedom* guarantees that (because in the near past all k processes were active) eventually every process in P will become passive.

It is possible to weaken the requirement that "*every* process in P will be able to eventually complete its pending operations", and only require that "*some* process in P will be able to eventually complete its pending operations". For one-shot objects (also called tasks), like consensus, most of our results apply also in this case.

Since the number of non-empty subsets of $\{1, ..., n\}$ is $2^n - 1$, there are $2^n - 1$ different progress conditions. They relate to known progress conditions as follows: The condition $\{n\}$-freedom is the same as fault-freedom; $\{1\}$-freedom is the same as obstruction-freedom; $\{1, ..., n\}$-freedom is the same as wait-freedom; $\{1, n\}$-freedom is the progress condition which implies both obstruction-freedom and fault-freedom. For $1 \leq k \leq n$, $\{1, ..., k\}$-freedom is the same as k-obstruction-freedom. We call these n conditions, *contention-based* progress conditions, since $\{1, ..., k\}$-freedom guarantees progress under contention of at most k processes. For $0 \leq t \leq n-1$, $\{n-t, ..., n\}$-freedom is the same as t-resiliency. We call these n conditions, *cooperation-based* progress conditions, since $\{n-t, ..., n\}$-freedom captures the ability to tolerate t faults.

Clearly, an object that satisfies T-freedom satisfies also S-freedom, for any $S \subset T$. For any given set S, we say that S-freedom is a *symmetric* progress conditions in the sense that a given process is not favored with respect to the others.

1.2 Asymmetric Progress Conditions

The notion of *asymmetric* progress conditions was coined and investigated in [13]. It is motivated by the observation that some processes may be more important than others and hence should get stronger progress guarantees. Thus, an asymmetric progress condition specifics the progress guarantees for each process separately. One such progress condition which is defined in [13], called (n, x)-liveness, satisfies wait-freedom for x processes and satisfies obstruction-freedom for the remaining $n - x$ processes.

In the literature, saying that an *object* is wait-free is the same as saying that each one of the *processes* is wait-free w.r.t. that object. Although using the term wait-freedom in two different ways may be confusing, it simplifies the discussion. Following this "tradition", we will say that an object is S-free iff each process is S-free w.r.t. that object. Clearly, a process that is T-free is also S-free, for any $S \subset T$. Asymmetric progress conditions can be practically motivated by modern multicore architectures where processes in different cores might be provided with different progress guarantees.

1.3 Our Contributions

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 process that invokes it. (2) *Validity*: In any run, if the consensus value is v, then some process invoked $o.propose(v)$. When $v \in \{0, 1\}$ the object is called a binary consensus object. By a consensus object we mean a binary consensus object; and by *n-consensus* we mean a multi-valued consensus object where $v \in \{0, 1, ..., n - 1\}$. The term *register* means an atomic read/write register. A simple (but not obvious) observation is that, for two positive integers m and n, and a set $S \subseteq \{1, ..., \min(m, n)\}$, it is *not* always possible to implement an S-free consensus object for n processes using S-free consensus objects for m processes and registers. Our results are:

New Definitions. We define exponentially many progress conditions and investigate their properties and relative strength. Together with the technical results, the new notions provide a deeper understanding of synchronization and concurrency.

General impossibility results. Let S be a subset of $\{1, ..., n\}$, where $n \geq 2$. $|S|$ is the number of elements in S, and $max.S$ and $min.S$ are the largest and the smallest elements in S, respectively. The *width* of S, denoted $width.S$, is defined as follows: $width.S = 1 + max.S - min.S$. We prove the following two impossibility results,

- For any set $|S| \geq 2$, it is not possible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers.
- For any two sets S and T, and integer k, if $|T| \geq 2$, $k \in T$, $k \notin S$ and $k \leq width.T$ then it is not possible to implement a T-free consensus object for n processes using any number of S-free consensus objects for n processes and registers.

It follows from the results that: (1) for any $2 \leq k \leq n$, it is not possible to implement a $\{1, k\}$-free consensus object for k processes using any number of wait-free consensus

objects for $n-1$ processes and registers; and (2) For any $n > 2$, it is not possible to implement a wait-free consensus object for two processes using any number of $\{1, n\}$-free consensus objects for n processes and registers.

Cooperation vs. contention. It follows from the above impossibility results that objects which satisfy cooperation-based progress conditions can not be used to implement objects which satisfy contention-based progress conditions. However, objects that satisfy cooperation-based conditions can be implemented from objects that satisfy the corresponding contention-based conditions. More formally,

- It is not possible to implement $\{1, 2\}$-free consensus object for n processes using $\{2, ..., n\}$-free consensus objects for n processes and registers. However, for $2 \leq k \leq n$, it is possible to implement an $\{n-k+1, ..., n\}$-free consensus object for n processes using $\{1, ..., k\}$-free consensus objects for n processes and registers.

This result is rather surprising, given the fact that while cooperation-based conditions imply fault-freedom, contention-based conditions do not imply the fault-freedom.

Asymmetric progress conditions. We establish a connection between symmetric and asymmetric conditions, which enables us to apply the general impossibility results for proving new results also for asymmetric conditions. For example, we show that:

- For any two integers k_1 and k_2 such that $1 \leq k_1 < k_2 \leq n$, it is not possible to implement a consensus object for n processes that satisfies $\{k_1, k_2\}$-freedom for $n - k_2 + 1$ processes and satisfies $\{k_1\}$-freedom for all the other processes, using any number of wait-free consensus objects for $k_2 - k_1$ processes and registers.

Atomic registers. When only registers are used, we have a complete characterization under which symmetric conditions consensus is solvable, and prove impossibility results for the asymmetric case. For the symmetric case, we show that:

- For any set S, it is possible to implement an S-free consensus object for n processes using registers if and only if $|S| = 1$.

The results generalize the famous FLP result for the case of one faulty process [6,18].

Universality. We generalize results regarding the universality of consensus from [10]. An object o is *S-universal* for n processes if any object which has sequential specification has an S-free linearizable implementation using registers and objects of type o for n processes. We prove that,

- For any positive integer n, and any non-empty set $S \subseteq \{1, ..., n\}$, an S-free consensus object for n processes is S-universal for n processes.

The result implies that an object o is S-universal for n processes if and only if an S-free consensus object for n processes can be implemented from objects of type o and registers. The *wait-free hierarchy* [10], is an infinite hierarchy of objects, such that the objects at level i are exactly those objects which are $\{1, ..., i\}$-universal for i processes, but are not $\{1, ..., i+1\}$-universal for $i+1$ processes. We will explain, how to define other interesting hierarchies.

1.4 Related Work

The consensus problem was formally defined in [20]. The impossibility result that there is no consensus algorithm that can tolerate even a single crash failure in an asynchronous model was first proved for the message-passing model in [6], and later has been extended for the shared memory model in which only atomic registers are supported, in [18]. A recent survey which covers many related impossibility results can be found in [4]. The power of shared objects has been studied extensively in environments where processes may fail benignly, and where every operation is wait-free. In [10], Herlihy classified objects by their consensus numbers and defined the wait-free hierarchy. Additional results regarding the wait-free hierarchy can be found in [14,16].

Objects that can be used, together with registers, to build wait-free implementations of any other object are called *universal objects*. Previous work provided methods, called universal constructions, to transform sequential specifications of arbitrary shared objects into wait-free concurrent implementations that use universal objects [10,21]. In [21] it is proved that sticky bits are universal, and independently, in [10] it is proved that wait-free consensus objects are universal. A bounded space version of the universal construction from [10] appears in [15]. Linearizability is defined in [12].

Two extensively studied conditions are wait-freedom [10] and obstruction-freedom [11]. It is shown in [11] that obstruction-free consensus is solvable using registers. Various contention management techniques have been proposed to improve obstruction-freedom under contention [7,22]. Other works investigated boosting obstruction-freedom by making timing assumption [1,5] and using failure detectors [8]. Wait-free consensus algorithms that use registers in the absence of contention and revert to using strong synchronization operations when contention occurs, are presented in [2,17,19].

The notion of *asymmetric* progress conditions was coined in [13], where the (n, x)-liveness condition which guarantees wait-freedom for x processes and obstruction-freedom for the remaining $n - x$ processes, was defined. The following results are proven in [13]: (1) It is not possible to implement an $(n, 1)$-*live* consensus object using wait-free consensus objects for $n - 1$ processes and registers; (2) For $1 \leq x < n - 1$, an (n, x)-live consensus object is strictly weaker than an $(n, x + 1)$-live consensus object, thereby establishing a hierarchy for (n, x)-liveness; (3) It is not possible to implement a consensus object for n processes which guarantees both fault-freedom and obstruction-freedom for one process and only obstruction-freedom for the remaining $n-1$ processes, using wait-free consensus objects for $n-1$ processes and registers; (4) It is possible to implement a consensus object for $n \geq x$ processes that satisfies a condition called asymmetric group-based progress condition using (x, x)-live consensus objects and registers.

The notion of k-obstruction-freedom is presented in [24], as part of a transformation that is used to fuse objects which avoid locking and locks together in order to create new types of shared objects. In [25], a new classification for evaluating the strength of shared objects is proposed. The classification is based on finding, for each object of type o, the largest k for which it is possible to solve consensus for any number processes, using any number of objects of type o and registers, assuming that the required progress condition is k-obstruction-freedom. The main technical result in [25] is that the new classification is equivalent to Herlihy's traditional classification.

Although progress conditions and adversaries are two seemingly different notions, they are actually closely related. In [3], a precise way is presented to characterize adversaries by introducing the notion of disagreement power: the biggest integer k for which the adversary can prevent processes from agreeing on k values when using registers only; and it is shown how to compute the disagreement power of an adversary. Our formalism for expressing progress conditions is not expressive enough to express all the adversaries considered in [3], and vice versa. In the last section, we generalize our formalism to express both.

2 Preliminaries

Our model of computation consists of an asynchronous collection of n processes that communicate via shared objects. An *event* corresponds to an atomic step performed by a process. For example, the events which correspond to accessing registers are classified into two types: read events which may not change the state of the register, and write events which update the state of a register but does not return a value. We use the notation e_p to denote an instance of an arbitrary event at a process p.

A *run* is a pair $(f,\ R)$ where f is a function which assigns initial states (values) to the objects and R is a finite or infinite sequence of events. An implementation of an object from a set of other objects, consists of a non-empty set C of runs, a set N of processes, and a set of shared objects O. For any event e_p at a process p in any run in C, the object accessed in e_p must be in O. Let $x = (f,\ R)$ and $x' = (f',\ R')$ be runs. Run x' is a *prefix* of x (and x is an *extension* of x'), denoted $x' \leq x$, if R' is a prefix of R and $f = f'$. When $x' \leq x$, $(x - x')$ denotes the suffix of R obtained by removing R' from R. Let $R;T$ be the sequence obtained by concatenating the finite sequence R and the sequence T. Then $x;T$ is an abbreviation for $(f,\ R;T)$.

Process p is *enabled* at run x if there exists an event e_p such that $x;e_p$ is a run. For simplicity, we write xp to denote either $x;e_p$ when p is enabled in x, or x when p is not enabled in x. Register r is a *local* register of p if only p can access r. For any sequence R, let R_p be the subsequence of R containing all events in R which involve p. Runs $(f,\ R)$ and $(f',\ R')$ are *indistinguishable* for a set of processes P, denoted by $(f,\ R)[P](f',\ R')$, iff for all $p \in P$, $R_p = R'_p$ and $f(r) = f'(r)$ for every local register r of p. When $P = \{p\}$ we write $[p]$ instead of $[P]$. It is assumed that the processes are deterministic, that is, if $x;e_p$ and $x;e'_p$ are runs then $e_p = e'_p$.

The runs of an asynchronous implementation of an object must satisfy several properties. For example, if a *write* event which involves p is enabled at run x, then the same event is enabled at any finite run that is indistinguishable to p from x. In the following proofs, we will implicitly make use of few such straightforward properties.

3 Impossibility Results

We use S and T to denote non-empty sets which are subsets of $\{1, ..., n\}$; $|S|$ is the number of elements in S, and $max.S$ and $min.S$ are the largest and the smallest elements in S, respectively. The *width* of S, denoted $width.S$, is defined as follows: $width.S = 1 + max.S - min.S$. Thus, the width of the set $\{1, ..., n\}$ is n. We notice that it is always the case that $width.S \geq |S|$.

Theorem 1. *For any set $|S| \geq 2$, it is not possible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers.*

It follows immediately from Theorem 1 that for any $2 \leq k \leq n$, it is not possible to implement a $\{1, k\}$-free consensus object for n processes using any number of wait-free consensus objects for $k - 1$ processes and registers. Next we consider the relative strength of different condition for the same number of processes.

Theorem 2. *For any two sets S and T, and integer k, if $|T| \geq 2$, $k \in T$, $k \notin S$ and $k \leq width.T$ then it is not possible to implement a T-free consensus object for n processes using any number of S-free consensus objects for n processes and registers.*

It follows from Theorem 2 that: For any $n > 2$, it is not possible to implement a wait-free consensus object for two processes using any number of $\{1, n\}$-free consensus objects for n processes and registers. Next we prove the theorems. For lack of space, the proofs of all the lemmas appear only in the full version. (The full version of the paper can be downloaded from: www.faculty.idc.ac.il/gadi .)

3.1 A Detailed Proof

The proofs of Theorem 1 and Theorem 2 use the following notions, abbreviations, and lemmas. Let N be the set of all n processes, and let $P \subseteq N$. A finite run x is (P, v)-*valent* if in all extensions of x, by processes in P only, where a decision is made, the decision value is v ($v \in \{0, 1\}$). A run is P-*univalent* if it is either $(P, 0)$-valent or $(P, 1)$-valent, otherwise it is P-*bivalent*. We say that two P-univalent runs are P-*compatible* if they have the same valency, that is, either both runs are $(P, 0)$-valent or both are $(P, 1)$-valent. Finally, we say that process $p \in P$ is a P-*decider* at run x if for every extension y of x by steps of processes from P only (i.e. $x[N - P]y$), the run yp is P-univalent. Recall that we assume that $S \subseteq \{1, ..., n\}$.

Lemma 1. *Let $|S| \geq 2$, and let P be a set of processes such that $|P| = max.S$. Then, for every $p \in P$, there is at least one subset of P, denoted $p.SP$, of size $min.S$ which does* not *include p.*

Lemma 2. *For a set S and non-empty sets of processes P and Q such that $|P| \in S$, $|Q| \in S$ and $Q \subseteq P$, in any S-free consensus object, if two P-univalent runs are indistinguishable for Q and the state of all the objects that (processes in) Q can access are the same at these runs, then these runs must be P-compatible.*

Lemma 3. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Then, every S-free consensus object has a P-bivalent empty run.*

Lemma 4. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Let y be a run of an S-free consensus object, and let $p \in P$ and $q \in P$ be two different processes such that (1) $y \neq yp$ and $y \neq yq$, (2) the runs yp and yqp are P-univalent and not P-compatible. Then, in their two next events from y, p and q are accessing the same object, and this object is not a register.*

Lemma 5. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. For every S-free consensus object there is a P-bivalent run x and process $p \in P$ such that p is a P-decider at x.*

Lemma 6. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Every S-free consensus object has a P-bivalent run y and two processes $p \in P$ and $q \in P$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their two next events from y, p and q are accessing the same object, and this object is not a register.*

Lemma 7. *Let $|S| \geq 2$ and let P be a set of processes such that $|P| = max.S$. Every S-free consensus object has a P-bivalent run y, a set $Q \subseteq P$ of size $width.S$, and two processes $p \in Q$ and $q \in Q$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their next events from y, all the $width.S$ processes in Q, are accessing the same object, and this object is not a register.*

Proof of Theorem 1. It follows from Lemma 7 that every implementation of an S-free consensus object for n processes, must use an object, say o, which at least $width.S$ processes must be able to access at the same run, and o is not a register. Thus, it is not possible to implement an S-free consensus object for n processes using any number of wait-free consensus objects for $width.S - 1$ processes and registers. □

Proof of Theorem 2. Assume to the contrary that there is such an implementation of a T-free consensus object for n processes from S-free consensus objects for n processes and registers. Let P be a set of processes such that $|P| = max.T$. It follows from Lemma 7 that such an implementation has a P-bivalent run y, a set $Q \subseteq P$ of size $width.T$, and two processes $p \in Q$ and $q \in Q$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their next events from y, all the $width.S$ processes in Q, are accessing the same object, say o, and this object is not a register. Thus, it must be the case that o is an S-free consensus object.

Assume that at the end of y, just before the $width.T$ processes access o, $n - k$ processes fail and the remaining k active processes, including p and q, are about to access o. Since there are only k active processes and $k \in T$, the implementation of a T-free consensus object must guarantee that these k processes will eventually properly terminate. However, since $k \notin S$, the S-free consensus object o does not guarantee that any of the remaining k active processes will ever get a response from o. Assume none of the k processes ever gets a response for o. Although the k processes may continue to take steps, because yp and yqp are P-univalent and not P-compatible, the final decision value (of the T-free consensus object) depends on getting a response from o. Without a response from o, it is not possible to determine whether the prefix of the current run is yp or yqp. Thus, the k processes will never be able to terminate. A contradiction. □

4 Cooperation vs. Contention

It follows from the impossibility results that objects which satisfy cooperation-based progress conditions can not implement objects which satisfy contention-based progress conditions. More formally,

Theorem 3. *It is not possible to implement $\{1,2\}$-free consensus object for n processes using $\{2,...,n\}$-free consensus objects for n processes and registers.*

Proof. Let T be the set $\{1,2\}$, and $k = 1$. Then, (1) $|T| \geq 2$, (2) $k \in T$, (3) $k \notin \{2,...,n\}$, and (4) $k \leq width.T$. Thus, the result follows from Theorem 2. □

Next we show that objects that satisfy cooperation-based conditions can be implemented from objects that satisfy the corresponding contention-based conditions.

Theorem 4. *For $2 \leq k \leq n$, it is possible to implement an $\{n-k+1,...,n\}$-free consensus object for n processes using $\{1,...,k\}$-free consensus objects for n processes and registers.*

To prove theorem 4, we first generalize a known result for wait-freedom, namely, that multi-valued consensus can be implemented from binary consensus ([23], page 329).

Lemma 8. *For any $k \geq 2$, $n \geq 2$ and $S \subseteq \{1,...,n\}$, an S-free k-consensus object for n processes can be implemented from S-free binary consensus objects for n processes and atomic bits.*

Proof. To implement a singe k-consensus object, we use $\lceil \log k \rceil$ binary consensus objects, which are numbered 0 through $\lceil \log k \rceil - 1$, and k bits which are numbered 0 through $k-1$ and are initialized to 0. To propose a value $v \in \{0,...,k-1\}$, p does the following: (1) it sets the bit number v to 1; (2) it proposes the binary encoding of v, bit by bit, to the binary consensus objects in an increasing order starting from number 0. If at some point during the second step the bit p has proposed is not accepted as the consensus value at the corresponding binary consensus object, p stops proposing v, scan the bits and chooses one of the bits that are set to 1, say v', which also matches the values that has successfully proposed so far and continues to propose the value v'. This procedure continues until p proposes, to all the $\lceil \log k \rceil$ binary consensus objects. The value that its binary encoding was successfully proposed to all the $\lceil \log k \rceil$ binary consensus objects is the final consensus value. □

Proof of Theorem 4: Build a tree of degree k with $\lceil n/k \rceil$ leaves, and where each node of the tree is a $\{1,...,k\}$-free k-consensus object. Each participating process is progressing from a leaf to the root, where at each level of the tree it accesses a k-consensus object, competing against at most $k-1$ processes in its neighbor's subtree. As a process advances towards the root, it plays the role of process 0 (i.e., proposes 0) when it arrives from the left most subtree, of process $k-1$ when it arrives from the right most subtree, or of process $0 \leq i \leq k-1$ when it arrives from the i'th subtree. The winner at each node is the process its value is being agreed upon. Only a winner at a given node continues to progress towards the root. The value agreed at the root is the final decision value. Each of the processes that accesses the root writes the final decision value at a special register called *decision*, and decides on that value. Each process that loses at some node other than the root, spins on the *decision* register until a value is written into it and decides on that value. □

5 Asymmetric Progress Conditions

As already mentioned, the notion of asymmetric progress conditions was coined and investigated in [13]. Let *APC* be an Asymmetric Progress Condition; we define *max.APC*, *min.APC* and *width.APC* as follows,

- *max.APC* is the largest $1 \leq k \leq n$ such that (at least) $n - k + 1$ processes are $\{k\}$-free, or 0 if no such k exists.
- *min.APC* is the smallest $1 \leq k \leq n$ such that *every* process is $\{k\}$-free, or 0 if no such k exists.
- $width.APC$ equals $1 + max.APC - min.APC$ if $min.APC \neq 0$, or 0 otherwise.

Thus, for the asymmetric progress condition $(n, 1)$-liveness (as defined in [13]), $max.(n, 1)\text{-}liveness = n$, $min.(n, 1)\text{-}liveness = 1$ and $width.(n, 1)\text{-}liveness = n$.

Lemma 9. *Let O be a consensus object for n processes that satisfies an asymmetric progress condition APC such that $min.APC \geq 1$. Using O and a single register it is possible to implement a consensus object for n processes that satisfies the symmetric progress condition (min.APC, max.APC)-freedom.*

Proof. Let *decision* be a register which is initially set to -1. Each process tries to reach a decision by accessing O. A process that reaches a decision writes the decision value into *decision* and terminates. Each process infinitely often reads *decision*, and if the value read is different from -1, it decides on that value and terminates. Since every subset of *max.APC* processes includes at least one $\{max.APC\}$-free process, this implementation clearly satisfies (*min.APC*, *max.APC*)-freedom. Another way to view this implementation is: once O returns a value to some processes, it keeps this value in an internal private register, and thereafter returns it immediately to every process that accesses it. □

Lemma 10. *Let APC be a an asymmetric progress condition such that $1 \leq min.APC < max.APC \leq n$, and let P be a set of processes such that $|P| = max.APC$. Every consensus object for n processes that satisfies APC has a P-bivalent run y, a set $Q \subseteq P$ of size $width.APC$, and two processes $p \in Q$ and $q \in Q$ such that: (1) p is a P-decider at y; (2) the runs yp and yqp are P-univalent and not P-compatible; and (3) in their next events from y, all the $width.APC$ processes in Q, are accessing the same object, and this object is not a register.*

Proof. Assume to the contrary that O is a consensus object for n processes that satisfies *APC*, and O does not have a run y with *all* the three properties as mentioned in Lemma 10. By Lemma 9, using O and a single register it is possible to implement a consensus objects O' for n processes that satisfies the symmetric progress condition (*min.APC*, *max.APC*)-freedom. Thus, also O' does not have such a a run y. However, this contradicts Lemma 7. □

Theorem 5. *For any asymmetric progress condition APC such that $1 \leq min.APC < max.APC \leq n$, it is not possible to implement a consensus object for n processes that satisfies APC using any number of wait-free consensus objects for $width.APC - 1$ processes and registers.*

Proof. The proof is similar to that of Theorem 1. It follows from Lemma 10 that every implementation of a consensus object for n processes that satisfies *APC*, must use an object, say o, which at least $width.APC$ processes must be able to access at the same run, and o is not a register. Thus, it is not possible to implement a consensus object for n processes that satisfies APC using any number of wait-free consensus objects for $width.APC - 1$ processes and registers. □

It is proven in [13] that it is not possible to implement an $(n, 1)$-*live* consensus object using any number of wait-free consensus objects for $n - 1$ processes and registers; and that this result holds even when the requirement that one process should be wait-free is replaced with the much weaker requirement that one process is $\{1, n\}$-free. These important results are special cases of the following corollary of Theorem 5.

Corollary 1. *For any two positive integers k_1 and k_2 such that $1 \leq k_1 < k_2 \leq n$, it is not possible to implement a consensus object for n processes, that satisfies $\{k_1, k_2\}$-freedom for $n - k_2 + 1$ processes and satisfies $\{k_1\}$-freedom for all the other processes, using any number of wait-free consensus objects for $k_2 - k_1$ processes and registers.*

Another interesting result from [13] is that: For $1 \leq x < n-1$, an (n, x)-live consensus object is strictly weaker than an $(n, x+1)$-live consensus object, thereby establishing a hierarchy for (n, x)-liveness. Using Lemma 10 it is possible to slightly generalize this result.

6 Atomic Registers

For the case where only registers are used, we present a complete characterization under which symmetric progress conditions consensus is solvable, and prove impossibility results for the asymmetric case.

Theorem 6

- *For any set S, it is possible to implement an S-free consensus object for n processes using registers if and only if $|S| = 1$.*
- *For any asymmetric progress condition APC, it is not possible to implement a consensus object for n processes that satisfies APC using registers if $width.APC > 1$.*

Proof. It follows from Theorem 1 that it is not possible to implement an S-free consensus object for n processes using registers if $|S| \geq 2$, and it follows from Theorem 5 that it is not possible to implement a consensus object for n processes that satisfies APC using registers if $width.APC > 1$.

Next, we show that for any integer $1 \leq k \leq n$, it is possible to implement a $\{k\}$-free consensus object for n processes using registers. The algorithm (i.e., implementation) proceeds in rounds. The notion of a *round* is used only for the sake of describing the algorithm. We do *not* assume a synchronous model of execution in which all the processes are always executing the same round.

Each process has a preference for the decision value in each round; initially this preference is the input value of the process. If no decision is made in a round then the processes advance to the next round, and try again to reach agreement.

IMPLEMENTING $\{k\}$-FREE CONSENSUS FOR n PROCESSES USING REGISTERS (WHERE $k \in \{1, ..., n\}$): program for process p_i with input in_i (where $in_i \in \{0, 1\}$ and $i \in \{1, ..., n\}$) .

shared registers
$x[0..\infty, 0..1]$ infinite array of bits, initially $x[0, 0] = x[0, 1] = 1$ and all other entries are 0
$flag[1..\infty, 1..n]$ infinite array of bits, initially all entries are 0
$decide$ ranges over $\{\perp, 0, 1\}$, initially $\perp$

local registers
r_i integer, initially 1
v_i bit, initially in_i ; l_i,$count_i$ integers, initial values are immaterial

```
1  while decide = ⊥ do
2      if x[r_i, 0] = 0 and x[r_i, 1] = 0 then x[r_i, v_i] := 1 fi   /* preferred value */
3      flag[r_i, i] := 1                                  /* signal participation */
4      if x[r_i − 1, 1 − v_i] = 0 then decide := v_i      /* no conflict in r_i − 1 */
5          else repeat                                            /* k-barrier */
6                   count_i = 0                  /* initialize local counter */
7                   for l_i = 1 to n do if flag[r_i, l_i] = 1 then count_i := count_i + 1 fi od
8               until (count_i ≥ k)                  /* at least k participate */
9               if x[r_i, 0] = 1 then v_i := 0 else v_i := 1 fi    /* value for r_i + 1 */
10     fi
11     r_i := r_i + 1
12 od
13 decide(decide)
```

In round $r \geq 1$, process p_i first checks if the bit of its preference v_i and of the opposite value $1 - v_i$ are set. If both bits are not set, p_i sets its preference bit v_i by writing 1 to $x[r, v_i]$ (line 2). Then, p_i sets its participation bit by writing 1 to $flag[r_i, i]$ (line 3). Next, p_i reads the bit $x[r - 1, 1 - v_i]$. If the bit $x[r - 1, 1 - v_i]$ is not set, then every process that reaches round r with the conflicting preference $1 - v_i$ will find that only $x[r, v_i]$ is set to 1, will never set $x[r, 1 - v_i]$ to 1. Consequently, process p_i can safely decide on v_i, and it writes v_i to *decide* (line 4). Otherwise, waits until it notices that at least k processes are participating in round r (lines 5–8). After that p_i updates its preference in an attempt to agree with the other processes (line 9). Then, p_i proceeds to round $r + 1$ (line 11).

If *exactly* k processes with possibly conflicting preferences participate in round r, then they will reach line 9, only *after* all of them set their flags in line 3. This implies that once some process reaches line 9, no process is at line 2, and hence all the k processes will reach round $r + 1$ with the same preference which is the value chosen in line 9. When all processes reach a round with the same preference, a decision is reached either in that round or the next round. □

7 Universality

In [10], the notion of universality is introduced in the context of wait-freedom. An object o is (wait-free) *universal* for n processes if any object which has sequential specification

has a wait-free linearizable implementation using registers and objects of type o in a system with n processes. Below we generalize the notion of wait-free universality.

> **Definition.** An object o is *S-universal* for n processes if any object which has sequential specification has an S-free linearizable implementation using registers and objects of type o for n processes.

One of the important results proved in [10], is that wait-free consensus for n processes is universal for n processes. Next we generalize this result.

Theorem 7. *For any positive integer n, and any non-empty set $S \subseteq \{1, ..., n\}$, an S-free consensus object for n processes is S-universal for n processes.*

To prove the result, we present a universal construction that implements any S-free object o for n processes from S-free consensus objects for S processes and registers. The construction conceptually mimics the original construction for the wait-free model from [10]. In the full version of the paper, we give such a construction, which is similar to the one for the wait-free model from [23]. A similar type of a universality result (with a similar proof) can be proved also for asymmetric progress conditions.

Corollary 2. *For any object o, any positive integer n, and any non-empty set $S \subseteq \{1, ..., n\}$, o is S-universal for n processes if and only if an S-free consensus object for n processes can be implemented from objects of type o and registers.*

The *wait-free hierarchy* is an infinite hierarchy of objects, introduced in [10], such that the objects at level i of the hierarchy are exactly those objects which are $\{1, ..., i\}$-universal for i processes, but are not $\{1, ..., i+1\}$-universal for $i+1$ processes. For that hierarchy, by the above definition, (1) no object at level less than i together with registers can implement any object at level i; and (2) each object at level i together with registers can implement any object at level i or at a lower level.

The wait-free hierarchy is meaningful because it can be defined using only the (contention-based) progress conditions $\{1, ..., k\}$-freedom, for all k. In such a case, there is a total order, based on the stronger than relation, between all these conditions. Similar such hierarchies, in which there is a total order between the conditions, can be naturally defined. For example, by using the cooperation-based progress conditions, the cooperation hierarchy can be defined as follows: For a given system of n processes, the objects at level i of the hierarchy are exactly those objects which are $\{n-i+1, ..., n\}$-universal for n processes, but are not $\{n-i, ..., n\}$-universal for n processes.

8 Discussion

It is possible to extend the definitions of progress conditions in various ways. Below we define two such new interesting extensions.

Definition. For any non-empty set $S \subseteq \{1, ..., n\}$ and an integer $1 \leq k \leq n$, the progress condition (S, k)*-freedom* guarantees that for every set of processes P, if at some point in a computation $active.P = |P|$ and $|P| \in S$, then (at least) $\min\{k, |P|\}$ processes in P will be able to eventually complete their pending operations, provided

that (1) all the processes not in P do not take steps for long enough; and (2) none of the processes in P fails.

We notice that in a system of n processes, (S, n)-*freedom* is the same as S-*freedom*; and $(\{1, ..., n\}, 1)$-freedom is the same as a known condition called *non-blocking* [12] (sometimes also called lock-freedom).

Definition. Let $W_1, ..., W_n$ be sets of sets of process identifiers such $P \in W_i$ only if $p_i \in P$. The progress condition $(W_1, ..., W_n)$-*freedom* guarantees that for every set of processes P and every process p_i, if at some point in a computation $active.P = |P|$ and $P \in W_i$, then process p_i will be able to eventually complete its pending operations, provided that (1) all the processes not in P do not take steps for long enough; and (2) none of the processes in P fails.

Each one of the adversaries considered in [3] corresponds to some $(W_1, ..., W_n)$-free progress condition, which has the following property: For every set P, if $P \in W_i$ and $p_j \in P$ then $P \in W_j$. We notice that satisfying this property, completely precludes the ability to express the asymmetric progress conditions defined in the introduction. That is, w.r.t. this definition, this property distinguishes between symmetric and asymmetric progress conditions (adversaries).

Additional interesting questions are: exploring the complexity and computability of problems like set-consensus, renaming, etc. under various new progress conditions; exploring the relation to failure detectors, by possibly extending known results for wait-freedom [9]; defining meaningful hierarchies; better understanding of the relations between different progress conditions; adding timing assumptions.

Known open problems, like the robustness of the wait-free hierarchy or whether a queue object can be implemented from a set of test-and-set objects, fetch-and-add objects, swap objects and atomic registers, for $n \geq 3$, can now be studied in our more general setting.

The study should not be limited to shared memory systems only. Consider for example n senders that are trying to broadcast the same message to a single receiver, and it is required that at least one of the senders succeeds to transmit, without collisions, whenever an *odd* number of senders broadcast at the same time. This required progress condition, and similar ones, that are sometimes expressed using the notion of a conflict graph, can be easily formally expressed and studied within our general framework.

References

1. Aguilera, M.K., Toueg, S.: Timeliness-based wait-freedom: a gracefully degrading progress condition. In: Proc. 27rd ACM Symp. on Principles of Distributed Computing, pp. 305–314 (2008)
2. Attiya, H., Guerraoui, R., Kouznetsov, P.: Computing with reads and writes in the absence of step contention. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 122–136. Springer, Heidelberg (2005)
3. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Tielmann, A.: The disagreement power of an adversary. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 8–21. Springer, Heidelberg (2009)
4. Fich, F.E., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing 16(2-3), 121–163 (2003)

5. Fich, E.F., Luchangco, V., Moir, M., Shavit, N.: Obstruction-free algorithms can be practically wait-free. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 78–92. Springer, Heidelberg (2005)
6. 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)
7. Guerraoui, R., Herlihy, M.P., Pochon, B.: Towards a theory of transactional contention managers. In: Proc. of the 24th Symp. on Principles of Dist. Computing, pp. 258–264 (2005)
8. Guerraoui, R., Kapalka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. Distributed Computing 20(6), 415–433 (2008)
9. Guerraoui, R., Kouznetsov, P.: Failure detectors as type boosters. Distributed Computing 20, 343–358 (2008)
10. Herlihy, M.P.: Wait-free synchronization. ACM Trans. on Programming Languages and Systems 13(1), 124–149 (1991)
11. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proc. of the 23rd Int. Conf. on Dist. Computing Systems (2003)
12. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS 12(3), 463–492 (1990)
13. Imbs, D., Raynal, M., Taubenfeld, G.: On asymmetric progress conditions. In: Proc. 29th ACM Symp. on Principles of Distributed Computing (to appear, 2010)
14. Jayanti, P.: Robust wait-free hierarchies. Journal of the ACM 44(4), 592–614 (1997)
15. Jayanti, P., Toueg, S.: Some results on the impossibility, universality, and decidability of consensus. In: Segall, A., Zaks, S. (eds.) WDAG 1992. LNCS, vol. 647, pp. 69–84. Springer, Heidelberg (1992)
16. Lo, W.-K., Hadzilacos, V.: All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. SIAM Journal on Computing 30(3), 689–728 (2000)
17. Luchangco, V., Moir, M., Shavit, N.: On the uncontended complexity of consensus. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 45–59. Springer, Heidelberg (2003)
18. Loui, M.C., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing Research 4, 163–183 (1987)
19. Merritt, M., Taubenfeld, G.: Resilient consensus for infinitely many processes. In: Fich, F.E. (ed.) DISC 2003. LNCS, vol. 2848, pp. 1–15. Springer, Heidelberg (2003)
20. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)
21. Plotkin, S.A.: Sticky bits and universality of consensus. In: Proc. 8th ACM Symp. on Principles of Distributed Computing, pp. 159–175 (1989)
22. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: Proc. of the 24th Symp. on Principles of Dist. Computing, pp. 240–248 (2005)
23. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Pearson/Prentice-Hall (2006) ISBN 0-131-97259-6
24. Taubenfeld, G.: Contention-sensitive data structures and algorithms. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 157–171. Springer, Heidelberg (2009)
25. Taubenfeld, G.: On the computational power of shared objects. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 270–284. Springer, Heidelberg (2009)

Scalable Quantum Consensus for Crash Failures

Bogdan S. Chlebus[1], Dariusz R. Kowalski[2], and Michał Strojnowski[3]

[1] Department of Computer Science and Engineering
University of Colorado Denver, U.S.A.
[2] Department of Computer Science
University of Liverpool, U.K.
[3] Instytut Informatyki
Uniwersytet Warszawski, Poland

Abstract. We present a scalable quantum algorithm to solve binary consensus in a system of n crash-prone quantum processes. The algorithm works in $\mathcal{O}(\text{polylog } n)$ time sending $\mathcal{O}(n \text{ polylog } n)$ qubits against the adaptive adversary. The time performance of this algorithm is asymptotically better than a lower bound $\Omega(\sqrt{n/\log n})$ on time of classical randomized algorithms against *adaptive* adversaries. Known classical randomized algorithms having each process send $\mathcal{O}(\text{polylog } n)$ messages work only for *oblivious* adversaries. Our quantum algorithm has a better time performance than deterministic solutions, which have to work in $\Omega(t)$ time for $t < n$ failures.

1 Introduction

Quantum computing is a paradigm of information processing that attempts to harness the phenomena of quantum mechanics to enhance computing power. Recent success stories in this area indicate that structuring computation on the quantum level can enhance performance so that classical complexity bounds could be surpassed.

Quantum *centralized computation* can be represented by quantum circuits. There is a finite number of types of quantum gates that together are universal, in the sense that they can approximate a given general unitary transformation of qubits with arbitrary precision of approximation. This model of quantum centralized computing can be generalized to incorporate faults, see [1].

Quantum *distributed computing* is a study of quantum machines which process locally either qubits or bits and communicate among themselves by either quantum or classical channels. The motivation for seeking distributed quantum-computing solutions is that building a big single quantum computer with many qubits may be prohibitively difficult and expensive, so replacing such a machine by a network of small quantum computers could be a viable solution to carry out computation on a scale that would yield practical benefits. Recent work on distributed quantum computing was surveyed by Broadbent and Tapp [5], Buhrman and Röhrig [6], and Denchev and Pandurangan [14].

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 236–250, 2010.

The classical versus quantum communication. Quantum communication channels can transmit qubits, similarly as classical channels transmit bits. It is known that quantum communication can be simulated by first distributing entangled pairs of qubits and next sending classical bits, for a cost of two bits per qubit; this mechanism is often called teleportation. When both qubits are sent and entangled pairs are used, then it is possible to transmit two classical bits by sending one qubit over a quantum channel, this mechanism is called super-dense coding. The optimality of such simulations between modes of communication was investigated by Bennett and Wiesner [4], Cleve, van Dam, Nielsen, and Tapp [12], and Holevo [19]. A conclusion is that if entangled pairs can be distributed or quantum channels are available, then there exist mutual simulations between classical and quantum channels with only a constant overhead. For more on classical versus quantum communication complexity see the papers by Cleve and Buhrman [11], de Wolf [13], and Ta-Shma [26].

Scalability. A postulate to require that each process contributes a sub-linear number of operations during an execution motivates the notion of scalability as a desirable characteristic of distributed algorithms, see [8,9,20,21,22]. Given a performance metric, it is natural to assume $\mathcal{O}(n \operatorname{polylog} n)$ to be the amount contributed by all processes running a scalable algorithm in a large distributed system. Scalability comes in two variants: global and local [9]. *Global scalability* means that $\mathcal{O}(n \operatorname{polylog} n)$ is a bound on the complexity. *Local scalability* means that $\mathcal{O}(\operatorname{polylog} n)$ is the maximum contribution to the complexity per process.

We are concerned with the scalability of communication measured as the number of transmitted classical bits and quantum qubits. In our algorithms, the nodes communicate among themselves using a sparse network to save on communication. In each round, a node receives messages from its neighbors, processes them and sends new messages to the neighbors. This means that scalability hinges on good properties of the underlying networks. While quantum information gets proliferated, all local processing by nodes is performed by quantum circuitry that implements suitable unitary transformations. Only after this has been concluded, measurements are made to determine classical bits. What follows after that is classical computation along with classical communication.

The problem. We study the *binary consensus problem.* Each process is initialized with either 0 or 1 and it must eventually decide on a value from among the initial ones. The decision value is to be the same across all the processes. A deterministic version of the consensus problem is normally specified by the standard requirements of agreement, validity, and termination. A randomized version of the problem has agreement formulated as ε*-agreement*, for $0 < \varepsilon < 1$: all the processes decide on the same value with the probability at least $1 - \varepsilon$.

Our results. We develop scalable quantum algorithms to solve binary consensus in a system of n crash-prone processes that use both quantum and classical channels to communicate. The algorithms are scalable with respect to both time and communication, in the sense that they work in time $\mathcal{O}(\operatorname{polylog} n)$ by

sending $\mathcal{O}(n \operatorname{polylog} n)$ qubits and bits. The algorithms are randomized in the outcomes of measurements of some qubits, so their performance bounds are expected. No pre-entangled qubits pre-distributed among the nodes are used. We give four consensus solutions, two classical randomized and two quantum ones. All the algorithms solve consensus with ε-agreement, for ε treated as a parameter. We begin with a randomized algorithm RandomizedConsensus which solves consensus when $t < n/3$. Algorithm QuantumConsensus is obtained from that randomized one by packing all possible random choices by way of superpositions; the algorithm achieves its performance bounds against the adaptive adversary. Next we give a randomized algorithm ExtendedRandomizedConsensus which is more involved but works for any upper bound on the number of crashes $t < n$. Finally we transform that randomized algorithm to ExtendedQuantumConsensus which also works for any upper bound on the number of crashes $t < n$. The randomized algorithms have their performance bounds proven against the content-oblivious adversary, while the quantum algorithms achieve their performance bounds against the adaptive adversary.

Previous work. Quantum consensus solutions were given by Ben-Or and Hassidim [3]. Their algorithms work in the expected $\mathcal{O}(1)$ time and send $\Omega(n^2)$ qubits against adaptive adversaries that control crashes. The algorithms are adaptations to the model of quantum distributed computing of algorithms developed originally for the classical models. Chlebus, Kowalski and Strojnowski [9] gave a globally-scalable deterministic solution of consensus which sends $\mathcal{O}(n \log^4 n)$ bits and is fast in the sense that its time complexity is $\mathcal{O}(t)$ when t is the number of crashes the algorithm tolerates. Randomized classical solutions for consensus problem have been studied against adversaries of various power. Bar-Joseph and Ben-Or [2] proved a lower bound $\Omega(\sqrt{n/\log n})$ on the expected time of randomized solution for consensus against the adaptive adversary when the number of crashes is a constant fraction of n; they also gave a randomized algorithm of time $\mathcal{O}(\sqrt{n/\log n})$ for at most $n/10$ crashes that uses $\Omega(n^2)$ bits of communication. Chlebus and Kowalski [8] developed a locally-scalable randomized consensus solution which terminates in the expected $\mathcal{O}(\log n)$ time while the expected number of bits that each process sends and receives is $\mathcal{O}(\log n)$ if the number of crashes is a constant fraction of the number of processes n and crashes are controlled by an oblivious adversary. Gilbert and Kowalski [17] gave a globally-scalable randomized consensus algorithm that achieves $\mathcal{O}(n)$ bits of communication, and terminates in $\mathcal{O}(\log n)$ time with high probability while tolerating up to $f < n/2$ crash failures.

Related work. The $\Omega(tn)$ lower bound on the message complexity of deterministic solutions for f arbitrary (Byzantine) faults was proved by Dolev and Reischuk [15] and Hadzilacos and Halpern [18]. In contrast to that, Chlebus and Kowalski [7] developed a deterministic early-stopping consensus solution for crash failures of $\mathcal{O}(n \operatorname{polylog} n)$ message complexity; that algorithm has $\Omega(n^2)$ bit communication complexity. Chor, Merritt and Shmoys [10] showed that

randomization allows to obtain a constant expected time algorithm against the oblivious adversary, if a minority of processes may crash.

Scalability has been studied in various distributed settings. King, Saia, Sanwalani and Vee [22] considered scalability in peer-to-peer networks. Holtby, Kapron and King [20] gave a tradeoff between the time required to solve an almost everywhere Byzantine consensus by a randomized scalable solution and how many processes eventually decide on a common value. King and Saia [21] gave a scalable algorithm to agree on a small representative committee of processes in a Byzantine message passing, as a step to facilitate scalable solutions of consensus and leader election. Chlebus, Kowalski and Strojnowski [9] showed that consensus cannot be solved deterministically by an algorithm that is locally scalable with respect to message complexity and can tolerate any number of crashes.

A discussion of six natural models of distributed computing including quantum ones was presented by Gavoille, Kosowski, and Markiewicz [16]. A starting point was the classical graph model of (anonymous) distributed computing [24]. Standard assumptions about this model in its classical extension allow for the knowledge by each node of its name and the number of nodes, while a quantum extension of this model allows for a distribution of pre-entangled states among the nodes of the underlying network. Each of these models could be equipped by either classical or quantum communication channels represented by the edges of the underlying network. The consensus problem requires some communication among the nodes to occur in any of these models, as argued in [16]. This can be contrasted with the problem of leader election: a leader can be elected in such a quantum distributed system without any communication by using only pre-entangled qubits, see [14]. On the other hand, Kobayashi, Matsumoto, and Tani [23] showed that leader election can be solved in anonymous quantum distributed networks without pre-entanglement.

2 Technical Preliminaries

We consider a synchronous distributed system with n processes. The processes are prone to crashes; we denote by $t < n$ an upper bound on the number of crashes that may occur in an execution. The numbers t and n are known to the algorithms, in that they may be part of code. The system can be represented as a network: the processes play the roles of nodes, with some pairs of nodes connected by communication channels that can transfer either qubits or classical bits.

Adversaries. The adversaries we consider have the numbers t and n as parameters. The *oblivious* adversary decides prior to the start of an execution which processes will crash; the timing of crashes during the execution is left for the adversary to control. The *adaptive* adversary can choose online in the course of execution which processes to crash and when this occurs. The adaptive adversary can see the states of all the processes at any round and the contents of all the messages and random bits as soon as they have been generated. We

adjust the notion of adaptive adversary to quantum communication as follows: the adaptive adversary can see the qubit's amplitudes, in the sense that they are available as numbers. This availability in a round occurs just after any operation on the qubit has been completed in the round. The *content-oblivious* adversary is a weakening of the adaptive adversary obtained by imposing a restriction that the adversary cannot read the states of processes nor see the contents of messages. Other than that, the adversary can monitor the traffic of messages and may choose which nodes to crash at any round in an execution.

In a quantum model of computation, there is a way to neutralize some of the power of the adaptive adversary. This may be achieved by having the states of qubits that affect decisions made in the course of an execution contain multiple configurations in superpositions. These qubits need to be processed in a suitably passive way so as not to have them represent any bias until measurements.

Quantum centralized processing. We discuss here the aspects of centralized quantum computing that are used in the pseudo-codes of our algorithms. A centralized quantum algorithm can be represented as a circuit with quantum gates and wires; see [25] for an informative introduction to the related topics. We represent the contents of quantum registers in the standard computational basis. A requirement imposed on a transformation implemented by a quantum circuit is to be unitary. There are small-size sets of gates that are universal in the sense that any unitary transformation can be approximated by a collection of these universal gates. For any collection of universal gates, there exists a unitary operation that requires a simulating circuit to be of exponential size with respect to the number of qubits. Our sequential algorithms run by the nodes require polynomial size in the number n of nodes using standard types of gates. Next we review the information sufficient to show this.

Creating uniform superpositions. An application of the Hadamard gate H to every qubit in a register $X = |00\ldots0\rangle$ creates an equal superposition of all the possible values in this register. If X consists of k qubits, then this works as follows: $\mathsf{H}^k X = \frac{1}{2^{k/2}} \sum_x |x\rangle$, where this summation notation means that x's are interpreted as written in binary and they run from $x = 0$ through $x = 2^k - 1$.

Control by a qubit. Any operation U we know how to implement can be extended to one controlled by an additional qubit. If $|x\rangle$ is the control qubit then the controlled operation acts like U when $|x\rangle = |1\rangle$ and it acts like identity when $|x\rangle = |0\rangle$. For instance, the controlled-not operation, denoted Controlled-Not, flips a qubit subject to the control qubit being $|1\rangle$, otherwise it leaves the qubit intact. Formally, if $|x\rangle = \alpha|0\rangle + \beta|1\rangle$, then the operation Controlled-Not$(|x\rangle, |y\rangle)$ produces $\alpha|0\rangle|y\rangle + \beta|1\rangle|1 \text{ xor } y\rangle$. Given two qubits $|ab\rangle$, we can swap their contents to obtain $|ba\rangle$ by applying Controlled-Not on the registers three times in a sequence: the first Controlled-Not is controlled by the first qubit, the next Controlled-Not by the second qubit, and the third instance of Controlled-Not again by the first qubit. This can be generalized to multi-qubit registers by matching qubits in pairs: for two quantum registers of equal size A and B, the operation Swap(A,B) exchanges the contents of these two registers to obtain BA from AB. The

operation Swap(A,B) controlled by a qubit x is denoted by Controlled-Swap(x, A, B). Formally, if $|x\rangle = \alpha|0\rangle + \beta|1\rangle$, then the operation Controlled-Swap($|x\rangle, A, B$) produces $\alpha|0\rangle AB + \beta|1\rangle BA$. Although cloning of qubits is impossible, entangled copies of registers can be created. Let A and B be quantum registers of equal size, with B initially set to $|00\ldots0\rangle$. An application of Controlled-Not pairwise to the corresponding qubits of A and B creates an entangled copy of each qubit of register A in the corresponding qubit of B. Formally, if $A = \sum_x \alpha_x|x\rangle$, then the operation Entangled-Copy(A, B) produces $\sum_x \alpha_x|xx\rangle$.

Control by a condition. We may condition operations on properties of vectors in the computational base. Let x and y be binary strings of length k. Let $F(x, y)$ be any boolean function on the domain of sequences of $2k$ classical bits, which may be interpreted as defined on vectors $|x\rangle|y\rangle$ in the computational basis. Let A and B be quantum registers of k qubits each, and C be a single qubit register. Let U be a quantum operation on C. The operation U conditioned on F acts like U provided that F is true. More precisely, if $AB = \sum_{x,y} \alpha_{x,y}|xy\rangle$ then Conditional-U($F(A, B), C$) means $\sum_{x,y} \alpha_{x,y}|xy\rangle|$ if $F(x, y)$ then $U(C)\rangle$. A circuit for U conditioned on F can be implemented as follows. For each x and y such that $F(x, y)$ holds true, use a circuit implementing U conditioned on $2k$ qubits being exactly xy, and next compose all these circuits by applying them in sequence one by one. The size of the resulting quantum circuit is $\mathcal{O}(2^{2k}s)$, where s is the size of a circuit implementing U. We will use Conditional-Not conditioned on $F(x, y)$ which is 1 for $x > y$. In our algorithm, the number k will be $\mathcal{O}(\log n)$, so the size of the needed circuit for Conditional-Not is polynomial in n.

Graphs that facilitate communication. Let $G = (V, E)$ be an undirected graph. Let $C \subseteq V$ denote a set of nodes of G. We say that an edge (v, w) of G is *internal for* C if v and w are both in C. We say that an edge (v, w) of G *connects* C_1 *and* C_2, or *is between* C_1 *and* C_2, if one of its ends is in C_1 and the other in C_2, for any disjoint sets of nodes C_1 and C_2. The *subgraph of* G *induced by* C, denoted $G|_C$, is the subgraph of G containing the nodes in C and all the edges internal for C.

Let α, β, δ and ℓ be positive integers and $0 < \varepsilon < 1$ be a real number.

Edge-expansion: graph G is said to satisfy (ℓ, α, β)-*edge-expansion* if, for any two disjoint sets $X, Y \subseteq V$ of *at least* ℓ nodes in each of them, there are at least α edges between X and Y in graph G, and for any two disjoint sets $W, Z \subseteq V$ of *at most* ℓ nodes each, there are at most β edges between W and Z in graph G.

Edge-density: graph G is said to be (ℓ, α, β)-*edge-dense* if, for any set $X \subseteq V$ of *at least* ℓ nodes, there are at least $\alpha|X|$ edges internal for X, and for any set $Y \subseteq V$ of *at most* ℓ nodes, there are at most $\beta|Y|$ edges internal for Y.

Compactness: graph G is said to be $(\ell, \varepsilon, \delta)$-*compact* if, for any set $B \subseteq V$ of at least ℓ nodes, there is a subset $C \subseteq B$ of at least $\varepsilon\ell$ nodes such that each node's degree in $G|_C$ is at least δ. (We call any such set C a *survival set for* B.)

A graph is said to have ℓ-*expanding* property, or to be an ℓ-*expander*, if any two subsets of ℓ nodes each are connected by an edge. Observe that (ℓ, α, β)

-edge-expansion implies the ℓ-expanding property for $\alpha > 0$. In our algorithm we will use ℓ-expanders for $\ell = n/2^i$ and for $\ell = (2/3)^i n/64$, for $i \geq 1$. There exist ℓ-expanders of the maximum degree $\mathcal{O}(\frac{n}{\ell} \log n)$, as can be proved by the probabilistic method. Explicit constructions of such expanders with $\mathcal{O}(\frac{n}{\ell}$ polylog $n)$ bound on node degrees were given by Ta-Shma, Umans, and Zukerman [27].

We use the notation $\lg x$ to denote the binary logarithm $\log_2 x$. For a given n, we define $\delta = 24 \lg n$ and $\gamma = 2 \lg n$. Let $\mathcal{R}(n, y)$ be a random variable whose value is a graph of n nodes such that each pair of nodes is connected by an edge with probability y, independently over all such pairs. Let k be an integer parameter that satisfies $25\delta \leq k \leq 2n/3$. Let $G(k)$ denote $\mathcal{R}(n, y)$ for $y = 24\delta/k$.

Lemma 1 ([9]). *For every n and k such that $25\delta \leq k \leq \frac{2n}{3}$, a random graph $G(k)$ whp is $(k/64, \delta/8, \delta/4)$-edge-dense, $(k/64)$-expanding, $(k, 3/4, \delta)$-compact, and the degree of each node is between $22\frac{n}{k}\delta$ and $26\frac{n}{k}\delta$.*

The notation $N_G^i(W)$ is used to denote the set of all the nodes in V that are of distance at most i in graph G from some node in W, with the neighborhood $N_G^1(v)$ of v denoted as $N_G(v)$. We define dense neighborhoods as follows, where γ and δ are two integer parameters: For a node $v \in V$, a set $S \subseteq N_G^\gamma(v)$ is said to be a (γ, δ)-*dense-neighborhood for* v when each node in $S \cap N_G^{\gamma-1}(v)$ has at least δ neighbors in S.

Lemma 2 ([9]). *If a graph $G = (V, E)$ of n nodes is $(k, 3/4, \delta)$-compact, $(k/64, \delta/8, \delta/4)$-edge-dense and $(k/64)$-expanding, then for any node v, any (γ, δ)-dense-neighborhood for a node $v \in V$ has at least $k/64$ nodes, for $\gamma \geq 2 \lg n$. Moreover, for any two nodes v, w, any (γ, δ)-dense-neighborhood for v is connected by an edge with any (γ, δ)-dense-neighborhood for w, for any $\gamma \geq 2 \lg n$.*

3 Randomized Algorithm

A pseudo-code of algorithm RandomizedConsensus is given in Figure 1. The algorithm is an adaptation of a deterministic algorithm given by Chlebus, Kowalski, and Strojnowski [9]. The deterministic algorithm has to work in time $\Omega(t)$. In contrast to that, we show how to trade worst-case performance for expected time that is polylogarithmic in n. The pattern of communication used by the algorithm is completely deterministic, in the sense that no random bits nor qubits are involved in choosing when and where to send a message.

A process $\mathtt{p}$ uses the following private variables: binary $\mathtt{candidate_p}$, boolean $\mathtt{convinced_p}$, and an integer variable $\mathtt{ticket_p}$. Process $\mathtt{p}$ stores its *candidate value* at variable $\mathtt{candidate_p}$. We want all the processes to decide on one among these initial values. The variable $\mathtt{convinced_p}$ is initialized to `true`. A process $\mathtt{p}$ stays *convinced* if the variable $\mathtt{convinced_p}$ is never set to `false` in Stage 3.

We use networks G modeled as sparse graphs that were also employed in [9]. The graph G for n processes is $(2n/3, 3/4, \delta)$-compact, $(n/96)$-expanding and

S1. initialize $\texttt{candidate}_\texttt{p}$ to the input value;
S2. assign a number selected uniformly at random in the range 1 through n^3 to $\texttt{ticket}_\texttt{p}$ independently from the other processes
S3. for $5 \lg n$ rounds, in each round perform:
  if convinced then
  – send the pair $(\texttt{ticket}_\texttt{p}, \texttt{candidate}_\texttt{p})$ to every neighbor $\texttt{q}$ in G
  – if less than δ messages were received then set $\texttt{convinced}_p \leftarrow \texttt{false}$
  – for each received message of the form $(\texttt{ticket}_\texttt{m}, \texttt{candidate}_\texttt{m})$ do
    if $\texttt{ticket}_\texttt{m} > \texttt{ticket}_\texttt{p}$ then
      set $(\texttt{ticket}_\texttt{p}, \texttt{candidate}_\texttt{p}) \leftarrow (\texttt{ticket}_\texttt{m}, \texttt{candidate}_\texttt{m})$
S4. if not convinced then inquire about candidate values using graphs $G_1, G_2, \ldots, G_{\lg n}$
S5. decide on $\texttt{candidate}_\texttt{p}$

Fig. 1. Algorithm RandomizedConsensus, a pseudo-code for process $\texttt{p}$

$(n/96, \delta/8, \delta/4)$-edge-dense, while each node has degrees between 33δ and 39δ, for $\delta = 24 \lg n$. Such a graph exists by Lemma 1 for $k = 2n/3$; its edges represent communication channels. Graph G_i is $n/2^i$-expanding of $\mathcal{O}(2^i \log n)$ maximum degree, for $1 \leq i \leq \lg n$; its edges represent classical channels.

Stage 4 takes $2 \lg n$ rounds during which the processes act as follows: A process $\texttt{p}$ that is still unconvinced at the start of this stage sends a request message to all its neighbors in graph G_1 in the first round of the stage. A convinced process receiving a request sends back a message. If $\texttt{p}$ receives at least one response, then $\texttt{p}$ sets $\texttt{candidate}_\texttt{p}$ to the maximum value among values $\texttt{candidate}_\texttt{q}$ received and sets $\texttt{convinced}_\texttt{p} \leftarrow \texttt{true}$. If no answer has been received, then $\texttt{p}$ sends request messages to all its neighbors in graph G_2 in the third round and collects responses in the forth one. In general, if a process $\texttt{p}$ is unconvinced in round $2i - 1$ of the stage, for $1 \leq i \leq \lg n$, then $\texttt{p}$ sends a request message to all its neighbors in graph G_i and collects responses, if any, in round $2i$. Responses in round $2i$ are sent only to the processes that have sent requests. If $\texttt{p}$ receives at least one response, then $\texttt{p}$ sets $\texttt{candidate}_\texttt{p}$ to the maximum value among $\texttt{candidate}_\texttt{q}$ values received and sets $\texttt{convinced}_\texttt{p} \leftarrow \texttt{true}$ to become convinced.

Consider an execution of Stage 3 of algorithm RandomizedConsensus. Let B_2 be a set that consist of processes that have not crashed by the beginning of Stage 3 and B_3 a set of processes that have not crashed by the end of Stage 3. Lemma 3 follows from observations made in [9] regarding deterministic ways of scheduling communication adopted to algorithm RandomizedConsensus.

Lemma 3. *The following properties hold for any possible timing of crashes of the nodes in $B_2 \setminus B_3$: If there is a (γ, δ)-dense-neighborhood for $p \in B_3$ in graph $G|_{B_3}$, then process p becomes convinced, unless it crashes in Stage 3. If there is no (γ, δ)-dense-neighborhood for $p \in B_2$ in graph $G|_{B_2}$, then process p does not become convinced. Any process in a survival set C for B_3 becomes convinced, unless it crashes in Stage 3.*

Lemma 4. *Algorithm* RandomizedConsensus *solves consensus with* $\frac{1}{2}$*-agreement against the content-oblivious adversary when* $t < n/3$.

Proof. Termination and validity follow by inspection of the pseudo-code. Consider the crashes that occur by the end of Stage 3. There are less than $n/3$ of them, by the assumption. By the compactness property of G, as formulated in Lemma 1, there exists a survival set C for B_2 of at least $3k/4 = \frac{n}{2}$ elements and with each node in $G|_C$ of degree at least δ. Each node in C receives at least δ messages per round of Stage 3, and so each node in C stays convinced through the end of Stage 3.

Now consider what happens during Stage 3: Each time a message with a higher ticket number than that of p is received, the obtained ticket and the associated candidate value replace the original ticket along with the candidate value. Then this ticket and candidate value are propagated by process p to its neighbors, as long as Stage 3 is not over yet. The adversary may fail at most $n/3$ processes from C, thus at least $n/6$ of them will be convinced by the beginning of Stage 4. One may observe that no processes with distinct candidate values stay convinced throughout Stage 3. To see why this is the case, suppose there were two processes v and w in B_3 that stay convinced. By Lemma 3, each one of these two nodes v and w has its (γ, δ)-dense-neighborhood in $G|_{B_3}$. By Lemma 2, these two dense neighborhoods are connected by an edge in $G|_{B_3}$. Combine this with the inequality $2\gamma+1 \leq 5 \log n$, where $5 \lg n$ is the number of rounds of Stage 3, to see that the maximum of the two values would reach the other process and replace the smaller value within Stage 3.

If there remain some unconvinced processes by the end of Stage 3, then they become convinced during Stage 4, by an argument similar to one about algorithm LPC_1 in [9]. If a process p with the highest ticket number is unique and it belongs to the set C then Stage 3 is concluded with all the processes in C having adopted both the ticket and the candidate value of p. The probability that the process with the greatest ticket number is in C equals at least the ratio of $|C|$ to the number n of all the processes, which is at least $1/2$. □

Theorem 1. *Algorithm* RandomizedConsensus *solves consensus with* $\frac{1}{2}$*-agreement in time* $\mathcal{O}(\log n)$ *while sending* $\mathcal{O}(n \log^3 n)$ *bits against the content-oblivious adversary, if only* $t < n/3$.

Proof. The property of solving consensus is given in Lemma 4. The time performance follows directly from the design of the stages. Messages are sent/received only in Stages 3 and 4. In each round of Stage 3, each non-faulty process sends $\mathcal{O}(\log n)$ messages, with a message carrying $\mathcal{O}(\log n)$ bits. It follows that a total of $\mathcal{O}(n \log^3 n)$ bits are sent. The number of bits sent in Stage 4 is $\mathcal{O}(n \log^2 n)$ by the analysis of a corresponding part of algorithm LPC_1 given in [9]. □

Algorithm RandomizedConsensus can be iterated: a consecutive iteration uses the candidate values from the end of the previous iteration, so that once agreement is reached, it will be maintained in future iterations. As the error is one-sided and iterations are independent, the probabilities of multiple errors multiply. Iterating algorithm RandomizedConsensus k times gives the following:

Corollary 1. *For any integer $k > 0$, there is a randomized algorithm that solves consensus with 2^{-k}-agreement in time $\mathcal{O}(k \log n)$ while sending $\mathcal{O}(kn \log^3 n)$ bits against the content-oblivious adversary, if only $t < n/3$.*

4 Quantum Algorithm

In this section we give a quantum algorithm QuantumConsensus solving consensus against crashes controlled by adaptive adversaries. A pseudo-code of the algorithm is given in Figure 2. The algorithm is an adaptation of the randomized algorithm RandomizedConsensus. Except for the same private variables to store bits, a process p uses two private quantum registers: a $3 \lg n$-qubit register $|\texttt{Ticket}_\texttt{p}\rangle$ and a single-qubit register $|\texttt{Vote}_\texttt{p}\rangle$.

The quantum effects used in QuantumConsensus allow to show properties of the algorithm that hold for an adaptive adversary that are analogous to the properties of algorithm RandomizedConsensus with respect to the oblivious adversary. The intuition is that a quantum message carries a superposition of ticket values, so even when the adversary can see the qubits transmitted, this does not reveal any information about the outcome of a future measurement. Next we formalize this argument.

Influential tickets. We say that a group of k qubits is a *ticket* if this group is initialized with an equal superposition of all 2^k possible states in the computational basis. A group of qubits is *influential* if the qubits among them are used in unitary transformations only as either (1) control qubits or (2) qubits determining conditions on which operations are conditioned or (3) in swaps or controlled swaps or conditional swaps with other influential qubits.

Lemma 5. *Consider an execution of an algorithm in which quantum messages carry qubits that belong to influential tickets. Then the adaptive adversary gains nothing from seeing the states of the qubits in these messages, as compared to the content-oblivious adversary.*

Proof. The superpositions of states of influential qubits, until a measurements that collapses the group to a sequence of classical bits, stay intact or are switched with the superpositions of other influential qubits. The measurement of influential qubits at any time gives any possible configuration of the influential bits with equal probability. It follows that the influential qubits do not carry any bias towards a future decision. Therefore if the adversary needs to prevent some of the messages from being dispatched by crashing the senders before transmissions, then this can be decided without seeing the contents of the messages. □

Lemma 6. *Algorithm* QuantumConsensus *solves consensus with $\frac{1}{2}$-agreement against the adaptive adversary, when $t < n/3$.*

Proof. The qubits in registers `Ticket` are influential tickets, therefore Lemma 5 applies. If a message is entangled with some qubits held by the sender, then

S1. initialize $\texttt{candidate}_\texttt{p}$ to the input value;
prepare a one-qubit vote: $|\texttt{Vote}_\texttt{p}\rangle = |\texttt{candidate}_\texttt{p}\rangle$
S2. generate a superposition of all the numbers in the range 1 through n^3:
$|\texttt{Ticket}_\texttt{p}\rangle \leftarrow \mathsf{H}^{3\lg n}|00\ldots0\rangle = \frac{1}{2^{3/2\lg n}} \sum_{a=1}^{n^3} |a\rangle$
S3. for $5\lg n$ rounds, in each round perform:
if convinced then
– apply Entangled-Copy($|\texttt{Ticket}_\texttt{p}\texttt{Vote}_\texttt{p}\rangle, \texttt{M}$) to create messages M entangled with $|\texttt{Ticket}_\texttt{p}\texttt{Vote}_\texttt{p}\rangle$ and next send one such M to each neighbor q in G
– if less than δ messages were received then set $\texttt{convinced}_p \leftarrow \texttt{false}$
– for each received message of the form $|\texttt{Ticket}_\texttt{m}\texttt{Vote}_\texttt{m}\rangle$, set a qubit S to $|0\rangle$ and apply two quantum operations:
• Conditional-Not($\texttt{Ticket}_\texttt{m} > \texttt{Ticket}_\texttt{p}$, S)
• Controlled-Swap(S, $|\texttt{Ticket}_\texttt{m}\texttt{Vote}_\texttt{m}\rangle$, $|\texttt{Ticket}_\texttt{p}\texttt{Vote}_\texttt{p}\rangle$)
S4. measure the private registers $|\texttt{Ticket}_\texttt{p}\rangle$ and $|\texttt{Vote}_\texttt{p}\rangle$;
set $\texttt{candidate}_\texttt{p} \leftarrow \texttt{Vote}_\texttt{p}$ to the outcome of the measurement of $|\texttt{Vote}_\texttt{p}\rangle$
S5. if not convinced then inquire about candidate values using graphs $G_1, G_2, \ldots, G_{\lg n}$
S6. decide on $\texttt{candidate}_\texttt{p}$

Fig. 2. Algorithm QuantumConsensus, a pseudo-code for process p

the states of the sender and the receiver become entangled upon receipt of the message by the receiver. So an execution of the algorithm creates an entangled quantum global state. When a measurement is performed, the collapses to classical states have the effect of instantaneous communication: the quantum Ticket registers collapse to classical bits and the global state of the system is such as if the measurement was performed in the beginning of the execution and then algorithm RandomizedConsensus were executed. Therefore the argument used in Lemma 4 applies. □

Theorem 2. *Algorithm* QuantumConsensus *solves consensus against the adaptive adversary with* $\frac{1}{2}$*-agreement in time* $\mathcal{O}(\log n)$ *and sending* $\mathcal{O}(n \log^3 n)$ *qubits, when* $t < n/3$.

Proof. A proof analogous to that of Theorem 1 applies. □

Algorithm QuantumConsensus has the property that no entanglement is created by its execution, so it can be iterated similarly as the randomized classical version. As the error is one-sided and iterations are independent, the probabilities of multiple errors multiply. Iterating algorithm QuantumConsensus k times gives the following:

Corollary 2. *For any integer* $k >$*, there exists a quantum algorithm that solves consensus against the adaptive adversary with* 2^{-k}*-agreement in time* $\mathcal{O}(k \log n)$ *and using* $\mathcal{O}(kn \log^3 n)$ *qubits, when* $t < n/3$.

S1. initialize $\texttt{candidate}_\texttt{p}$ to the input value
S2. assign a number selected uniformly at random in the range 1 through n^3 to $\texttt{ticket}_\texttt{p}$ independently from the other processes
S3. for $i = 1$ to $\log_{3/2} n + 1$:
$\texttt{level}_\texttt{p} \leftarrow 1$
for $j = 1$ to $\log_{3/2} n$ do:
– if $\texttt{level}_\texttt{p} = j$ then set $\texttt{convinced}_\texttt{p} \leftarrow \texttt{true}$ else set $\texttt{convinced}_\texttt{p} \leftarrow \texttt{false}$
– perform $5 \lg n$ rounds, in each round:
if convinced then
• send the pair $(\texttt{ticket}_\texttt{p}, \texttt{candidate}_\texttt{p})$ to every neighbor $\texttt{q}$ in $G(j)$
• if less than δ messages were received then set $\texttt{convinced}_p \leftarrow \texttt{false}$
• for each received message of the form $(\texttt{ticket}_\texttt{m}, \texttt{candidate}_m)$ do
if $\texttt{ticket}_\texttt{m} > \texttt{ticket}_\texttt{p}$ then
set $(\texttt{ticket}_\texttt{p}, \texttt{candidate}_\texttt{p}) \leftarrow (\texttt{ticket}_\texttt{m}, \texttt{candidate}_\texttt{m})$
else send back $(\texttt{ticket}_\texttt{p}, \texttt{candidate}_\texttt{p})$ to each process m from which a message was received
– if (not convinced) and $(\texttt{level}_\texttt{p} = j)$ then $\texttt{level}_\texttt{p} \leftarrow \texttt{level}_\texttt{p} + 1$
S4. decide on $\texttt{candidate}_\texttt{p}$

Fig. 3. Algorithm ExtendedRandomizedConsensus, a pseudo-code for process p

5 Randomized and Quantum Algorithms Extended

In this section we first show how to achieve time and communication performance comparable to these of RandomizedConsensus in the case of an arbitrary bound $t < n$ on the number of crashes. A pseudo-code of algorithm ExtendedRandomizedConsensus is given in Figure 3. The pattern of communication used by the algorithm is deterministic, in the sense that no random bits nor qubits are involved in choosing when and where to send a message. The private variables given in the pseudo-code are used similarly as in RandomizedConsensus unless indicated otherwise.

Instead of one graph G in algorithm RandomizedConsensus, we use $\log_{3/2} n$ different graphs $G(j)$, for $1 \leq j \leq \log_{3/2} n$, where graph $G(j)$ has the properties listed in Lemma 1 for $k = (2/3)^j n$.

Stages 1, 2 and 4 are the same as in RandomizedConsensus. Stage 3 repeats $(\lg_{3/2} n + 1) \cdot \lg_{3/2} n$ times Stage 3 from RandomizedConsensus, for different graphs $G(j)$. A process participates only in the iterations corresponding to its current level. If during participation in an iteration a process becomes non-convinced, then at the end of this iteration it increases its level by one. Note that the level is bounded by $\lg_{3/2} n$. Consider an execution of Stage 3 of algorithm ExtendedRandomizedConsensus. For a given iteration of the inner for-loop, we define the sets B_2 and B_3 as follows: B_2 is a set that consists of processes that have not crashed by the beginning of the considered iteration in Stage 3 and B_3 is a set of processes that have not crashed by the end of the considered

iteration in Stage 3. In the analysis of the algorithm, we explore communication properties described in Lemma 3 with respect to graph $G(\ell)$ and sets B_2 and B_3 for an iteration of the inner for-loop of Stage 3 for parameter j set to ℓ.

A proof of the following fact about the correctness is omitted:

Lemma 7. *Algorithm* ExtendedRandomizedConsensus *solves consensus with* $\frac{1}{2}$*-agreement against the content-oblivious adversary for any* $t < n$.

Theorem 3. *Algorithm* ExtendedRandomizedConsensus *solves consensus with* $\frac{1}{2}$*-agreement in time* $\mathcal{O}(\log^3 n)$ *and sending* $\mathcal{O}(n \log^5 n)$ *bits against the content-oblivious adversary, for any* $t < n$.

Proof. The correctness is guaranteed by Lemma 7. The time performance follows from the specification of the algorithm. We analyze the message complexity next. Messages are sent only in Stage 3. Observe that the number of responses is upper bounded by the number of inquiring messages. The key property is that in each iteration of the inner for-loop in Stage 3 for variable j equal to ℓ, for some $1 \leq \ell \leq \lg_{3/2} n$, the number of processes that start that iteration as convinced is at most $n/(3/2)^{\ell-1}$. This is clear for $\ell = 1$, as the number of all the processes is at most n. Suppose that for some $\ell > 1$ the key property does not hold in some iteration of the inner for-loop for $j = \ell$. Consider the preceding iteration of the inner loop for j equal to $\ell - 1 \geq 1$. By the assumption, at least $n/(3/2)^{\ell}$ participating processes became non-convinced during this iteration and increased their level to ℓ at the end of this iteration. This gives a contradiction since, by Lemma 3 applied to graph $G(\ell - 1)$ used in the considered iteration, at least $(3/4) \cdot n/(3/2)^{\ell}$ of these processes would be in a survival set and thus stay convinced until the end of the iteration and do not change the level. The message complexity of an iteration of the inner for-loop for any parameter j is at most $\mathcal{O}(n \log^2 n)$, since the maximum degree of graph $G(j)$ is $\mathcal{O}((3/2)^j \log n)$, by Lemma 1, and there are $\mathcal{O}(\log n)$ rounds in each iteration. The total number of iterations is $(\lg_{3/2} n + 1) \cdot \lg_{3/2} n$ and each message carries $\mathcal{O}(\log n)$ bits. □

Algorithm ExtendedRandomizedConsensus can be iterated: a consecutive iteration uses the candidate values from the end of the previous iteration, so that once agreement is reached, it is maintained in the following iterations. As the error is one-sided and iterations are independent, the probabilities of multiple errors multiply. Iterating algorithm ExtendedRandomizedConsensus k times gives the following:

Corollary 3. *For any integer* $k > 0$*, there exists a randomized algorithm that solves consensus with* 2^{-k}*-agreement in time* $\mathcal{O}(k \log^3 n)$ *while sending* $\mathcal{O}(kn \log^5 n)$ *bits against the content-oblivious adversary, for any* $t < n$.

Now we transform the extended randomized solution to a quantum algorithm called ExtendedQuantumConsensus. The approach is similar to the one applied in Section 4; we omit pseudo-code due to space limits. In the first stage we initialize a qubit for a vote. The second stage sets an equal superposition of all

the vectors in the computational basis. The third stage implements controlled swaps based on conditional negation conditioned on comparisons among the tickets. Finally the qubits are measured and a decision is made on the candidate values. The performance of the algorithm is summarized in the next fact:

Theorem 4. *Algorithm* ExtendedQuantumConsensus *solves consensus with* $\frac{1}{2}$*-agreement in time* $\mathcal{O}(\log^3 n)$ *and sending* $\mathcal{O}(n \log^5 n)$ *qubits against the adaptive adversary, for any* $t < n$.

Iterating the algorithm a suitable number of times we obtain an algorithm that gives the following fact:

Corollary 4. *For any positive integer* k*, there exists a quantum algorithm that solves consensus with* 2^{-k}*-agreement in time* $\mathcal{O}(k \log^3 n)$ *while sending* $\mathcal{O}(kn \log^5 n)$ *bits against the adaptive adversary, for any* $t < n$.

References

1. Aharonov, D., Ben-Or, M.: Fault-tolerant quantum computation with constant error rate. SIAM Journal on Computing 38(4), 1207–1282 (2008)
2. Bar-Joseph, Z., Ben-Or, M.: A tight lower bound for randomized synchronous consensus. In: Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC), pp. 193–199 (1998)
3. Ben-Or, M., Hassidim, A.: Fast quantum Byzantine agreement. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), pp. 481–485 (2005)
4. Bennett, C.H., Wiesner, S.J.: Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. Physical Review Letters 69(20), 2881–2884 (1992)
5. Broadbent, A., Tapp, A.: Can quantum mechanics help distributed computing? SIGACT News 39(3), 67–76 (2008)
6. Buhrman, H., Röhrig, H.: Distributed quantum computing. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 1–20. Springer, Heidelberg (2003)
7. Chlebus, B.S., Kowalski, D.R.: Time and communication efficient consensus for crash failures. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 314–328. Springer, Heidelberg (2006)
8. Chlebus, B.S., Kowalski, D.R.: Locally scalable randomized consensus for synchronous crash failures. In: Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 290–299 (2009)
9. Chlebus, B.S., Kowalski, D.R., Strojnowski, M.: Fast scalable deterministic consensus for crash failures. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC), pp. 111–120 (2009)
10. Chor, B., Merritt, M., Shmoys, D.B.: Simple constant-time consensus protocols in realistic failure models. Journal of the ACM 36(3), 591–614 (1989)
11. Cleve, R., Buhrman, H.: Substituting quantum entanglement for communication. Physical Review A 56(2), 1201–1204 (1997)
12. Cleve, R., van Dam, W., Nielsen, M., Tapp, A.: Quantum entanglement and the communication complexity of the inner product function. In: Williams, C.P. (ed.) QCQC 1998. LNCS, vol. 1509, pp. 61–74. Springer, Heidelberg (1999)

13. de Wolf, R.: Quantum communication and complexity. Theoretical Computer Science 287(1), 337–353 (2002)
14. Denchev, V.S., Pandurangan, G.: Distributed quantum computing: a new frontier in distributed systems or science fiction? SIGACT News 39(3), 77–95 (2008)
15. Dolev, D., Reischuk, R.: Bounds on information exchange for Byzantine agreement. Journal of the ACM 32(1), 191–204 (1985)
16. Gavoille, C., Kosowski, A., Markiewicz, M.: What can be observed locally? Round-based models for quantum distributed computing. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 243–257. Springer, Heidelberg (2009)
17. Gilbert, S., Kowalski, D.R.: Distributed agreement with optimal communication complexity. In: Proceedings of the 21st ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 965–977 (2010)
18. Hadzilacos, V., Halpern, J.Y.: Message-optimal protocols for Byzantine agreement. Mathematical Systems Theory 26(1), 41–102 (1993)
19. Holevo, A.S.: Bounds for the quantity of information transmitted by a quantum communication channel. Problems of Information Transmission 9(3), 177–183 (1973)
20. Holtby, D., Kapron, B.M., King, V.: Lower bound for scalable Byzantine agreement. Distributed Computing 21(4), 239–248 (2008)
21. King, V., Saia, J.: From almost everywhere to everywhere: Byzantine agreement with $\tilde{O}(n^{3/2})$ bits. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 464–478. Springer, Heidelberg (2009)
22. King, V., Saia, J., Sanwalani, V., Vee, E.: Towards secure and scalable computation in peer-to-peer networks. In: Proceedings of the 47th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 87–98 (2006)
23. Kobayashi, H., Matsumoto, K., Tani, S.: Fast exact quantum leader election on anonymous rings. In: Proceedings of the 8th Asian Conference on Quantum Information Science (AQIS), pp. 157–158 (2008)
24. Linial, N.: Distributive graph algorithms - global solutions from local data. In: Proceedings of the 28th Symposium on Foundations of Computer Science (FOCS), pp. 331–335 (1987)
25. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
26. Ta-Shma, A.: Classical versus quantum communication complexity. SIGACT News 30(3), 25–34 (1999)
27. Ta-Shma, A., Umans, C., Zuckerman, D.: Lossless condensers, unbalanced expanders, and extractors. Combinatorica 27(2), 213–240 (2007)

How Much Memory Is Needed for Leader Election

Emanuele G. Fusco[1,⋆] and Andrzej Pelc[2,⋆⋆]

[1] Computer Science Department, Sapienza, University of Rome, 00198 Rome, Italy
fusco@di.uniroma1.it
[2] Département d'informatique, Université du Québec en Outaouais,
Gatineau, Québec J8X 3X7, Canada
pelc@uqo.ca

Abstract. We study the minimum memory size with which nodes of a network have to be equipped, in order to solve deterministically the leader election problem. Nodes are unlabeled, but ports at each node have arbitrary fixed labelings which, together with the topology of the network, can create asymmetries to be exploited in leader election. We consider two versions of the leader election problem: strong LE in which exactly one leader has to be elected, if this is possible, while all nodes must terminate in a state "infeasible" otherwise, and weak LE, which differs from strong LE in that no requirement on the behavior of nodes is imposed, if leader election is impossible. Nodes are modeled as identical automata and we ask what is the minimum amount of memory of such an automaton to enable leader election.

We show that logarithmic memory is optimal for leader election in the class of arbitrary connected graphs. Weak LE can be achieved with $O(\log n)$ bits of memory for all connected graphs with at most n nodes and strong LE can be achieved with $O(\log n)$ bits of memory for all connected graphs with exactly n nodes (none of these assumptions can be entirely removed). On the other hand, we show that $\Omega(\log n)$ bits of memory are necessary to enable leader election even for the class of rings. By contrast we show that strong LE can be accomplished in the class of trees of maximum degree Δ using only $O(\log \log \Delta)$ bits of memory, without any additional information. This proves an exponential gap in memory requirements for leader election between the class of trees and the class of arbitrary graphs. Moreover, we prove that no automaton can solve the leader election problem for all trees, even in the weak form.

1 Introduction

Leader election is a well-known problem in distributed computing, first posed in [25]. Every node of a network has a boolean variable initialized to 0, and after

⋆ This work was done during the visit of Emanuele G. Fusco at the Research Chair in Distributed Computing of the Université du Québec en Outaouais.
⋆⋆ Partially supported by NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 251–266, 2010.

the election, exactly one node, called the *leader*, should change this value to 1. If nodes of the network have distinct labels, then leader election is always possible (e.g., the node with the largest label can become a leader). However, nodes may refrain from revealing their identities, e.g., for security reasons. Hence it is desirable to have leader election algorithms that do not rely on node identities but exploit asymmetries of the network due to its topology and to port labelings.

The amount of memory needed to solve distributed and network problems is a topic extensively studied in the literature, for such tasks as, e.g., network exploration [6,13,20,30], routing [11,12] and rendezvous [14,15,24]. Somewhat surprisingly, leader election has not been studied in this context. In this paper we study the minimum memory size with which nodes of a network have to be equipped for solving deterministically the leader election problem.

A network is modeled as an undirected connected graph. We assume that nodes are unlabeled, but ports at each node have arbitrary fixed labelings $0, \ldots, d-1$, where d is the degree of the node. We do not assume any coherence between port labelings at various nodes. Nodes are modeled as identical automata that communicate with each other along links of the network, and we ask what is the minimum amount of memory of such an automaton to enable deterministic leader election. We consider two versions of the leader election problem: *strong LE* and *weak LE*. In strong LE one leader has to be elected whenever this is possible, while all nodes must terminate in a state "unfeasible" when leader election is impossible. Weak LE differs from strong LE in that no requirement on the behavior of nodes is imposed, if leader election is impossible.

Our results. We show that logarithmic memory is optimal for leader election in the class of arbitrary connected graphs. Weak LE can be achieved with $O(\log n)$ bits of memory for all connected graphs with at most n nodes and strong LE can be achieved with $O(\log n)$ bits of memory for all connected graphs with exactly n nodes (none of these assumptions can be entirely removed). On the other hand, we show that $\Omega(\log n)$ bits of memory are necessary to enable leader election even for the class of rings. By contrast we show that strong LE can be accomplished in trees of maximum degree Δ using only $O(\log \log \Delta)$ bits of memory, without any additional information. This proves an exponential gap in memory requirements for leader election between the class of trees and the class of arbitrary graphs. Moreover, we prove that no automaton can solve the leader election problem for all trees, even in the weak form.

Related work. Leader election was first studied for rings, under the assumption that all labels are distinct. A synchronous algorithm, based on comparisons of labels, and using $O(n \log n)$ messages was given in [18]. It was proved in [16] that this complexity is optimal for comparison-based algorithms. On the other hand, the authors showed an algorithm using a linear number of messages but requiring very large running time. An asynchronous algorithm using $O(n \log n)$ messages was given, e.g., in [29] and the optimality of this message complexity was shown in [4]. Leader election in radio networks has been studied, e.g., in [19,21,27] and randomized leader election, e.g., in [32].

Many authors [1,2,3,8,22,23,31,33,35] studied various computing problems in anonymous networks. In particular, [5,35] characterize networks in which leader election can be achieved when nodes are anonymous. In [34] the authors study the problem of leader election in general networks, under the assumption that labels are not unique. They characterize networks in which this can be done and give an algorithm which performs election when it is feasible. They assume that the number of nodes of the network is known to all nodes and do not attempt to minimize the number of messages. In [10] the authors study feasibility and message complexity of sorting and leader election in rings with nonunique labels, while in [9] the authors provide algorithms for the generalized leader election problem in rings with arbitrary labels, unknown (and arbitrary) size of the ring and for both synchronous and asynchronous communication. In [17] the leader election problem is approached in a model based on mobile agents.

Memory size needed for tree canonizaton, a task related to symmetry breaking in tree networks, has been investigated in [26]. The author shows a centralized algorithm, deciding whether two directed trees are isomorphic or not, that works in logarithmic space. To the best of our knowledge, the present paper is the first study of the memory size of nodes required for leader election in arbitrary networks.

The model. The network is modeled as an undirected connected graph whose nodes are unlabeled, but ports at each node v have arbitrary fixed labelings $0, \ldots, d(v) - 1$, where $d(v)$ is the degree of the node. Unless otherwise specified, we will use the term "graph" to mean a graph with the above properties. Each node is a copy of the same input/output automaton $\mathcal{A}$ which is a quadruple $(\mathcal{S}, Q, \pi, \lambda)$, where $\mathcal{S}$ is a finite set of states, Q is the input/output alphabet, $\pi : \mathcal{S} \times Q \longrightarrow \mathcal{S}$ is the state transition function, and $\lambda : \mathcal{S} \longrightarrow Q$ is the output function. The alphabet Q is the set of finite sequences each of whose terms is a finite binary sequence called a *message*. All nodes start in the same state S_0, called the *initial state*. Computations in the network are organized in asynchronous phases. Consider a node v that is in state S at the beginning of some phase. Let $\lambda(S) = (m_0, m_1, \ldots, m_n)$ be the sequence of messages corresponding to state S (some of the messages m_i may be equal to the *empty message* θ: this is a reserved string of bits. Node v sends message m_i on port i, for all ports $i = 0, \ldots, d(v) - 1$. (All messages m_i, for $i \geq d(v)$ are ignored.) After sending, node v waits until a message m'_i (possibly empty) arrives on each port $i = 0, \ldots, d(v) - 1$. The sequence $\sigma = (m'_0, \ldots, m'_{d(v)-1})$ becomes an input symbol, under the influence of which node v transits to state $S' = \pi(S, \sigma)$. This ends the current phase for node v and at the beginning of the next phase node v is in state S'. There are three pairwise disjoint sets of states included in $\mathcal{S}$: the sets $\mathcal{L}$, $\mathcal{N}$ and $\mathcal{U}$. States in $\mathcal{L}$ are called *leader* states, states in $\mathcal{N}$ are called *non-leader* states, and states in $\mathcal{U}$ are called *infeasible* states. Once a node enters a state in one of these sets, it remains forever in the set (although it may change states). More formally, for any $\sigma \in Q$ and any states $S' \in \mathcal{L}$, $S'' \in \mathcal{N}$ and $S''' \in \mathcal{U}$, we have $\pi(S', \sigma) \in \mathcal{L}$, $\pi(S'', \sigma) \in \mathcal{N}$ and $\pi(S''', \sigma) \in \mathcal{U}$. For a given network, the task

of *leader election* (LE) consists in the following: all nodes except one eventually enter a *non-leader* state, and one node eventually enters a *leader* state.

We consider two versions of the leader election task for a class $\mathcal{C}$ of graphs :

• Weak LE. Let G be any graph in class $\mathcal{C}$. If leader election is possible for the graph G, then all nodes except one eventually enter a *non-leader* state, and one node eventually enters a *leader* state.
• Strong LE. Let G be any graph in class $\mathcal{C}$. If leader election is possible for the graph G, then all nodes except one eventually enter a *non-leader* state, and one node eventually enters a *leader* state. If leader election is impossible for the graph G, then all nodes eventually enter an *infeasible* state.

Hence weak LE differs from strong LE in that, in the case of impossibility of leader election, no restriction on the behavior of nodes is imposed: they can stop in an arbitrary state, or arbitrarily circulate through the set of states. In Section 2 we explain precisely when leader election is possible.

We say that an automaton $\mathcal{A}$ solves weak (resp. strong) LE in the class $\mathcal{C}$ of graphs, if the respective task can be carried out for every graph of the class $\mathcal{C}$ in which copies of $\mathcal{A}$ are placed in every node, under any adversarial scheduling of message deliveries.

We seek automata with small memory, measured by the number of states, or equivalently by the number of bits on which these states can be encoded. An automaton with K states requires $\Theta(\log K)$ bits of memory.

Due to lack of space, detailed descriptions of some algorithms and proofs of several results are omitted.

2 Preliminaries

We will use the following notion from [35]. Let G be a graph and v a node of G. The *view* from v is the infinite rooted tree $\mathcal{V}(v)$ with labeled ports, defined recursively as follows. $\mathcal{V}(v)$ has the root x_0 corresponding to v. For every node v_i, $i = 1, \ldots, k$, adjacent to v, there is a neighbor x_i in $\mathcal{V}(v)$ such that the port number at v corresponding to edge $\{v, v_i\}$ is the same as the port number at x_0 corresponding to edge $\{x_0, x_i\}$, and the port number at v_i corresponding to edge $\{v, v_i\}$ is the same as the port number at x_i corresponding to edge $\{x_0, x_i\}$. Node x_i, for $i = 1, \ldots, k$, is now the root of the view from v_i. By $\mathcal{V}^t(v)$ we denote the view $\mathcal{V}(v)$ truncated to depth t. We will use the following propositions directly following from [28,35].

Proposition 1. *For a n-node graph, $\mathcal{V}(u) = \mathcal{V}(v)$, if and only if $\mathcal{V}^{n-1}(u) = \mathcal{V}^{n-1}(v)$.*

Proposition 2. *Let u and v be two nodes in a graph G, such that $\mathcal{V}(u) = \mathcal{V}(v)$. Let $(a_1, \ldots, a_k)$ be a sequence of port numbers in G and let u' and v' be nodes in G, such that a path from u to u' and a path from v to v' correspond to the sequence $(a_1, \ldots, a_k)$. Then $\mathcal{V}(u') = \mathcal{V}(v')$.*

The following proposition expresses the feasibility of leader election in terms of views.

Proposition 3. *Leader election is possible in a graph G, if and only if views of all nodes are different.*

The above proposition establishes the uniqueness of views as a necessary and sufficient condition on the feasibility of leader election. We will show in this paper that if this condition is satisfied, then weak LE can be performed in the class of all graphs with at most n nodes using $O(\log n)$ bits of memory at each node. Observe that some bound on the size of the graphs in which leader election is to be performed is necessary: otherwise even weak LE cannot be done, regardless of memory size. Indeed, it follows from Theorem 2 proved in Section 3 that no automaton can solve even weak LE in the class of all cycles. On the other hand, we will show that strong LE is feasible in all graphs with exactly n nodes using $O(\log n)$ bits of memory at each node. Here we use a strong assumption that the number of nodes is known. Notice that this assumption cannot be weakened even to knowing a linear bound on the size of the graph. Indeed, consider the following well-known example. There are two cycles: a k-node cycle $C = (v_1, \ldots, v_k)$ and a $(2k)$-node cycle $C' = (w_1, \ldots, w_k, w'_1, \ldots, w'_k)$ with the following port labelings. In cycle C, at node v_1 port 0 corresponds to edge $\{v_1, v_2\}$, and at every node v_i, for $i > 1$, port 0 corresponds to edge $\{v_{i-1}, v_i\}$. In cycle C' at nodes w_1 and w'_1 port 0 corresponds to edge $\{w_1, w_2\}$ (resp. $\{w'_1, w'_2\}$) and at every node w_i and w'_i, for $i > 1$, port 0 corresponds to edge $\{w_{i-1}, w_i\}$ (resp. $\{w'_{i-1}, w'_i\}$). It is easy to see that if the adversary schedules synchronous computations, nodes w_i and w'_i in cycle C' will be always in the same state as node v_i in the cycle C (of course the labels are not known to the nodes and are used only for the description). Hence the solvability of strong LE in the class consisting of these two cycles (whose sizes differ only by a factor of 2) is impossible: an automaton providing a solution for C (where leader election is possible) will incorrectly elect two leaders in C' (where leader election is impossible), instead of stopping in an *infeasible* state.

In the sequel we will use the notion of a *Universal Exploration Sequence* (UXS) [20]. Let $(a_1, a_2, \ldots, a_k)$ be a sequence of integers. An *application* of this sequence to a graph G at node u is the sequence of nodes $(u_0, \ldots, u_{k+1})$ obtained as follows: $u_0 = u$, u_1 is the neighbor of u_0 such that the port at u_0 corresponding to edge $\{u_0, u_1\}$ has number 0; for any $1 \leq i \leq k$, u_{i+1} is the neighbor of u_i such that the port number at u_i corresponding to the edge $\{u_i, u_{i+1}\}$ is $(p + a_i)$ mod $d(u_i)$, where p is the port number at u_i corresponding to the edge $\{u_i, u_{i-1}\}$. A sequence $(a_1, a_2, \ldots, a_k)$ whose application to a graph G at any node u contains all nodes of this graph is called a UXS for this graph. A UXS for a class $\mathcal{G}$ of graphs is a UXS for all graphs in this class.

The following important result, based on a reduction from Koucký [20], is due to Reingold [30].

Proposition 4. *[30] For any positive integer n, there exists a UXS $Y(n) = (a_1, a_2, \ldots, a_k)$ for the class $\mathcal{G}_n$ of all graphs with at most n nodes, such that:*
- *k is polynomial in n,*
- *for any $i \leq k$, the integer a_i can be constructed using $O(\log n)$ bits of memory.*

The above result implies that a (usually non-simple) path $(u_0, \ldots, u_{k+1})$ traversing all nodes can be computed (node by node) using $O(\log n)$ bits of memory, for any graph with at most n nodes. Moreover, logarithmic memory suffices to walk back and forth on this path: to walk forward at node u_i, port $(p + a_i)$ mod $d(u_i)$ should be computed when coming by port p, to walk backward, port $(p - a_i)$ mod $d(u_i)$ should be computed.

For a fixed UXS $(a_1, a_2, \ldots, a_k)$ and a node u of a graph G, the *signature* of node u is the sequence $S(u) = ((b_1, b'_1) \ldots, (b_{k+1}, b'_{k+1}))$ of pairs of integers, such that b_i is the port number at node u_{i-1} corresponding to the edge $\{u_i, u_{i-1}\}$, and b'_i is the port number at node u_i corresponding to the edge $\{u_i, u_{i-1}\}$, where $(u_0, \ldots, u_{k+1})$ is the application of the UXS $(a_1, a_2, \ldots, a_k)$ to graph G at node u. Thus $S(u)$ is the sequence of pairs of port numbers corresponding to traversed edges in the walk starting at node u and constructed using the UXS $(a_1, a_2, \ldots, a_k)$.

Using Proposition 4, the following result has been proved in [7].

Proposition 5. *[7] Let $Y(n^2 + n)$ be a UXS for the class $\mathcal{G}_{n^2+n}$ of all graphs with at most n^2+n nodes, satisfying Proposition 4. Let G be a graph with at most n nodes. For every node u of G, let $S(u)$ be the signature of u corresponding to the UXS $Y(n^2+n)$. Then, for any nodes u_1, u_2 of G we have $\mathcal{V}(u_1) \neq \mathcal{V}(u_2)$, if and only if $S(u_1) \neq S(u_2)$.*

3 Leader Election in Arbitrary Graphs

3.1 Weak Leader Election

We first provide an intuitive description of our algorithm that solves weak LE in logarithmic memory. If nodes had large memories they could discover their whole signatures and then exchange them with all other nodes in the network. By Propositions 3 and 5, if leader election is possible, then all signatures must be different. When each node has compared its signature with signatures of all other nodes, a leader can be elected by selecting the node with the smallest signature in lexicographic order.

Since in our case the memory of a node is much too small to store an entire signature, the basic idea of our algorithm is to perform this comparison term by term, so that each node needs to recall the step of the comparison it is performing and two values only. This is made possible by exploiting universal exploration sequences from [30].

Fix n and denote by Y the UXS $Y(n^2+n)$ from Proposition 5. Let $Y[i]$ denote the i-th term of Y. Let G be any graph with at most n nodes. For every node u, let $S(u)$ be the signature of u corresponding to Y. We denote by $S(u)[i]$ the i-th term of the signature $S(u)$. The length of Y is denoted by $|Y|$.

Now fix nodes u and v. Let $(p, p') = S(u)[1]$. Node u compares (p, p') with $S(v)[1]$ (this can be done by an appropriate exchange of messages between u and v). If the two values coincide, node u proceeds to compare the second term of its signature with the second term of the signature of v. The comparison

proceeds for $|Y|$ steps or up to the time when two distinct values are found in the signatures. If $S(v)$ precedes $S(u)$ in lexicographic order, u gives up comparing its signature with other nodes and becomes a non-leader. If $S(u)$ precedes $S(v)$ in lexicographic order, or the two signatures are equal, node u remains a leader candidate, and starts comparing its signature with the signature of the next node in the application of Y at u. Node u becomes the leader, if it did not give up after $|Y|$ signature comparisons. Notice that $n-1$ comparisons are not enough, indeed the UXS can induce loops and thus some comparisons may result in comparing the signature of a node with itself. After $|Y|$ signature comparisons, a node is guaranteed to have compared its signature with those of all other nodes, in view of the properties of Y.

The main problem in comparing signatures in this way is the congestion caused by simultaneous execution of this task in all nodes of the network. Indeed, a node of large degree could simultaneously receive a request from each of its neighbors. Even in a network of constant maximum degree a node that is a crossing point of many paths defined by the UXS could receive up to $\Theta(n)$ simultaneous requests. Handling linearly many requests simultaneously would require exponentially many states.

To solve this problem, we implement a policy that allows each node to handle only one request per phase, by selecting the one with the smallest rank in lexicographic order and annihilating all others. Hence, many requests for getting a given element of a signature will be lost and will have to be sent again. Nevertheless we prove that eventually every node receives a reply to all its requests.

We first define the annihilating policy as a procedure which specifies which message survives when a node receives some input. Notice that this policy is implemented by correctly defining the transition function π of the automaton and does not require any additional state. If a node u receives more than one non-empty message in a phase, it purges its input by calling Procedure `annihilate`. All non-empty messages sent in the execution of the algorithm have at least two fields called field 1 and field 2.

Procedure `annihilate`
Let M be the set of non-empty messages received by node u in a given phase. The message $m \in M$ that survives is the one with the smallest rank in lexicographic order over field 1, field 2, and the incoming port number. Node u considers this unique message (and empty messages on other ports) as its input in the given phase. ■

From now on we assume that a node receives at most one non-empty message in each phase. When no message is specified to be sent by a node on a given port in a given phase, the message sent on that port is the empty message.

Procedure `get(`i, k`)`, whose details are omitted, allows a node u to obtain the k-th term of the signature of the i-th node in the application of Y at u. The procedure is defined for parameters $i \geq 0$ and $k > 0$. In particular, `get(`$0, k$`)` allows a node to obtain the k-th term of its own signature.

Algorithm `Weak Leader Elect`
Each node maintains two counters, k and i, initialized to 1. Counters track the step in the signature comparison process. In phase 0, each node initiates Procedure `get`$(0, 1)$.
• A node u that initiated `get`$(0, k)$ and did not receive yet a reply to its request, reinitiates `get`$(0, k)$ in each phase in which it receives only empty messages.
• When a node u gets a reply in Procedure `get`$(0, k)$, it stores $S(u)[k]$ and initiates `get`(i, k).
• A node that initiated `get`(i, k) and did not receive yet a reply to its request, reinitiates `get`(i, k) in each phase in which it receives only empty messages.
• When a node u gets a reply in Procedure `get`(i, k), it compares the outcome of the procedure, call it o, with $S(u)[k]$. If $o < S(u)[k]$ (in lexicographic order), then node u enters a *non-leader* state and it keeps participating in the algorithm only by acting as a relay node. If $o > S(u)[k]$ (in lexicographic order), then node u increases counter i by 1, sets counter k to 1, and initiates `get`$(0, 1)$. If $o = S(u)[k]$ and $k < |Y|$, then node u increases counter k by 1, and initiates `get`$(0, k)$. If $k = |Y|$, node u increases counter i by 1, sets counter k to 1, and initiates `get`$(0, 1)$.
• A node enters a *leader* state when its counter i reaches value $|Y| + 1$. A node in a *leader* state keeps participating in the algorithm only by acting as a relay node. ■

Lemma 1. *Algorithm* `Weak Leader Elect` *elects a unique leader whenever it is possible.*

Since Algorithm `Weak Leader Elect` requires every node to keep only constantly many counters of size logarithmic in n, we get the following theorem.

Theorem 1. *There exists an automaton with $O(\log n)$ bits of memory that solves weak LE in the class of graphs with at most n nodes.*

We now provide a matching lower bound on the memory size required to solve weak LE, which holds already on the class of rings, and is valid even for synchronous computations.

Theorem 2. *An automaton with $\Omega(\log n)$ bits of memory is required to solve weak LE in the class of rings with at most n nodes.*

3.2 Strong Leader Election

Algorithm `Strong Leader Elect` for solving strong LE in n-nodes networks can be obtained by making each node compare signatures of all other nodes, keeping count of how many distinct ones it finds in the network. As different signatures correspond to different views, and leader election is possible in a network with n nodes, if and only if all nodes have different views, the feasibility of LE in a n-node graph is equivalent to the existence of n distinct signatures. Algorithm `Weak Leader Elect` can then be applied, in solvable instances, in order to elect a unique leader.

Lemma 2. *Algorithm* `Strong Leader Elect` *elects a unique leader whenever it is possible. If leader election is impossible, all nodes enter an* infeasible *state.*

Algorithm `Strong Leader Elect` requires every node to keep only constantly many counters of size logarithmic in n, plus counters needed for executing Algorithm `Weak Leader Elect`. Hence, as a consequence of Theorem 1, we get the following result.

Theorem 3. *There exists an automaton with $O(\log n)$ bits of memory that solves strong LE in the class of graphs with exactly n nodes.*

Notice that the lower bound $\Omega(\log n)$ on the memory size needed to solve weak LE for the class of graphs with at most n nodes (proved in Theorem 2) does not immediately imply a similar lower bound for strong LE for the class of graphs with exactly n nodes. Indeed, the argument used for weak LE relies on rings of different sizes, while in strong LE the automaton is designed for a specific value of n. Nevertheless, a similar lower bound, showing that the automaton from Theorem 3 is also optimal, can be obtained as follows.

Proposition 6. *An automaton with $\Omega(\log n)$ bits of memory is required to solve strong LE in the class of n-nodes rings.*

4 Leader Election in Trees

In this section we provide an upper bound $O(\log\log\Delta)$ on the number of memory bits required to solve strong LE in trees with maximum degree Δ. We also show that no automaton can solve even weak LE in all trees.

A tree T has a *central node*, if and only if, it has even diameter D. The central node in this case is the unique node that is the starting point of (at least) two edge disjoint paths of length $D/2$. If a tree T has odd diameter D, then it has a *central edge*. This unique edge is the central edge of any path of length D in T.

Consider the set $\mathcal{T}_0$ of rooted trees where each node has label 0 at the port leading to its parent. Such a n-node tree can be encoded by a binary string of length $2n-2$. This is done by performing a depth-first visit of the tree, driven by increasing order of port numbers at each node, and writing 1 every time an edge is traversed going down, and 0 every time an edge is traversed going up. A tree $T \in \mathcal{T}_0$ can be reconstructed from its code as follows. Start from the root, making it the current node. In every step of the reconstruction we have a suffix σ of the code and a current node v. In the first step σ is the code. If the first element of σ is a 1, attach a new child to v, label the port connecting v to this child with the smallest port number not yet assigned at node v. Also assign label 0 to the port at the child connecting it to v. The child becomes the current node at the next step. If the first element of σ is a 0, the parent of v becomes the current node at the next step. In both cases, the first element of σ is removed.

A string s of length $2n-2$ belongs to the set $\mathcal{C}_n$ of *well formed codes*, if and only if it has $n-1$ ones, $n-1$ zeroes, and no prefix of s contains more zeroes than ones. The coding and decoding functions described above define a bijection

between the set $\mathcal{C}_n$ and the set of all n-node trees in $\mathcal{T}_0$. This is a subset of the trees we want to handle, as in general the port number leading to the parent of a node v is arbitrarily chosen between 0 and $d(v) - 1$. This difficulty can be overcome, thus defining a bijection between the class of all rooted trees with port numbers and their representations as code strings, as follows. Consider an internal node w. Let i be the port number, at w, of the edge connecting w to its parent. Let u be the child of w such that the edge between w and u has port number $i + 1$ at w. Then add the symbol $\diamond$ into the code before the first visit to u. Denote such an augmented code of a tree T by $B(T)$. From now on it is called the code of T. As symbol $\diamond$ has been added to the code, resulting in a ternary alphabet, we use 2 bits to represent each symbol in code $B(T)$ without ambiguities.

4.1 Strong LE in Trees Using $O(\log \log \Delta)$ Memory Bits

We first give an overview of our algorithm. Algorithm `Tree Leader Elect` solving strong LE in trees has three stages. In stage 1, the tree is pruned, starting from the leaves, until only the central node or the central edge remains. If only the central node remains after pruning, this node can be elected as a leader. Trees whose pruning ends with the central edge admit LE, if and only if the two subtrees rooted at the endpoints of the central edge are not isomorphic[1] (either because of differences in the topology or different port labelings).

Stage 2 determines, for trees with a central edge, whether the two subtrees are isomorphic or not. If not, a unique leader is elected, otherwise LE is impossible. Endpoints of the central edge coordinate the process of comparing the two subtrees, however, the memory of each node is so small that it cannot even save one port number. Hence, no single node can track the progress of the comparison on its own. Information allowing to perform the comparison is thus distributed among many nodes in the subtrees, so that no node needs to store more than constantly many counters, whose values are bounded by $\log \Delta$. The comparison is done by performing a pre-order visit of each subtree; flags are used to identify nodes in the path from the root of the subtree to the currently visited node, i.e., the owner of the token. This owner is responsible for generating the next symbol of the augmented code of the subtree rooted in one of the endpoints of the central edge, and for sending it up to this endpoint. Each of the endpoints of the central edge compares each symbol received from its subtree with the corresponding symbol obtained from the other subtree. If two mismatching symbols are found, the symmetry is broken and a leader is elected, otherwise, each endpoint resumes the visit of its subtree by broadcasting a request for the next symbol to all its neighbors. Among them, exactly one is flagged and will relay the request down the tree towards the current owner of the token.

While moving the token from a parent to its first child is easy, moving the token from a child to the next child, or determining that all children have been

[1] Rooted trees are isomorphic, if there exists a bijection from nodes of one to nodes of the other that maps the root to the root, preserving adjacencies and port numbers.

already visited and thus the token has to be sent back to the parent, is much harder, as port numbers determining the ranks of the children in the pre-order visit are too large to be kept in the memory of any single node.

Procedure `Move Token`, described below, is the fundamental building block of the algorithm. The procedure allows to move the token from a child to the next child to be visited (or back to the parent when all children have been already visited), using only one counter of $O(\log\log\Delta)$ bits at each node.

In stage 3, success or impossibility of LE is broadcast to all nodes, thus allowing non-leader nodes to correctly enter a *non-leader* or an *infeasible* state, according to the outcome of the election.

Procedure `Move Token`

The procedure is initiated by the parent w of the current owner v of the token.

We assume that either w is one of the endpoints of the central edge, or it has at least one neighbor that is marked *ascending* (namely, its parent). This invariant will hold at each call of the procedure in Algorithm `Tree Leader Elect`.

Each node u has a counter $c(u)$ that is set to 0 when the procedure is initiated. Let $p(u)$ be the port number, at w, connecting w to node u. Moreover, let x be the number of bits allowed by the memory size to be assigned to a counter $c(u)$. $O(\log\log\Delta)$ memory bits are enough to guarantee $2^{2^x} \geq \Delta$, and this is all we will need.

The following block of 4 phases is iterated 2^x times, counted by $c(w)$ ranging from 0 to $2^x - 1$. Nodes that are *candidates* to obtain the token are all the neighbors u of w, having $p(u) \geq 1$, that are not marked and do not own the token.

1. In phase 1, node w sends message $< c(w) >$ to all its neighbors.

2. In phase 2, let $k = \sum_{i=1}^{c(w)} s_i 2^{2^x - i}$, where $s_1 = 1$ if $p(v) \geq 2^{2^x - 1}$, $s_1 = 0$ otherwise, and, for $i > 1$, $s_i = 1$ if $p(v) \geq 2^{2^x - i} + \sum_{j=1}^{i-1} s_j 2^{2^x - j}$, and $s_i = 0$ otherwise. Node v sends message $< large >$, if $p(v) - k \geq 2^{2^x - c(w) - 1}$. Otherwise it sends message $< small >$.

3. In phase 3, node w relays the message, either $< large >$ or $< small >$, received from v in phase 2, together with counter $c(w)$. Then it increases counter $c(w)$ by one.

4. In phase 4, for each *candidate* node u, we have $k = \sum_{i=1}^{c(w)} s_i 2^{2^x - i}$, where $s_1 = 1$ if $p(u) - 1 \geq 2^{2^x - 1}$, $s_1 = 0$ otherwise, and, for $i > 1$, $s_i = 1$ if $p(u) - 1 \geq 2^{2^x - i} + \sum_{j=1}^{i-1} s_j 2^{2^x - j}$, and $s_i = 0$ otherwise. If w sent message $< large, c(w) >$ in phase 3, each *candidate* node u stops being a *candidate* if $p(u) - k - 1 < 2^{2^x - c(w) - 1}$. If w sent message $< small, c(w) >$ in phase 3, each *candidate* node u stops being a *candidate* if $p(u) - k - 1 \geq 2^{2^x - c(w) - 1}$.

After 2^x iterations of the above block, the remaining *candidate* node, if it exists, sends message $< success >$ to all its neighbors and obtains the token. The next phase is used by node w to inform its children (in particular node v), if the token has been moved to the next child or not. ■

By means of a binary search, Procedure `Move Token` chooses the node u such that $p(u) = p(v) + 1$ as the new owner of the token, if such a node exists among

children of the node w that initiated the procedure. The procedure can be easily modified to choose the node u such that $p(u) = p(v) + 2$, in the case when w is connected to its parent (i.e., its only neighbor marked *ascending*) through port $p(v) + 1$. This is done by letting a *candidate* node u lose (i.e., stop being a *candidate*), in phase 4, if $p(u) - k - 2 < 2^{2^x - c(w) - 1}$ when a $< large >$ message was sent by its parent, and if $p(u) - k - 2 \geq 2^{2^x - c(w) - 1}$, when a $< small >$ message was sent by its parent. Values s_i, which determine k, are computed in this case as follows. $s_1 = 1$ if $p(u) - 2 \geq 2^{2^x - 1}$, $s_1 = 0$ otherwise, and, for $i > 1$, $s_i = 1$ if $p(u) - 2 \geq 2^{2^x - i} + \sum_{j=1}^{i-1} s_j 2^{2^x - j}$, and $s_i = 0$ otherwise.

Lemma 3. *Let w be the parent of the current owner v of the token. Let j be the port number at w corresponding to edge $\{w, v\}$. Let v' be the child of w such that port $j+1$ at w corresponds to edge $\{w, v'\}$, if such a node exists. Procedure* `Move Token` *moves the token from v to v'.*

Lemma 4. *Algorithm* `Tree Leader Elect` *elects a unique leader in any tree T whenever it is possible. If leader election is impossible in T, then all nodes enter an* infeasible *state.*

Theorem 4. *There exists an automaton with $O(\log \log \Delta)$ bits of memory, that solves strong LE in the class of trees with maximum degree Δ.*

Proof. Stages 1 and 3 of Algorithm `Tree Leader Elect` require an automaton with only constantly many states. As for stage 2, the only task requiring more than constantly many memory bits is moving the token among sibling nodes, by means of calls to Procedure `Move Token`. All values exceeding $\log \Delta$ (e.g., port numbers) that appear in the description of Procedure `Move Token` are handled by the state transition function of the automaton, and never need to be stored in the memory of a node. Indeed, the procedure requires the parent w of the current owner of the token to store only the iteration number (which ranges from 0 to $2^x - 1 \in O(\log \Delta)$) and other nodes to remember only whether they are *candidates* or not. Hence $O(\log \Delta)$ states of the automaton are enough, which implies that $O(\log \log \Delta)$ memory bits are sufficient.

4.2 Impossibility of a Universal Leader Election Automaton for Trees

In this section we prove the following impossibility result which remains valid for strong LE as well.

Theorem 5. *No automaton can solve weak LE in the class of all trees.*

Proof. Assume, for contradiction, that an automaton $\mathcal{A}$ solves weak LE in the class of all trees. Let C be the number of states of automaton $\mathcal{A}$. Consider the following class of trees. A (h, k)-*tree* is a tree on $h + k + 2$ nodes, obtained by connecting the central node of a star with $h + 1$ nodes to the central node of a star with $k + 1$ nodes. Both ports corresponding to the edge connecting the centers are labeled 0. Hence, LE is possible in a (h, k)-tree, if and only if $h \neq k$.

Call v_h the node of degree $h+1$ and v_k the node of degree $k+1$ in a (h,k)-tree. Call *h-leaves* the leaves connected to v_h and *k-leaves* the leaves connected to v_k. $l_{h,p}$ denotes the h-leaf that is connected to node v_h by the edge whose corresponding port number at v_h is p.

Suppose that copies of automaton $\mathcal{A}$ are placed in all nodes of the tree. The *trace* $t_{h,k}(i)$ of node v_h is the sequence of states of v_h in a (h,k)-tree, up to phase i. The j-th term of this sequence, denoted by $t_{h,k}[j]$, for $j \leq i$, is the state of v_h at the beginning of phase j.

The state of a given leaf $l_{h,p}$ at the beginning of a given phase only depends on the trace of the central node v_h to which the leaf is connected, up to the previous phase.

For any automaton with C states, we can describe the transition function of a given h-leaf as a *transition matrix* mapping each pair of states (the state of v_h and the state of the h-leaf) to a new state of the h-leaf. This matrix has C^2 cells; each cell of the matrix contains a value chosen among the C possible states for the h-leaf. Hence at most $X = C^{C^2}$ distinct matrices exist. The value of p determines the transition matrix of the h-leaf $l_{h,p}$. The leaf $l_{h,p}$ is *equivalent* to $l_{h,q}$, denoted by $l_{h,p} \equiv l_{h,q}$, if $l_{h,p}$ and $l_{h,q}$ have the same transition matrix. Notice that two equivalent h-leaves are always in the same state.

In a (h,h)-tree, traces $t_{h,h}(i)$ of the two centers are equal, for any value i. Informally speaking, we can construct a transition function π' whose input consists of the current state of node v_h and messages from pairwise non-equivalent h-leaves. We can then construct an automaton $\mathcal{A}'$ having π' as its transition function. Now consider two (h,h)-trees. Tree T contains copies of automaton $\mathcal{A}$ in all nodes. Tree T' contains copies of automaton $\mathcal{A}'$ in the central nodes and copies of automaton $\mathcal{A}$ in all leaves. The function π' is such that, for all values i, traces $t_{h,h}(i)$ of central nodes in T are equal to traces $t'_{h,h}(i)$ of central nodes in T'.

Formally, let $\lambda^*(S) = m_0$, where $\lambda(S) = (m_0, m_1, \ldots, m_h)$ is the sequence of messages that is the output corresponding to state S in automaton $\mathcal{A}$. $J \subset \mathcal{S} \times Q$ is the set of *legal* inputs defined as follows. $\sigma = (S, m_0, m_1, \ldots, m_h)$ is in J, if and only if $(m_0, m_1, \ldots, m_h) \in Q$, $m_0 = \lambda^*(S)$, and $m_p = m_q$ for each pair (p,q) such that $l_{h,p} \equiv l_{h,q}$. An input of function π for the copy of automaton $\mathcal{A}$ placed in a central node of a (h,h)-tree is always legal. Indeed, the states of the two central nodes are equal in any phase and the states of equivalent h-leaves are equal in any phase. Two nodes in the same state always send the same message on the same port.

Hence we can define the following bijection α, whose domain is the set J of legal inputs and whose set of values J' has cardinality bounded by C^{X+1}. Let $\alpha : J \longrightarrow J'$, $\alpha((S, m_0, m_1, \ldots, m_h)) = (\lambda^*(S), m_{z_1}, \ldots, m_{z_r})$, where all integers z_i are indices of pairwise non-equivalent h-leaves and $r \leq X$. As α is a bijection, we can also define its inverse $\alpha^{-1} : J' \longrightarrow J$.

Once function α has been defined, the transition function π' we are seeking can be constructed as follows. $\pi' : J' \longrightarrow \mathcal{S}$, $\pi'(\alpha(\sigma)) = \pi(\alpha^{-1}(\alpha(\sigma))) = \pi(\sigma)$.

Claim. C^{X+1} is an upper bound on the number of reachable state configurations of nodes in a (h,h)-tree.

In any (h,h)-tree, traces with more than C^{X+1} terms will thus induce a repeated configuration of states of all nodes. As C^{X+1} does not depend on the value h, infinitely many triples (a,b,c), with $a < b < c$, exist, such that $t_{a,a}(i) = t_{b,b}(i) = t_{c,c}(i)$, for any value i. Pick such a triple of integers. We will now prove, by induction on j, that $t_{a,b}(j) = t_{b,a}(j) = t_{a,a}(j)$, for all values j. For $j = 0$, the base of induction holds, as the first term of any trace is the initial state of the automaton $\mathcal{A}$. Assuming $t_{a,b}(j) = t_{b,a}(j) = t_{a,a}(j)$ we have that $t_{a,b}[j+1] = t_{a,a}[j+1]$, since, by the inductive hypothesis, $t_{b,a}[j] = t_{a,a}[j]$ and all a-leaves are in the same state in phase j in an (a,b)-tree as in an (a,a)-tree. Similarly, $t_{b,a}[j+1] = t_{b,b}[j+1]$. As $t_{a,a}(i) = t_{b,b}(i)$ for any value i, we have $t_{a,b}[j+1] = t_{b,a}[j+1] = t_{a,a}[j+1]$, which concludes the proof by induction.

It follows that we can find three values, $a < b < c$, such that, for any assignment $x = a$, $x = b$, or $x = c$ and any assignment $y = a$, $y = b$, or $y = c$, traces of the central nodes in a (x,y)-tree are always equal. Hence, for all (x,y)-trees, any x-leaf has the same state, in a given phase, as the corresponding y-leaf (if it exists). Also the states of nodes v_x and v_y are equal in each phase. Hence, neither an a-leaf nor a b-leaf can become a leader, as this would cause election of two leaders in an (a,c)-tree or in a (b,c)-tree. The same holds for nodes v_a and v_b. On the other hand, weak LE requires election of a leader in an (a,b)-tree, contradiction. Hence, automaton $\mathcal{A}$ does not solve weak LE in the class of all trees, and the proof of the theorem is complete.

The argument from the proof of Theorem 5 shows in fact that an automaton with C states cannot solve weak LE in the class of trees of maximum degree $C^{C^{C^2}+1}$. This implies that the minimum number of memory bits of an automaton solving weak LE for all trees of maximum degree Δ is $\Omega(\log\log\log\Delta)$. In Theorem 4 we showed an automaton with $O(\log\log\Delta)$ bits of memory that solves strong LE in such trees. This yields the following open problem.

Problem 1. *What is the minimum number of memory bits of an automaton that can solve weak (respectively strong) LE in the class of trees of maximum degree Δ?*

In this paper we focused on the size of memory, while most of the literature was concerned with time and message complexity of leader election. This yields the following problem.

Problem 2. *What are the tradeoffs between memory size at nodes and time or message complexity of leader election?*

References

1. Attiya, H., Snir, M., Warmuth, M.: Computing on an Anonymous Ring. Journal of the ACM 35, 845–875 (1988)
2. Attiya, H., Snir, M.: Better Computing on the Anonymous Ring. Journal of Algorithms 12, 204–238 (1991)

3. Boldi, P., Vigna, S.: Computing anonymously with arbitrary knowledge. In: Proc. 18th ACM Symp. on Principles of Distributed Computing, pp. 181–188 (1999)
4. Burns, J.E.: A formal model for message passing systems, Tech. Report TR-91, Computer Science Department, Indiana University, Bloomington (September 1980)
5. Codenotti, B., Gemmell, P., Simon, J.: Symmetry breaking in anonymous networks: characterizations. In: Proc. 4th Israel Symposium on Theory of Computing and Systems ISTCS, pp. 16–26 (1996)
6. Cook, S.A., Rackoff, C.: Space Lower Bounds for Maze Threadability on Restricted Machines. SIAM J. Comput. 9, 636–652 (1980)
7. Czyzowicz, J., Kosowski, A., Pelc, A.: How to meet when you forget: Log-space rendezvous in arbitrary graphs, Proc. 29th ACM Symp. on Principles of Distributed Comp. PODC 2010 (2010)
8. Diks, K., Kranakis, E., Malinowski, A., Pelc, A.: Anonymous wireless rings. Theoretical Computer Science 145, 95–109 (1995)
9. Dobrev, S., Pelc, A.: Leader election in rings with nonunique labels. Fundamenta Informaticae 59, 333–347 (2004)
10. Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F.L., Santoro, N.: Sorting Multisets in Anonymous Rings. In: Proc. of the IEEE International Parallel and Distributed Processing Symposium IPDPS, Cancun, Mexico, pp. 275–280 (2000)
11. Fraigniaud, P., Gavoille, C.: Routing in Trees. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 757–772. Springer, Heidelberg (2001)
12. Fraigniaud, P., Gavoille, C.: A Space Lower Bound for Routing in Trees. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 65–75. Springer, Heidelberg (2002)
13. Fraigniaud, P., Ilcinkas, D.: Digraphs Exploration with Little Memory. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 246–257. Springer, Heidelberg (2004)
14. Fraigniaud, P., Pelc, A.: Deterministic rendezvous in trees with little memory. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 242–256. Springer, Heidelberg (2008)
15. Fraigniaud, P., Pelc, A.: Delays induce an exponential memory gap for rendezvous in trees. In: Proc. 22nd Ann. ACM Symposium on Parallel Algorithms and Architectures, SPAA 2010 (2010)
16. Fredrickson, G.N., Lynch, N.A.: Electing a leader in a synchronous ring. Journal of the ACM 34, 98–115 (1987)
17. Haddar, M.A., Kacem, A.H., Métivier, Y., Mosbah, M., Jmaiel, M.: Electing a leader in the local computation model using mobile agents. In: Proc. 6th ACS/IEEE International Conference on Computer Systems and Applications AICCSA 2008, pp. 473–480 (2008)
18. Hirschberg, D.S., Sinclair, J.B.: Decentralized extrema-finding in circular configurations of processes. Communications of the ACM 23, 627–628 (1980)
19. Jurdzinski, T., Kutylowski, M., Zatopianski, J.: Efficient algorithms for leader election in radio networks. In: Proc., 21st ACM Symp. on Principles of Distr. Comp. (PODC 2002), pp. 51–57 (2002)
20. Kouckỳ, M.: Universal Traversal Sequences with Backtracking. J. Comput. Syst. Sci. 65, 717–726 (2002)
21. Kowalski, D., Pelc, A.: Leader election in ad hoc radio networks: a keen ear helps. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 521–533. Springer, Heidelberg (2009)

22. Kranakis, E.: Symmetry and Computability in Anonymous Networks: A Brief Survey. In: Proc. 3rd Int. Conf. on Structural Information and Communication Complexity, pp. 1–16 (1997)
23. Kranakis, E., Krizanc, D., van der Berg, J.: Computing Boolean Functions on Anonymous Networks. Information and Computation 114, 214–236 (1994)
24. Kranakis, E., Krizanc, D., Morin, P.: Randomized Rendez-Vous with Limited Memory. In: Laber, E.S., Bornstein, C., Nogueira, L.T., Faria, L. (eds.) LATIN 2008. LNCS, vol. 4957, pp. 605–616. Springer, Heidelberg (2008)
25. LeLann, G.: Distributed systems: Towards a formal approach. In: Gilchrist, B. (ed.) Information processing'77, pp. 155–160. North Holland, Amsterdam (1977)
26. Lindell, S.: A logspace algorithm for tree canonization. In: Proc. 24th ACM Symposium on Theory of Computing (STOC 1992), pp. 400–404 (1992)
27. Nakano, K., Olariu, S.: Uniform leader election protocols for radio networks. IEEE Transactions on Parallel Distributed Systems 13, 516–526 (2002)
28. Norris, N.: Universal Covers of Graphs: Isomorphism to Depth $N - 1$ Implies Isomorphism to All Depths. Discrete Applied Mathematics 56(1), 61–74 (1995)
29. Peterson, G.L.: An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. ACM Transactions on Programming Languages and Systems 4, 758–762 (1982)
30. Reingold, O.: Undirected connectivity in log-space. Journal of the ACM 55, 1–24 (2008)
31. Sakamoto, N.: Comparison of Initial Conditions for Distributed Algorithms on Anonymous Networks. In: Proc. 18th ACM Symp. on Principles of Distributed Computing (PODC 1999), pp. 173–179 (1999)
32. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM J. on Computing 15, 468–477 (1986)
33. Yamashita, M., Kameda, T.: Computing on anonymous networks. In: Proc. 7th ACM Symp. on Principles of Distributed Computing (PODC 1988), pp. 117–130 (1988)
34. Yamashita, M., Kameda, T.: Electing a leader when procesor identity numbers are not distinct. In: Bermond, J.-C., Raynal, M. (eds.) WDAG 1989. LNCS, vol. 392, pp. 303–314. Springer, Heidelberg (1989)
35. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part I - characterizing the solvable cases. IEEE Trans. Parallel and Distributed Systems 7, 69–89 (1996)

Leader Election Problem versus Pattern Formation Problem*

Yoann Dieudonné[1], Franck Petit[2], and Vincent Villain[1]

[1] MIS, Université de Picardie Jules Verne Amiens, France
[2] LIP6/Regal, Université Pierre et Marie Curie, INRIA, CNRS, France

Abstract. Leader election and arbitrary pattern formation are fundamental tasks for a set of autonomous mobile robots. The former consists in distinguishing a unique robot, called the leader. The latter aims in arranging the robots in the plane to form any given pattern. The solvability of both these tasks turns out to be necessary in order to achieve more complex tasks.

In this paper, we study the relationship between these two tasks in a model, called *CORDA*, wherein the robots are weak in several aspects. In particular, they are *fully asynchronous* and they have no direct means of communication. They cannot remember any previous observation nor computation performed in any previous step. Such robots are said to be *oblivious*. The robots are also *uniform* and *anonymous*, i.e, they all have the same program using no global parameter (such as an identity) allowing to differentiate any of them. Moreover, none of them share any kind of common coordinate mechanism or common sense of direction, except that they agree on a common handedness (*chirality*).

In such a system, Flochini *et al.* proved in [9] that it is possible to elect a leader for $n \geq 3$ robots if it is possible to form any pattern for $n \geq 3$. In this paper, we show that the converse is true for $n \geq 4$ and thus, we deduce that both problems are equivalent for $n \geq 4$ in *CORDA* provided the robots share the same chirality.

Keywords: Mobile Robot Networks, Pattern Formation, Leader Election.

1 Introduction

Mobile robots working together to perform cooperative tasks in a given environment is an important, open area of research. *Teams* (or, *swarms*) of *mobile robots* provide the ability to measure properties, collect information and act in a given physical environment. Numerous potential applications exist for such multi-robot systems, to name only a very few: environmental monitoring, large-scale construction, risky area surrounding or surveillance, and exploration of awkward environments.

In a given environment, the ability for the swarm of robots to succeed in the accomplishment of the assigned task greatly depends on (1) *global* properties

* This work has been supported by the ANR projet *R-Discover* (08-ANR-CONTINT).

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 267–281, 2010.

assigned to the swarm, and (2) *individual* capabilities each robot has. Examples of such global properties are the ability to distinguish among themselves at least one (or, more) robots (*leader*), to agree on a common global direction (*sense of direction*), or to agree on a common handedness (*chirality*). The individual capacities of a robot are its moving capacities and its sensory organs.

To deal with cost, flexibility, resilience to dysfunction, and autonomy, many problems arise for handling the distributed coordination of swarms of robots in a *deterministic* manner. This issue was first studied in [13,14], mainly motivated by the minimal level of ability the robots are required to have in the accomplishment of basic cooperative tasks. In other words, the feasibility of some given tasks is addressed assuming swarm of autonomous robots either devoid or not of capabilities like (observable) identifiers, direct means of communication, means of storing previous observations, sense of direction, chirality, etc. So far, except the "classical" *Leader Election Problem* [1,6,9,12], most of the studied tasks are geometric problems, so that *Arbitrary Pattern Formation Problem*, *Line Formation*, *Gathering*, and *Circle Formation*—refer to [3,4,5,8,9,10,14] for these problems.

In this paper, we concentrate on two of the aforementioned problems: Leader Election Problem (LEP) and Arbitrary Pattern Formation Problem ($APFP$).

Definition 1 (LEP)**.** *[9] Given the positions of n robots in the plane, the n robots are able to deterministically agree on the same robot L called the leader. Initially, the robots are in arbitrary positions, with the only requirement that no two robots are in the same position.*

Definition 2 ($APFP$)**.** *[9] The robots have in input the same pattern, called the target pattern $\mathcal{P}$, described as a set of positions in the plane given in lexicographic order (each robot sees the same pattern according to the direction and orientation of its local coordinate system). They are required to form the pattern: at the end of the computation, the positions of the robots coincide, in everybody's local view, with the positions of $\mathcal{P}$, where $\mathcal{P}$ may be translated, rotated, and scaled in each local coordinate system. Initially, the robots are in arbitrary positions, with the only requirement that no two robots are in the same position, and that the number of positions prescribed in the pattern and the number of robots are the same.*

The issue of whether $APFP$ or LEP can be solved or not according to some capabilities of the robots is addressed in [9]. Not surprisingly, both problems are not deterministically solvable in general, due to the anonymity and the disorientation of the robots. This is especially true for LEP for which the impossibility of breaking a possible symmetry makes LEP unsolvable. For that matter, in [6] the authors provide a complete characterization (necessary and sufficient conditions) on the robots positions to elect a leader in a deterministic way.

A first relationship between $APFP$ and LEP is given by the following theorem:

Theorem 1. *[9] If it is possible to solve $APFP$ for $n \geq 3$ robots, then LEP is solvable too.*

Naturally, an interesting question arises from the above theorem: "*Is the converse true?*". In other terms: "*With robots devoid of sense of direction, does APFP becomes solvable whenever the robots have the possibility to distinguish a unique leader?*" In [16], the authors provide a positive answer to this question assuming that robots have a common handedness. The latter allows to infer the orientation of the x-axis once the orientation of the y-axis is given. Their result holds in the semi-synchronous model (SSM), *a.k.a.* Model SYm in the literature.

In this paper, we show that this result also holds for $n \geq 4$ robots in a fully asynchronous model, called CORDA [7]. Combined with Theorem 1 and provided the robots have the chirality property, we deduce that Leader Election and Pattern Formation are two *equivalent* problems in $CORDA$ for $n \geq 4$ robots, in the precise sense that, the former problem is solvable if and only if the latter problem is solvable.

The rest of the paper is organized as follows: In Section 2, we describe the distributed systems. The proof of equivalence is given in Section 3 for any $n \geq 4$ by providing an algorithm working in CORDA. (Due to lack of space, technical proofs have been ommitted.) Finally, we make concluding remarks in Section 4.

2 Model

We adopt the model CORDA (COordination and control of a set of Robots in a totally Distributed and Asynchronous environment) introduced in [7]. The *distributed system* considered in this paper consists of n robots $r_1, r_2, \cdots, r_n$—the subscripts $1, \ldots, n$ are used for notational purpose only. Each robot r_i is viewed as a point in a two-dimensional space unbounded and devoid of any landmark. When no ambiguity arises, r_i also denotes the position in the plane occupied by that robot. Each robot has its own local coordinate system and unit measure. The robots do not agree on the orientation of the axes of their local coordinate system, nor on the unit measure.

Definition 3 (Sense of Direction). *A set of n robots has sense of direction if the n robots agree on a common direction of one axis (x or y) and its orientation. The sense of direction is said to be* partial *if the agreement relates to the direction only —ie. they are not required to agree on the orientation.*

Given an x-y Cartesian coordinate system, the *handedness* is the way in which the orientation of the y axis (respectively, the x axis) is inferred according to the orientation of the x axis (resp., the y axis).

Definition 4 (Chirality). *A set of n robots has chirality if the n robots share the same handedness.*

In the rest of this paper, we assume that the robots have chirality and have no sense of direction.

The robot's life is viewed as an infinite sequence of cycles. Each cycle is a sequence of four states Wait-Observe-Compute-Move characterized as follows.

Life cycle. Initially, a robot is in the waiting state (**Wait**). Asynchronously and independently from other robots, it observes its surroundings (**Observe**) by using its sensors. The sensors return a set of all the positions occupied by at least one robot, with respect to its own coordinate system. Then, from its new observations the robot computes its next location (**Compute**) according to a given protocol which is the same one for all the robots. Once the computation is done, the robot moves towards its new location (**Move**). Finally, the robot returns to the waiting state.

The distance traveled by a robot in a cycle is unpredictable and thus, the robot may stop its motion before reaching the computed location. However, the distance traveled by a robot r in a move is neither infinite nor infinitesimally small. In particular, there exists a constant $\sigma_r > 0$ such that if the destination point is closer than σ_r, r will reach it; Otherwise, r will move towards it of at least σ_r. Note that no robot (including r) has the knowledge of σ_r. It is assumed that the amount of time spent in each phase of a cycle is finite but unpredictable and may be different for each cycle and for each robot. That is why the robots are considered to be fully asynchronous.

Finally we assume that the robots are *uniform* and *anonymous*, i.e, they all have the same program using no local parameter (such as an identity) allowing to differentiate any of them. Moreover, they have no direct means of communication and they are *oblivious*, i.e., none of them can remember any previous observation nor computation performed in any previous cycles.

3 Equivalence for $n \geq 4$

In this section we prove the main result of this paper:

Theorem 2. *In CORDA, assuming a group of $n \geq 4$ robots having chirality and devoid of any kind of sense of direction, LEP is solvable if and only if APFP is solvable.*

To prove Theorem 2, from Theorem 1, it remains to show the following lemma:

Lemma 1. *In CORDA, assuming a group of $n \geq 4$ robots having chirality and devoid of any kind of sense of direction, if LEP is solvable, then APFP is solvable.*

The remainding of this section is devoted to prove Lemma 1 by providing a protocol that forms an arbitrary target pattern assuming that, initially the robots are in a *leader configuration*, wherein the robots are able to deterministically elect a leader.

The overall idea of our algorithm consists of the three following main steps: First, by moving to some appropriate positions, the robots build a kind of global coordinate system. Next, they compute the final positions to occupy in order to form the pattern. Finally, the robots carefully move towards these final positions, while maintaining the global coordinate system invariant. In the next subsection

(Subsection 3.1), we provide basic definitions and properties leading to describe what is an (equivalent) agreement configuration. Then, in Subsection 3.2, we will give the distributed algorithm with its correctness proof.

3.1 Agreement Configuration

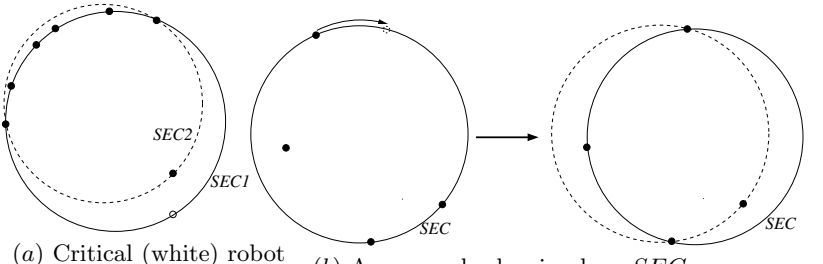
In the rest of this paper, we assume the set of all the positions $\mathcal{Q}$ occupied by the robots in the plane is the set of all the coordinates expressed in a cartesian coordinate system $\mathcal{S}$ which is unknown for all the robots. However, all the coordinates $\mathcal{Q}$ expressed in $\mathcal{S}$ coincide with all the cordinates $\mathcal{Q}$ expressed in everybody's local system where $\mathcal{Q}$ may be translated, rotated or scaled.

Definition 5 (Smallest enclosing circle). *[4] Given a set $\mathcal{Q}$ of $n \geq 2$ positions $p_1, p_2, \cdots, p_n$ on the plane, the smallest enclosing circle of $\mathcal{Q}$, called $SEC(\mathcal{Q})$, is the smallest circle enclosing all the positions in $\mathcal{Q}$.*

When no ambiguity arises, $SEC(\mathcal{Q})$ is shortly denoted by SEC and $SEC(\mathcal{Q}) \cap \mathcal{Q}$ indicates the set of all the positions both on $SEC(\mathcal{Q})$ and $\mathcal{Q}$. Besides, we say that a robot r is inside SEC if and only if r is not located on the circumference of SEC. In any configuration $\mathcal{Q}$, SEC is unique and can be computed in linear time [11]. Note that since the robots have the ability of chirality, they are able to agree on a common orientation of SEC, denoted $\circlearrowright$, in the sequel referred to as the clockwise direction.

Property 1. [15] SEC passes either through two of the positions that are on the same diameter (opposite positions), or through at least three positions. SEC does not change by eliminating or adding positions that are inside it. SEC does not change by adding positions on its boundary. However, it may be possible that SEC changes by either eliminating or moving positions on its circumference.

Examples showing the latter assertion of Property 1 are proposed in Figure 1.



(a) Critical (white) robot cannot be deleted without changing SEC.

(b) An example showing how SEC may change by moving one robot.

Fig. 1. Examples illustrating Property 1

Definition 6 (Critical position). *[10] Given a set $\mathcal{Q}$ of distinct positions. We say that a position p is critical iff $SEC(\mathcal{Q}) \neq SEC(\mathcal{Q} \setminus \{p\})$.*

An example of such a critical robot is given by Figure 1, Case (a). According to Property 1, a critical position cannot be inside SEC. So, the following corollary holds:

Corollary 1. *Let $\mathcal{Q}$ be a configuration. If there exists a critical position p in $\mathcal{Q}$, then p is on the circumference of $SEC(\mathcal{Q})$.*

Before giving other properties about critical positions, we need to define extra notions.

Definition 7 ($adjacent(r, C, \circlearrowleft)$). *Given a circle C and a group of robots located on it, we say that $r' = adjacent(r, C, \circlearrowleft)$ if r' is the next robot on C just after r in the clockwise direction.*

In the same way, we can define $adjacent(r, \circlearrowleft)$ in the counterclockwise direction. When no ambiguity arises, $adjacent(r, C, \circlearrowleft)$ is shortly denoted by $adjacent(r, \circlearrowleft)$. Sometimes, if $r' = adjacent(r, \circlearrowleft)$, we simply say that r' and r are *adjacent*.

Definition 8 ($angle(p, c, p', \circlearrowleft)$). *Given a circle C centered at c and two points p and p' located on it, $angle(p, c, p', \circlearrowleft)$ is the angle centered at c from p to p' in the clockwise direction.*

In the same way, we can define $angle(p, c, p', \circlearrowleft)$ in the counterclockwise direction.

The following properties are fundamental results about smallest enclosing circles:

Lemma 2. *[2] Let r_i, r_j and r_k be three consecutive robots on SEC centered at c such that $r_j = adjacent(r_i, \circlearrowleft)$ and $r_k = adjacent(r_j, \circlearrowleft)$. If $angle(r_i, c, r_k, \circlearrowleft) \leq 180^o$, then r_j is non-critical and SEC does not change by eliminating r_j.*

Corollary 2. *Let $SEC(\mathcal{Q})$ be the smallest circle enclosing all the positions in $\mathcal{Q}$. For all couple of positions r_i and r_j in $SEC(\mathcal{Q}) \cap \mathcal{Q}$ such that $r_j = adjacent(r_i, \circlearrowleft)$, we have $angle(r_i, c, r_j, \circlearrowleft) \leq 180^o$.*

Lemma 3. *[2] Given a smallest enclosing circle with at least four robots on it, there exists at least one robot which is not critical.*

Definition 9 (Concentric Enclosing Circle). *Given a set P of distinct positions. We say that C^P is a concentric enclosing circle if and only if it is centered at the center c of SEC, has a radius strictly greater than zero and it passes through at least one position in P.*

In the following, $\mathcal{SC}^P$ and $|\mathcal{SC}^P|$ respectively denote the set of all the concentric enclosing circle in P and its cardinality. For some k such that $1 \leq k \leq |\mathcal{SC}^P|$, C_k^P indicates the k^{th} greatest concentric enclosing circle in P and $\bigcup_{i=1}^{k} C_i^P$ is

the set of the k first greatest enclosing circles in P. Moreover, we assume that a position (or robot) located inside a concentric enclosing circle C_k^P is not on the circumference of C_k^P. $C_i^P \cap P$ indicate the set of all the positions both on C_i^P and P.

Remark 1. From Definition 9, SEC is the greatest concentric enclosing circle of SC (i.e., $SEC = C_1$) and the center of SEC cannot be a concentric enclosing circle.

From Definition 9, we can introduce the notion of *agreement configuration*:

Definition 10 (Agreement Configuration). *A configuration $\mathcal{Q}$ is an agreement configuration if and only if both following conditions hold:*
1. There exists a robot r_l in $\mathcal{Q}$ such that r_l is the unique robot located on the smallest concentric enclosing circle $C^{\mathcal{Q}}_{|\mathcal{SC}^{\mathcal{Q}}|}$,
2. There is no robot at the center of $SEC(\mathcal{Q})$.

In an agreement configuration, r_l is called the *leader*.

Definition 11. *Two agreement configuration $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is said to be* equivalent *if and only if both following conditions hold:*
1. $SEC(\mathcal{Q}_1)$ and $SEC(\mathcal{Q}_2)$ are superimposed.
2. Let c_1 and c_2 be respectively the center of $SEC(\mathcal{Q}_1)$ and the center of $SEC(\mathcal{Q}_2)$. Let r_{l1} and r_{l2} be respectively the leader in $\mathcal{Q}_1$ and the leader in $\mathcal{Q}_2$. $[c_1, r_{l1})$ and $[c_2, r_{l2})$ are superimposed.

3.2 Protocol

Starting from a leader configuration, the protocol, shown in Algorithm 1, allows to form any target pattern $\mathcal{P}$. It is a compound of two procedures presented below:
1. Protocol $<$`Leader`$\rightsquigarrow$`Agreement`$>$ transforms an arbitrary leader configuration into an agreement configuration;
2. Protocol $<$`Agreement`$\rightsquigarrow$`Pattern`$>$ transforms an agreement configuration into a pattern $\mathcal{P}$.

Algorithm 1. Form an arbitrary pattern starting from a leader configuration ($n \geq 4$).

$\mathcal{P}$:= the target pattern ;
if the robots do not form the target pattern
then **if** the robots do not form an agreement configuration
then Execute $<$`Leader`$\rightsquigarrow$`Agreement`$>$;
else Execute $<$`Agreement`$\rightsquigarrow$`Pattern`$>$;

Algorithm 2. Procedure $<$ `Leader` $\rightsquigarrow$ `Agreement` $>$ for any robot r_i in an arbitrary leader configuration

$\mathcal{Q}$:= the configuration where the robots currently lies;
$r_l := Leader(\mathcal{Q})$;
c := center of $SEC(\mathcal{Q})$
if r_l is located at c
then r_k := the closest robot to $c \in \mathcal{Q} \setminus \{r_l\}$;
p := the middle of the segment $[r_l; r_k]$;
if I am r_l **then** $MoveTo(p, \rightarrow)$;
else **if** r_l is not critical
then p := the middle of the segment $[r_l; c]$;
if I am r_l **then** $MoveTo(p, \rightarrow)$;
else /* r_l is critical and r_l is on SEC*/
r_k := the first non-critical robot starting from r_l on SEC in clockwise.
if I am r_k
then p := the middle of the segment $[r_k; c]$; $MoveTo(p, \rightarrow)$; **endif**

Procedure $<$ `Leader` $\rightsquigarrow$ `Agreement` $>$. In a leader configuration, the following theorem holds:

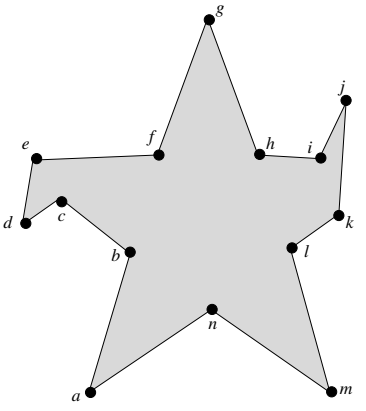
Theorem 3. *[6] If the robots are in a leader configuration, then they can distinguish a unique leader which is one of the closest robot to the center of the smallest enclosing circle of the configuration, provided that they share the property of chirality.*

From Theorem 3, we know that we can distinguish a unique robot r_l, called the leader, which is one of the robots closest to the center c of $SEC(\mathcal{Q})$. However, according to Definition 10, if r_l is at the center of $SEC(\mathcal{Q})$ or if r_l is not the unique robot closest to the center of $SEC(\mathcal{Q})$, $\mathcal{Q}$ is not an agreement configuration. In that case, Procedure $<$ `Leader` $\rightsquigarrow$ `Agreement` $>$ allows to transform the leader configuration into an agreement configuration. Algorithm 2 describes Procedure $<$ `Leader` $\rightsquigarrow$ `Agreement` $>$. In Algorithm 2, we use two subsoutines: $Leader(\mathcal{Q})$ and $MoveTo(p, \rightarrow)$. The former returns the unique leader from a leader configuration $\mathcal{Q}$. The latter allows a robot r to move towards the point p, using a straight movement.

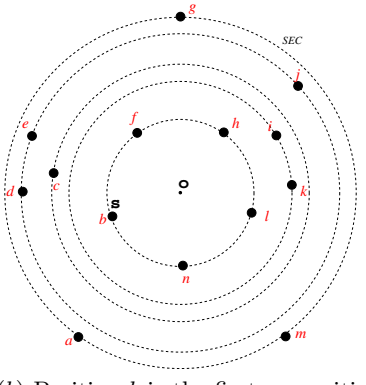
Procedure $<$ `Agreement` $\rightsquigarrow$ `Pattern` $>$ Intuitively, once the robots are in an agreement configuration, they can also agree on their final positions—refer to Definition 12. Then, some selected robots (Definition 14) begin to occupy them, starting from those situated on SEC, and then on all the circles concentric to SEC from the largest to the smallest passing through at least one of the final positions—refer to Definition 13. During this phase, the final positions are maintained unchanged, by making sure that the robots remain in an equivalent agreement configuration until the pattern is formed. In particular, we make sure that no angle above 180^o is created on SEC—otherwise, according to Corollary 2 SEC changes—and that

the leader of the agreement configuration remains the unique closest robot from the center of SEC and do not leave the radius where it is located.

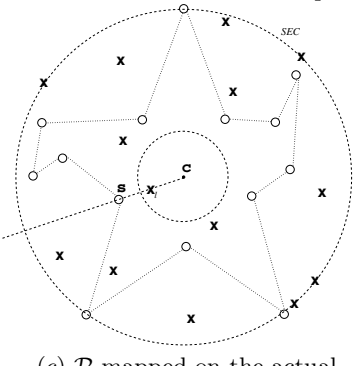
Before presenting Procedure $<$`Agreement`$\rightsquigarrow$`Pattern`$>$ shown in Algorithm 3, we need the following definitions:



(a) Positions a to n form the target pattern $\mathcal{P}$.



(b) Position b is the first non-critical position s in $\mathcal{P}$.



(c) $\mathcal{P}$ mapped on the actual agreement configuration $\mathcal{Q}$ formed by the robots (Robots are depicted as x's in the figure).

Fig. 2. An example showing a pattern $\mathcal{P}$ mapped on an agreement configuration $\mathcal{Q}$— Definition 12

Definition 12 ($Map(\mathcal{Q},\mathcal{P})$)**.** *Let $\mathcal{Q}$ and $\mathcal{P}$ be respectively an agreement configuration formed by the robots in the plane and a target pattern.*
$Map(\mathcal{Q},\mathcal{P})$ is the set of all the final positions $\mathcal{P}$ expressed in the plane where the robots currently lie and computed as follows:
1. First, the center of $SEC(\mathcal{P}))$ is translated to the center of $SEC(\mathcal{Q})$.
2. Then, let o, c, r_l and s be respectively the center of $SEC(\mathcal{Q})$, the center of $SEC(\mathcal{P})$, the leader in $\mathcal{Q}$ and the first non-critical position (in the lexicographic

order) located on the smallest concentric enclosing circle of $\mathcal{P}$. $\mathcal{P}$ is rotated so that the half-line $[o, r_l)$ is viewed as the half-line $[c, s)$.
3. Finally, $\mathcal{P}$ is scaled with respect to the radius of $SEC(\mathcal{Q})$ in order that all the distances are expressed according to the radius of $SEC(\mathcal{Q})$. In particular $SEC(\mathcal{Q}) = SEC(\mathcal{P})$.

An example showing the construction of Definition 12 is given in Figure 2.

Definition 13 ($(k, \mathcal{P})$-partial pattern). *Let $\mathcal{Q}$ and $\mathcal{P}$ be respectively an agreement configuration formed by the robots in the plane and a target pattern. We say that:*
1. $\mathcal{Q}$ is a $(0, \mathcal{P})$-partial pattern if the leader in $\mathcal{Q}$ is inside the smallest concentric enclosing circle of $Map(\mathcal{Q}, \mathcal{P})$.
2. $\mathcal{Q}$ is a $(k, \mathcal{P})$-partial pattern with $1 \leq k \leq Min(|\mathcal{SC}^{\mathcal{Q}}|, |\mathcal{SC}^{\mathcal{P}}|)$ if the three following properties holds:

a. $\mathcal{Q}$ is a $(0, \mathcal{P})$-partial pattern.
b. $C_k^{Map(\mathcal{Q},\mathcal{P})} \cap Map(\mathcal{Q}, \mathcal{P}) \subseteqq C_k^{\mathcal{Q}} \cap \mathcal{Q}$.
c. $\bigcup_{i=1}^{k-1} C_i^{\mathcal{Q}} \cap \mathcal{Q} = \bigcup_{i=1}^{k-1} C_i^{Map(\mathcal{Q},\mathcal{P})} \cap Map(\mathcal{Q}, \mathcal{P})$.

In the sequel, we say that $\mathcal{Q}$ is a *maximal* $(k, \mathcal{P})$-partial pattern if $\mathcal{Q}$ is a $(k, \mathcal{P})$-partial pattern and not a $(k+1, \mathcal{P})$-partial pattern.

Definition 14 (Extra robots). *Let $\mathcal{P}$ and $\mathcal{Q}$ be respectively a target pattern and a configuration formed by the robots in the plane such that $\mathcal{Q}$ is a maximal $(k, \mathcal{P})$-partial pattern. We say that a robot r is an extra robot if one of the two following properties holds:*

1. *$k = 0$, r is inside $SEC(\mathcal{Q})$, and r is not the leader in $\mathcal{Q}$;*
2. *$k \geq 1$ and*
   (a) *either r is inside the enclosing circle $C_k^{Map(\mathcal{Q},\mathcal{P})}$ and r is not the leader in $\mathcal{Q}$;*
   (b) *or r is on the circumference of $C_k^{Map(\mathcal{Q},\mathcal{P})}$ and r does not occupy a position in $C_k^{Map(\mathcal{Q},\mathcal{P})} \cap Map(\mathcal{Q}, \mathcal{P})$.*

The routine $Nearest\_extra\_robot(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \mathcal{Q}, Map(\mathcal{Q}, \mathcal{P}))$ returns an extra robot r such that r is the closest extra robot to $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ which is not located on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$. If several candidates exists, then the extra robots inside $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ have priority. Finally, if there is again several candidates then these latter ones are located on the same concentric circle C centered at the center c of $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ and the routine returns the extra robot, located on C, which is the closest in clockwise to the intersection between C and the half line $[c, r_l)$ (with r_l the leader in $\mathcal{Q}$).

$Nearest\_free\_point(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \mathcal{Q}, r)$ returns the nearest position from r which is located on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ and not occupied by any robot belonging to $\mathcal{Q}$. If there are two nearest positions then the routines returns the position which is

Algorithm 3. Procedure $<$ `Agreement` $\rightsquigarrow$ `Pattern` $>$ for any robot r_i in an agreement configuration

$\mathcal{Q}$:= the configuration where the robots currently lies;
$\mathcal{P}$:= the target pattern; /* $\mathcal{P}$ is the same for all the robots */
$r_l := Leader(\mathcal{Q})$;
s := the first non-critical position located on the smallest concentric enclosing circle of $Map(\mathcal{Q}, \mathcal{P})$;
if the robots do not form any $(k, \mathcal{P})$-partial pattern
then /*r_l is not inside the smallest concentric enclosing circle of $Map(\mathcal{Q}, \mathcal{P})$ */
  p := the middle of the segment $[c; s]$;
  if I am r_l **then** $MoveTo(p, \rightarrow)$;
else /* the robots form a $(k, \mathcal{P})$-partial pattern */
  if the center of $SEC(\mathcal{Q}) \in Map(\mathcal{Q}, \mathcal{P})$
  then x := the center of $SEC(\mathcal{Q})$;
  else $x := s$;
  $Final\_Positions := Map(\mathcal{Q}, \mathcal{P}) \setminus \{x\}$;
  if all the positions in $Final\_Positions$ are occupied
  then **if** I am r_l **then** $MoveTo(x, \rightarrow)$;
  else k := the maximal k for which $\mathcal{Q}$ is a $(k, \mathcal{P})$-partial pattern;
    if there is at least one extra robot not located on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$
    then $r := Nearest\_extra\_robot(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \mathcal{Q}, Map(\mathcal{Q}, \mathcal{P}))$;
      $p := Nearest\_free\_point(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \mathcal{Q}, r)$;
      if I am r **then** $MoveTo(p, \rightarrow)$;
    else $Arrange(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, Final\_Positions)$

the closest in clockwise to the intersection between $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ and the half line $[c, r_l)$ (with c the center of $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ and r_l the leader).

$MoveTo(p, C, \circlearrowright)$ allows a robot to move toward a position p located on the circle C by moving along the boundary of C in clockwise. $MoveTo(p, C, \circlearrowleft)$ is similar but in counterclockwise.

$Arrange(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, Final\_Positions)$ allows all the robots on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ to occupy all the positions in $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})} \cap Final\_Positions$. The function is described by Algorithm 4 in which we use the following notions:

Definition 15 ($arc(p, p', C, \circlearrowright)$)**.** *Given a circle C and two points p, p' located on it, $arc(p, p', C, \circlearrowright)$ is the arc of circle C from p to p' in the clockwise direction, p being excluded (p' being included).*

Definition 16 (P-$arc(p_i, p_{i+1}, C, \circlearrowright)$)**.** *Given a target pattern $\mathcal{P}$ and an agreement configuration $\mathcal{Q}$, we say that $arc(p_i, p_{i+1}, C, \circlearrowright)$ is a P-$arc(p_i, p_{i+1}, C, \circlearrowright)$ if and only if the three following properties holds:*
1. C is one of the concentric enclosing circle of $Map(\mathcal{Q}, \mathcal{P})$
2. p_i and p_{i+1} belong to $Final\_Positions$
3. $p_{i+1} = adjacent(p_i, C, \circlearrowright)$

Remark 2. From Definition 15, we know that p_i is not located on $P\text{-}arc(p_i, p_{i+1}, C, \circlearrowleft)$.

In the remainder, we say that a P-arc is *free* if there is no robot located on it. In Figure 3, the circles denote the positions to achieve. The crosses depict the robots. The P-arc starting after f (f excluded) and finishing at a is free.

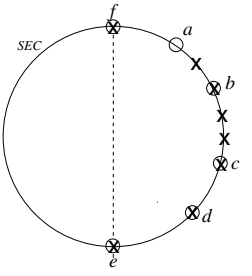


Fig. 3. An example showing a Deadlock Chain and a Deadlock Breaker

Definition 17 (Deadlock Chain). *A Deadlock Chain is a consecutive sequence of P-arc starting from a free P-arc P_0 and followed in the counterclockwise direction by a P-arc P_1 such that:*
1. P_1 is a $P\text{-}arc(p, p', C, \circlearrowleft)$ such that $angle(p, c, p', \circlearrowleft) = 180^o$ and there is only one robot r on it and r is located at p',
2. and P_1 is followed in counterclockwise by a consecutive sequence (possibly empty) of
$P\text{-}arc(p, p', C, \circlearrowleft)$ such that there is only one robot r on each of them and r is located at p',
and that consecutive sequence (possibly empty) is followed by a $P\text{-}arc(p, p', C, \circlearrowleft)$ such that there is at least two robots on it and one of them is located at p'. This P-arc is called the last P-arc of the deadlock chain.

In Figure 3, the segment starting from Position a (a included) to Position b (b excluded) forms a deadlock chain.

Definition 18 (Deadlock Breaker). *Let $P\text{-}arc(p, p', C, \circlearrowleft)$ be the last P-arc of a deadlock chain. The deadlock breaker is the robot located at p'.*

In Figure 3, the robot located at Position c is the deadlock breaker.

Sketch of Correctness Proof of Algorithm 1. We first show that by executing Algorithm 1, the smallest enclosing circle $SEC(\mathcal{Q})$ remains invariant—Lemma 4.

Lemma 4. *According to Algorithm 1, the smallest enclosing circle $SEC(\mathcal{Q})$ remains invariant.*

Algorithm 4. *Arrange*$(C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, Final\_Positions)$ executed by robot r_i on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$

/* I am r_i */
$p :=$ the closest position in $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})} \cap Final\_Positions \setminus \{r_i\}$ to r_i in clockwise;
if $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})} = SEC(\mathcal{Q})$
then **if** there is no robot in $arc(r_i, p, C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \circlearrowleft)$ or I am a deadlock breaker
then **if** I am a deadlock breaker
then $t :=$ the position s.t. $angle(r_i, c, t, \circlearrowleft) = \frac{1}{2} angle(r_i, c, p, \circlearrowleft)$;
$p := t$;
endif
$r_{i-1} := adjacent(r_i, SEC, \circlearrowleft)$;
$p' :=$ the position such that $angle(r_{i-1}, c, p', \circlearrowleft) = 180^o$;
$p'' :=$ the closest point to r_i in clockwise in $\{p; p'\}$;
if r_i is not located at p'' **then** $MoveTo(p'', SEC, \circlearrowleft)$;
endif
else **if** there is no robot in $arc(r_i, p, C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \circlearrowleft)$
then $MoveTo(p, C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}, \circlearrowleft)$;

Next, we prove that if the robots form a leader configuration which is not a final pattern $\mathcal{P}$ and not an agreement configuration, they eventually form an agreement configuration—Lemma 5.

Lemma 5. *If the robots form a leader configuration which is not a final pattern $\mathcal{P}$ and not an agreement configuration, they form an agreement configuration in a finite number of cycles.*

Starting from such a configuration, $Map(\mathcal{Q}, \mathcal{P})$ remains invariant or the target pattern $\mathcal{P}$ is formed—Lemma 6 and Corollary 3. Note that Corollary 3 assures that the two parts of Algorithm 1 (Protocol $<$ `Leader` $\rightsquigarrow$ `Agreement` $>$ and Protocol $<$`Agreement`$\rightsquigarrow$`Pattern`$>$) work in the asynchronous model *CORDA* even if the unique robot closest to the center of SEC is not still.

Lemma 6. *Starting from an agreement configuration $\mathcal{Q}$, the robots remain in an equivalent agreement configuration or the target pattern $\mathcal{P}$ is formed in a finite number of cycles.*

Corollary 3. *From an agreement configuration, $Map(\mathcal{Q}, \mathcal{P})$ remains invariant or the target pattern $\mathcal{P}$ is formed.*

It follows that from an agreement configuration which is not a $(k, \mathcal{P})$-partial pattern, the robots eventually form a $(0, \mathcal{P})$-partial pattern—Lemma 7.

Lemma 7. *From an agreement configuration which is not a $(k, \mathcal{P})$-partial pattern, the robots form a $(0, \mathcal{P})$-partial pattern in a finite number of cycles.*

From this point on, note that according to Algorithm 3, $Final\_Positions$ is equal to all the positions in $Map(\mathcal{Q}, \mathcal{P})$ except:

1. either the center c of $SEC(\mathcal{Q})$ if $c \in Map(\mathcal{P}, \mathcal{Q})$,
2. or the first non critical position located on the smallest concentric enclosing circle of $Map(\mathcal{Q}, \mathcal{P})$ if $c \notin Map(\mathcal{P}, \mathcal{Q})$

Next, we show by induction that, from a configuration being a maximal $(k, \mathcal{P})$-partial pattern, the robots eventually form a $(k+1, \mathcal{P})$-partial pattern or the target pattern $\mathcal{P}$ is formed—Lemmas 8 to 10.

Lemma 8. *Let $\mathcal{P}$ be a target pattern and let $\mathcal{Q}$ be a configuration which is a maximal $(k, \mathcal{P})$-partial pattern such that $1 \leq k < |\mathcal{SC}^{\mathcal{P}}|$. If all the extra robots are on $C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ then, all the positions in $Final\_Positions \cap C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ are occupied in a finite number of cycles.*

Lemma 9. *Let $\mathcal{P}$ be a target pattern and let $\mathcal{Q}$ be a configuration which is a maximal $(0, \mathcal{P})$-partial pattern. If all the extra robots are on $SEC(Map(\mathcal{Q}, \mathcal{P}))$ then, all the positions in $Final\_Positions \cap C_{k+1}^{Map(\mathcal{Q},\mathcal{P})}$ are occupied in a finite number of cycles.*

Lemma 10. *Let $\mathcal{P}$ be a target pattern and let $\mathcal{Q}$ be a configuration which is a maximal $(k, \mathcal{P})$-partial pattern. The robots form a $(k+1, \mathcal{P})$-partial pattern or the target pattern is formed, in a finite number of cycles.*

From Lemma 10 and by induction we deduce the following theorem:

Theorem 4. *Starting from a leader configuration, Algorithm 1 allows to solve $APFP$ in $CORDA$ among a group of $n \geq 4$ robots having chirality and devoid of any kind of sense direction.*

4 Conclusion

We studied the relationship between $APFP$ and LEP among robots having chirality in $CORDA$. We gave an algorithm allowing to form an arbitrary pattern starting from any geometric configuration wherein the leader election is possible. Combined with the result in [9], we deduce that $APFP$ and LEP are equivalent, *i.e.,* it is possible to solve $APFP$ for $n \geq 4$ if and only if LEP is solvable too. The possible equivalence for $n = 3$ remains an open problem.

Notice that our solution would not always guarantee the invariance of SEC if $n = 3$ and all the robots are placed on it. Indeed, in this particular case, if there does not exists two robots that are on the same diameter it would be impossible to remove one of the three without creating an angle greater than 180^o (which is not the case when $n \geq 4$). This is why the given solution only works if we have four robots or more. In a future work, we would like to investigate this case. Also, the issue of knowing whether the equivalence is still valid without chirality will be one of our main concern in future.

References

1. Canepa, D., Gradinariu Potop-Butucaru, M.: 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)
2. Cieliebak, M., Prencipe, G.: Gathering autonomous mobile robots. In: 9th International Colloquium on Structural Information and Communication Complexity (SIROCCO 9), pp. 57–72 (2002)
3. Cohen, R., Peleg, D.: Local spreading algorithms for autonomous robot systems. Theor. Comput. Sci. 399(1-2), 71–82 (2008)
4. Defago, X., Konagaya, A.: Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In: 2nd ACM International Annual Workshop on Principles of Mobile Computing (POMC 2002), pp. 97–104 (2002)
5. Dieudonné, Y., Labbani-Igbida, O., Petit, F.: Circle formation of weak mobile robots. TAAS 3(4) (2008)
6. Dieudonné, Y., Petit, F.: Deterministic leader election in anonymous sensor networks without common coordinated system. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 132–142. Springer, Heidelberg (2007)
7. 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)
8. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Distributed coordination of a set of autonomous mobile robots. In: IEEE Intelligent Veichle Symposium (IV 2000), pp. 480–485 (2000)
9. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. Theor. Comput. Sci. 407(1-3), 412–447 (2008)
10. Katreniak, B.: Biangular circle formation by asynchronous mobile robots. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 185–199. Springer, Heidelberg (2005)
11. Megiddo, N.: Linear-time algorithms for linear programming in r^3 and related problems. SIAM J. Comput. 12(4), 759–776 (1983)
12. Prencipe, G.: Distributed coordination of a set of autonomous mobile robots. Technical Report TD-4/02, Dipartimento di Informatica, University of Pisa (2002)
13. Suzuki, I., Yamashita, M.: Agreement on a common x-y coordinate system by a group of mobile robots. Intelligent Robots: Sensing, Modeling and Planning, 305–321 (1996)
14. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots - formation of geometric patterns. SIAM Journal of Computing 28(4), 1347–1363 (1999)
15. Welzl, E.: Smallest enclosing disks (balls and ellipsoids). In: Maurer, H.A. (ed.) New Results and New Trends in Computer Science. LNCS, vol. 555, pp. 359–370. Springer, Heidelberg (1991)
16. Yamashita, M., Suzuki, I.: Characterizing geometric patterns formable by oblivious anonymous mobile robots. Theoretical Computer Science (to appear, 2010)

Rendezvous of Mobile Agents in Directed Graphs

Jérémie Chalopin[1], Shantanu Das[1], and Peter Widmayer[2]

[1] LIF, CNRS & Aix-Marseille University, France
jeremie.chalopin@lif.univ-mrs.fr, shantanu.das@acm.org
[2] Institute of Theoretical Computer Science, ETH Zürich, Switzerland
widmayer@inf.ethz.ch

Abstract. We study the problem of gathering at the same location two mobile agents that are dispersed in an unknown and unlabeled environment. This problem called Rendezvous, is a fundamental task in distributed coordination among autonomous entities. Most previous studies on the subject model the environment as an undirected graph and the solution techniques rely heavily on the fact that an agent can backtrack on any edge it traverses. However, such an assumption may not hold for certain scenarios, for instance a road network containing one-way streets. Thus, we consider the case of strongly connected directed graphs and present the first deterministic solution for rendezvous of two anonymous (identical) agents moving in such a digraph. Our algorithm achieves rendezvous with detection for any solvable instance of the problem, without any prior knowledge about the digraph, not even its size.

Keywords: Distributed Algorithm, Directed Graph, Leader Election, Rendezvous, Anonymous Networks, Mobile Agents, Graph Exploration.

1 Introduction

One of the fundamental problems in distributed coordination is the task of gathering together two autonomous entities that are dispersed in a unknown environment. The problem, called *rendezvous* problem, has been studied both for robots moving in a terrain or software agents moving in a network. In the former case, the environment is a bounded (or unbounded) region of the infinite two-dimensional plane. While in the latter case, the environment is modeled as a graph where the two agents are initially located at distinct nodes of the graph. When the two dispersed entities can not communicate from a distance, solving rendezvous is essential for an exchange of information or for achieving even the simplest form of coordination between the mobile entities. The rendezvous problem belongs to the class of symmetry-breaking problems (e.g. leader election is another such problem) that are central to study of computability in distributed systems. The importance of the problem is evident from the large volume of literature [10, 11, 14–17, 19] dedicated to solving the problem under various conditions and restrictions.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 282–296, 2010.

For rendezvous in graphs, almost all previous results were restricted to undirected graphs. However, in many scenarios, it is natural to model the environment as a directed graph. For instance, in a communication network such as the internet, it may be possible to communicate in only one direction along some of the channels. Another example of a directed graph is a road network containing some one way streets.

To the best of our knowledge, there are no known results on rendezvous of agents in arbitrary and unknown directed graphs. Moreover, the solutions for the undirected graphs do not carry over to the case of digraphs, as they heavily rely on the ability of the agents to backtrack on a traveled path.

In this paper, we show how two identical agents can rendezvous in an unknown digraph. The agents are allowed to move along the arcs of the digraph from the tail end to head end. We assume the digraph to be strongly connected so that any node of the digraph can be reached. An agent located at any node v of the digraph can choose one of the several arcs outgoing from v and this choice is made by a deterministic algorithm which specifies the moves of the agents. The idea is to design an algorithm that when executed by each agent ensures that after a finite number of moves, the agents terminate in exactly one node of the digraph. In the most general setting, the nodes of the graph are anonymous (i.e. the agents can not distinguish between nodes of the same degree) and the agents themselves do not have any unique identifiers either. In this setting, rendezvous can not be solved in arbitrary graphs, using deterministic algorithms. We thus focus on the problem of "rendezvous with detect" where the agents rendezvous if and only if it is deterministically possible for the given instance, and otherwise stop and report that problem is not solvable. Solving rendezvous usually requires the agents to completely explore the digraph. Exploration of unknown and unlabeled graphs (or digraphs) is difficult unless the agents are allowed to mark the nodes. As in several previous papers on the topic [3, 4, 9, 11, 13], we allow the agents to mark nodes of the digraphs by writing on whiteboards available at each node.

Related Works: The problems of rendezvous and leader election has been extensively studied from the point of view of computability in unknown and unlabeled graphs, starting from the work of Angluin [2]. Many researchers have focussed on characterizing the conditions under which distributed coordination problems (such as leader election) can be solved in the absence of unique identifiers for the participating entities. Characterizations of the solvable instances for leader election in *message passing systems* have been provided by Boldi *et al.* [7] and by Yamashita and Kameda [18] among others. The rendezvous problem which falls under the same class of symmetry breaking problems, has been solved under various different assumptions such as distinct labels for the agents, sense of direction information, or under a synchronous setting (e.g. [4, 10, 16, 17, 19]). The problem has been studied for specific topologies such as rings [11], tori [17], trees [15] as well as in the most general case of an unlabeled terrain [14]. The idea of solving rendezvous by marking the starting locations with tokens was first proposed by Baston and Gal [5].

Most of the above results are for undirected graphs. Boldi and Vigna [7, 8] considered directed graphs and they introduced the concept of *fibrations* to illustrate certain properties of digraphs. For instance, they showed that deterministic leader election is solvable only in *fibration-free* digraphs.

For directed graphs, even exploration using deterministic algorithms is relatively difficult compared to undirected graphs. The problem of exploring and constructing a map of an unknown strongly connected digraph has been studied previously, from the point of view of minimizing the number of arc traversals (also called the "moves" complexity). For exploration of node-labelled digraphs, the best known algorithm [1] has a cost of $O(d^{logd} * m)$ moves for a digraph G having deficiency d (i.e. d arcs are needed to make G Eulerian). Another optimization criteria is minimizing the amount of memory required by an agent to explore a directed graph (See e.g. [13])

When the nodes of the digraph are not labelled, it is necessary to mark the nodes so that they can be recognized on subsequent visits. Exploration of unlabeled and unknown digraphs using pebbles to mark vertices, has been studied by Bender et al. [6]. A related problem of searching for a black hole has been recently studied for directed graphs (see [9]).

Our Results: We study the rendezvous problem for two identical agents starting from arbitrary locations in strongly connected directed graphs in the absence of unique identifiers. We present an algorithm for solving *Rendezvous with Detect* in any strongly connected digraph. Our results give a characterization of the solvable instances for rendezvous in this setting. Thus, our result generalizes previous results on rendezvous in undirected graphs to the more general case of directed graphs. The algorithm considered in this paper requires $O(m.n)$ moves for graphs of n nodes and m edges, whereas for undirected graphs, the same problem can be solved in $O(m)$ moves. Our algorithm is universal and works for digraphs of arbitrary size and topology (i.e. no prior knowledge about the digraph is required).

2 Definitions and Properties

2.1 Our Model

The environment where the agents are operating is represented by a strongly connected directed graph $G = (V, A)$ where V is the set of nodes and A is the set of arcs of G. We denote by n and m, the number of nodes and the number of arcs of G, respectively. There are two identical mobile agents located in two distinct nodes of the digraph G. The agents execute the same algorithm and start from the same initial state (though not necessarily at the same time). The agents do not have any prior knowledge of the graph G, not even n, the size of G. Every action performed by an agent takes a finite but otherwise unpredictable amount of time. The node from where an agent a starts the algorithm (i.e. the initial location) is called the *homebase* of agent a, denoted by $h(a)$. The objective of the agents is to solve rendezvous i.e. meet at any unspecified node

of the digraph G. We denote an instance of the rendezvous problem by (G, χ_p) where $\chi_p : V \rightarrow \{0, 1\}$ is a node-labeling of G such that $\chi_p(v) = 1$ if there exists an agent a such that $h(a) = v$, and $\chi_p(v) = 0$ otherwise.

The agents can traverse the arcs of G only from the tail end to the head end (but not the other way). Since G is strongly connected each node has at least one incoming and at least one outgoing arc. We denote by $d_{in}(v)$ (resp. $d_{out}(v)$) the number of incoming (resp. outgoing) arcs at v. The arcs going out from a node v are locally oriented i.e. they are labelled as $1, 2, \ldots, d_{out}(v)$. Similarly, the arcs incoming at a node v are labelled as $1, 2, \ldots, d_{in}(v)$ and an agent arriving at node v by an arc e, knows the label of e at v. Note that each arc $e = (u, v)$ of G has two labels, one at the tail end u and the other at the head-end v. The arc labeling of G (sometimes called the port numbering) is specified by $\lambda : A(G) \rightarrow \mathbb{N}^2$. We call this the Duplex arc-labeling model. Note that this model corresponds to the *Port-to-Port*(PP) model in the message passing system, as defined by Yamashita and Kameda [18].

We will also consider the more general case of Simplex arc-labeling where only outgoing arcs at a node v are labelled, but not the incoming arcs. In other words, an agent arriving at node v does not know through which arc it arrived. In this case, the arc labeling of G is specified by $\lambda : A(G) \rightarrow \mathbb{N}$.This model corresponds to the *Port-to-Mailbox*(PM) model in the message passing system of Yamashita and Kameda [18]. The difference between the two models and the effect it has on the computations, is further discussed in Section 4.

Each node $v \in G$ has a whiteboard and any agent visiting node v can read or write to that whiteboard. We use label(v) to denote the contents of the whiteboard of the node v. Initially all whiteboards are empty, so label$(v)= \phi$, for each $v \in V(G)$. Whenever the two agents are in the same node, they see each other and stop.

2.2 Coverings of Digraphs

For the definitions given below, we consider multigraphs, i.e. digraphs that possibly contain parallel arcs and self loops. Such a multigraph is denoted by $D = (V(D), A(D), s, t)$ where $V(D)$ is a set of vertices, $A(D)$ is a set of arcs, and s and t are two functions that assign to each arc two elements of $V(D)$: a source and a target. A path between two vertices u and v in D is a sequence of arcs $a_1, a_2, \ldots a_p$ such that $s(a_1) = u, \forall 1 \leq i \leq p-1, t(a_i) = s(a_{i+1})$ and $t(a_p) = v$.

A *homomorphism* γ between the multigraph D and the multigraph D' is a mapping γ: $V(D) \cup A(D) \rightarrow V(D') \cup A(D')$ such that for each arc $a \in A(D)$, $\gamma(s(a)) = s(\gamma(a))$ and $\gamma(t(a)) = t(\gamma(a))$. An homomorphism γ is an *isomorphism* if γ is bijective. We now define the notion of graph coverings, borrowing the terminology of Boldi and Vigna[8].

Definition 1. *A* covering projection *is a homomorphism φ from D to D' satisfying the following: (i) For each arc a' of $A(D')$ and for each node v of $V(D)$ such that $\varphi(v) = v' = t(a')$ there exists a unique arc a in $A(D)$ such that $t(a) = v$*

and $\varphi(a) = a'$. (ii) For each arc a' of $A(D')$ and for each node v of $V(D)$ such that $\varphi(v) = v' = s(a')$ there exists a unique arc a in $A(D)$ such that $s(a) = v$ and $\varphi(a) = a'$.

If the homomorphism φ satisfies only property (i) above, it is called a *fibration*, whereas if it satisfies only property (ii), it is called an *opfibration*. If a covering projection $\varphi : D \to D'$ exists, D is said to be a *covering* of D' via φ and D' is called the base of φ. The notions of coverings extend to labelled digraphs in an obvious way: the homomorphisms must preserve the labeling. Throughout the paper we will consider digraphs where the vertices and arcs are labelled with labels from a recursive label set L (The labeling of D is denoted by μ_D). We will consider homomorphisms which preserve these labelings.

Definition 2. *A labeled digraph (D, μ_D) is said to be* covering-minimal *if there does not exist any labeled multigraph $(D', \mu_{D'})$, where D' is not isomorphic to D, such that (D, μ_D) is a covering of $(D', \mu_{D'})$ via a label-preserving covering projection.*

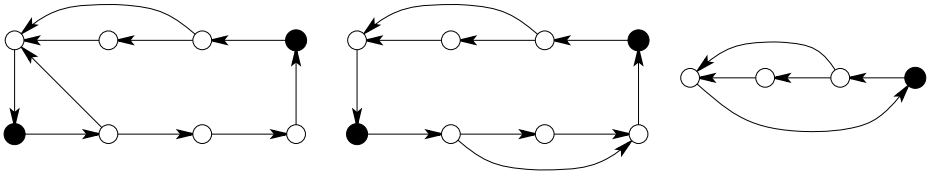


Fig. 1. There exists an opfibration from digraph G_1 to H and a covering projection from digraph G_2 to H. So, G_2 covers H, while G_1 is covering minimal. (The arc labeling is not shown in the figure and the node-labeling is illustrated by colors black and white.)

2.3 Impossibility Result

We now present a characterization of labelled digraphs where it is possible to solve rendezvous of mobile agents. First let us consider some known computability results for digraphs.

Lemma 1 ([8]). *Given a digraph $G(V, A)$ and an arbitrary (non-injective) labeling μ_G on G, if there exists a label-preserving covering projection from the labelled digraph (G, μ_G) to a labeled multigraph (H, μ_H), such that $|V(H)| < |V(G)|$ then it is impossible to solve leader election using message-passing in (G, μ_G).*

Lemma 2. *For a directed graph G, a port numbering λ on the arcs of G, and a bicoloring $\chi_p : V \to \{0, 1\}$ of the nodes of G, if there exists an algorithm for solving rendezvous of mobile agents that start from nodes of same color in G, then there is an algorithm for solving leader election by message passing in (G, λ, χ_p).*

The above result follows trivially from the fact that a mobile agent algorithm can be simulated by a message-passing algorithm where the messages are simply an encoding of the agent's states and their algorithm (see [3] for details). Once rendezvous is solved on the digraph G, there is exactly one node that contains all the agents and this node can be the designated leader.

Based on the above results, it follows that rendezvous is not possible in a labelled digraph (G, λ, χ_p) which is not covering minimal. In the next section we present an algorithm the solves rendezvous for all strongly connected digraphs that are covering minimal. Thus, we have the following result:

Theorem 1. *Given any strongly connected directed graph G, a port numbering λ on the arcs of G, and node labeling χ_p that denotes the initial placement of two identical agents in G, Rendezvous is solvable if and only if (G, λ, χ_p) is covering minimal.*

The above result holds irrespective of whether the port numbering λ is a simplex or duplex arc-labeling. In the next section, we present an algorithm for rendezvous assuming duplex arc-labeling (i.e. when both incoming and outgoing arcs are locally oriented). In section 4, we will show how to extend the algorithm for the case of simplex arc-labeling.

3 Rendezvous Algorithm

A usual strategy for deterministic rendezvous in unknown graphs is the following. Each agent marks its initial location, explores the graphs and builds a labelled map of the graph. From the map and initial position of the agents in the map, the agents can compute a unique location (if it exists, i.e. if the map is asymmetric) and both agents move to that location.

Exploration of a digraph is relatively easy if each agent is provided with a unique marker distinct from that of the other agent (for instance, suppose that one agent has a spray-can of *blue* paint and the other agent has a spray-can of *red* paint). An agent can explore the digraph G by following a depth-first strategy and writing on each visited node using the distinct marker (so that it can recognize it on later visits). Such an algorithm for exploring a digraph is presented as Algorithm Simple-Explore (See Algorithm 1). During the algorithm, the agent maintains a counter that is incremented whenever a new node is visited. The (counter,marker) pair provides a unique label for each node v that is visited by the agent. It is easy to see that the algorithm succeeds in building a map of any strongly connected digraph G.

Lemma 3. *On executing Algorithm Simple-Explore using a unique marker, on any strongly connected digraph G, the agent outputs a map of the digraph.*

Lemma 4. *Algorithm Simple-Explore requires $O(m \cdot n)$ moves by an agent.*

In general, the agents would not have access to distinct markers at the beginning of the algorithm. We now show how two identical agents may obtain distinct

Algorithm 1. SimpleExplore(MARKER)

```
MAP ← φ; Count ← 1;
Write (MARKER, Count) on current node;
Increment Count;
Add current node to MAP;
while ∃ a reachable node in MAP that has unexplored arcs do
    Go to the closest reachable node u that has unexplored arcs;
    Choose the unexplored arc e with smallest label;
    Traverse arc e = (u, v) to reach node v;
    if v is marked with MARKER, (i.e. v ∈ MAP) then
        add arc e to MAP;
    else
        Write (MARKER,Count) on node v;
        Increment Count;
        Add node v and arc e to MAP;
Return MAP;
```

markers for exploring the digraph. Each agent starts by exploring the digraph and whenever it arrives at ith node in its traversal, it simply writes the integer i on the node. The agent a maintains a partial map M_a containing the nodes and arcs visited so far by the agent (The map M_a is a subgraph of G). The problem is that when the agent arrives at an already labelled node, it can not determine whether v was marked by itself (i.e. $v \in M_a$) or node v was marked by the other agent (i.e. $v \notin M_a$). At this stage the agent does not know how to update its map M_a. So, the agent a executes a *checking* procedure (to be described later), in order to determine if node v was marked by another agent. At the end of the checking procedure, either the agent a realizes that the node v is marked by the other agent, or, the agent a is able to come back to a known node u in its map M_a. In the latter case, the agent continues with the traversal without adding the node v to its map. While in the former case, the agent would have detected an asymmetry i.e. a difference in the maps of the two agents. Thus, in this case, the agent starts a new round of exploration (i.e. it executes Algorithm 1) using its current map M_a as the unique marker to mark nodes during the exploration. We present below a high level description of the algorithm. A more detailed description (pseudocode) is given in Algorithm 2. We need the following definition.

Definition 3. *For any two nodes $u, u' \in G$, (u, u') is called a pseudo-arc if there is a (not necessarily simple) directed cycle $(u, v_0, v_1, \ldots, v_t, u' = u_0, u_1, \ldots, u_t = u)$ for some $t \geq 0$, such that label(u_i)=label(v_i), $0 \leq i \leq t$, $\lambda(v_i, v_{i+1}) = \lambda(u_i, u_{i+1})$, $0 \leq i < t$, and $\lambda(u_t, v_0) = \lambda(v_t, u_0)$. We say that an agent traversed the pseudo-arc (u, u') whenever it traverses the path $(u, v_0, v_1, \ldots, v_t, u')$.*

Notice that if $u_i = v_i$, $0 \leq i \leq t$ in the above definition, then the pseudo-arc corresponds to an actual arc in $A(G)$. On the other hand if $t = 0$ i.e. $u = u'$, then the pseudo-arc corresponds to a pair of symmetric arcs between u and v_0.

Algorithm DirRDV

Round I: The agent executes the following steps:

1. The agent executes an exploration procedure during which it traverses the graph, marks the nodes that it visits with a counter (unless the node is already marked) and adds any node that it marks, to its local map (the map is a labelled subgraph of G containing the nodes marked by the agent). The exploration is interrupted whenever either the agent reaches an already marked node v or there are no more unexplored arcs that can be reached from the current node.
2. If the agent has reached an already marked node v, the agent executes procedure *Check-Path* at current node v using its current map, in order to determine if the node v belongs to its map. If procedure Check-Path returns false then node v does not belong to its map and the agent has detected an asymmetry in the digraph. In this case the agent executes round two of the algorithm. Otherwise if Check-Path returns true, the agent would have returned to a known node u (as shown later) and the agent continues the exploration from node u.
3. If there are no unexplored arcs incident to any reachable node in the map, then the agent terminates the exploration and executes procedure Check-MAP. If the procedure returns true then the current instance is not covering minimal and thus, the agent declares that rendezvous is not possible. If procedure Check-MAP returns false then the agent has detected an asymmetry in the digraph. In this case the agent executes round two of the algorithm.

During Round-I, whenever the agent arrives at an unmarked node v, it writes on the whiteboard the label of v which consists of the round number, value of the counter, the current map, and the number of unexplored arcs incident to v. The partial map M_a maintained by the agent a contains each node marked by this agent (along with its label) and each explored arc which the agent used to reach an unmarked node (such an arc is called *discovery arc*). For any explored arc that leads to a marked node, the agent adds a *pseudo-arc* to its map.

Round II: Let M_a be the map obtained by an agent a at the end of first round. The agent executes algorithm Simple-Explore using M_a as the marker. Let (G, μ_G) be the output of the procedure and λ be the arc labeling part of μ_G. The agent can obtain the labeling χ_p by assigning $\chi_p(v) := 1$ for any node v whose label has counter value of 1; $\chi_p(u) = 0$ for any other node u. If (G, λ, χ_p) is covering minimal then the agent computes a unique rendezvous location (that depends only on G, λ and χ_p) and moves to that location using its map. Otherwise the agent declares the problem is not solvable and terminates.

Procedure Check-Path: This procedure is executed at a marked node v, by an agent a that reached v through the arc $e = (u, v)$. Let the current map of agent a be M which is a labelled subgraph of G. If the label of v does not appear in M then the procedure returns false. Otherwise there is a node u' in M that

Algorithm 2. DirRDV

```
/* Algorithm for rendezvous with detect, for two agents.          */
Write HOME on the starting node;
begin
    Round := 1; // Begin 1st Round
    MAP := φ;
    Count := 1;
    Write visited(Round, MAP, Count) on current node;
    Increment Count;
    Add current node to MAP;
    while ∃ a reachable node in MAP that has unexplored arcs do
        Go to the closest reachable node u that has unexplored arcs;
        Choose the unexplored arc e with smallest label;
        Mark e as explored at node u;
        Traverse arc e to reach node v;
        if v is already marked then
            Result := execute Check-Path (MAP, u, e, label(v));
            if Result = true, (i.e. ∃u0 ∈ MAP s.t. label(v)=label(u0)) then
                Add pseudo-arc (u, u0) to MAP;
                Write the new MAP at node u0;
            else
                Go to next round;
        else
            Add node v and arc e to MAP;
            Write visited(Round, MAP, Count) on node v;
            Increment Count;
    Result := Execute Check-Map(MAP);
    if Result = true then
        return "Rendezvous is not possible" ;
    else
        Go to next round;
end
begin
    Round := 2; // Begin 2nd Round
    MARKER:= (Round, MAP);
    Map2 := Simple-Explore (MARKER);
    v := RV-point(Map2);
    Go to node v using Map2;
end
```

has the same label as v. In this case there exists a directed path P from u' to u in M. The agent tries to traverse, from the current node v, the path having a same label-sequence as path P. If such a path does not exist or the agent detects a discrepancy between the path it is traversing and the path P in M, then we know that node v was marked by the other agent and thus, $v \notin M$. In this case, the procedure returns false. Otherwise the procedure returns true and this implies either node u' is same as node v (i.e. arc e completes a cycle in M)

or u' and v are symmetric nodes in G. In either of these cases, the agent has returned to the known node u' in its map, at the end of the checking procedure.

Procedure Check-Map: The agent executes this procedure to check the accuracy of the map that it obtained. The agent first writes the current map on the whiteboard of each node that it originally marked. Then the agent traverses the path corresponding to each pseudo arc in the map and compares the label of each node with the corresponding node in its map. If there is a mismatch, the procedure returns false (i.e. the agent has detected an asymmetry in the digraph). Otherwise the procedure returns true at the end of the traversal. (In this case the map is the base of a covering on the labelled graph (G, λ, χ_p) as shown later.)

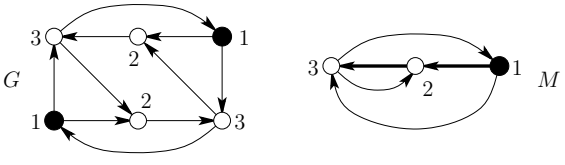


Fig. 2. The labelled digraph M is the map computed by the agents executing our algorithm on G. The thin arcs in M are the pseudo-arcs and black nodes are homebases.

Lemma 5. *If procedure Check-Path returns true for the arc (u, v), then the agent has returned to a known node u_0 in its current map and it has discovered a new pseudo-arc (u, u_0).*

Proof. Let v be the current node and $e = (u, v)$ be the last arc traversed by the agent (let's call it agent a), at the start of the procedure. The agent a is trying to determine whether node v belongs to its current map M_a. Note that $u \in M_a$ and there exists a node $u_0 \in M_a$ such that $label(u_0) = label(v)$ (otherwise the procedure would have returned false). If v was marked by agent a then $v = u_0$ (since the agent would not mark two nodes with the same label). In this case the agent returns to node u_0 at the end of the procedure after traversing the path $P = (u_0, u)$ followed by arc e.

Otherwise node v was marked by the other agent (say agent b), i.e. $v \neq u_0$. At the end of the procedure the agent arrives at a node w such that label(w)=label(v) (otherwise the procedure would return false). Note that either $w = v$ or $w = u_0$, since there are at most two nodes in the digraph having the same label. Suppose $w = v$ and let us consider the directed path P' traversed by the agent during the procedure. This path has the same labels as path P above and ends at the same node (v) as path P. There can not be two distinct incoming arcs with the same label at any node[1]. Thus the paths P and P' are the same, i.e. $u_0 = v$; A contradiction! Thus, the node $w \neq v$, which implies that $w = u_0$. So, the agent would finally arrive at node $u_0 \in M_a$ in either case. □

[1] Recall that we assume duplex arc-labeling.

Lemma 6. *If an agent reaches round two of the algorithm* **DirRDV***, then the two agents have distinct maps at the end of round one.*

Proof. An agent reaches round two, if during round one, either an execution of procedure Check-Path returns false or an execution of Check-MAP returns false. Suppose an agent a executes procedure Check-Path at node v after traversing arc $e = (u, v)$ and the procedure returns false. This implies that node $v \notin M_a$, the current map of agent a. Let M_b be the current map of the other agent. We know $v \in M_b$. If label(v) does not exist in M_a, then $M_a \neq M_b \bigcup H$ for any (possibly empty) subgraph H of G. Thus, even if agent b adds more nodes and arcs to its map, its map would never be same as M_a. Now let us consider the other case when one of the nodes $u_0 \in M_a$ has the same label as node $v \in M_b$. In this case there is no directed path starting from v that has the same sequence of labels as the path $P = (u_0, u) \in M_a$. The agent b can not mark any other node w to its map such that label(w)=label(v). Thus, the map of agent b would never contain any path corresponding to the path P in M_a. Thus the map M_a would be distinct from the map obtained by agent b at the end of its execution of round one of the algorithm.

Let us consider the other scenario when the procedure check-MAP returns false. In this case, either the map M_a is not strongly connected or the maps written on some of the nodes do not correspond to the map carried by the agent. In the latter case it is easy to see that the maps carried by the two agents are distinct. In the former case, notice that (G, λ, χ_p) is not a covering of M_a (since G is strongly connected and M_a is not). This implies that the map of the other agent (which is node disjoint from M_a) can not be an exact copy of M_a. □

The algorithm terminates unsuccessfully if the procedure CheckMAP returns true. This happens when the map constructed by the agent at the end of its exploration (i.e. when there are no more unexplored arcs) is a strongly connected digraph M such that the map written on any node that the agent visits is identical. In this case we can show that the original labelled digraph (G, λ, χ_p) is a covering of M.

Lemma 7. *If the procedure Check-MAP returns true, then the labeled digraph* (G, λ, χ_p) *is not covering minimal.*

Proof. Omitted.

Theorem 2. *For any strongly connected digraph G and a duplex arc-labeling λ on G, algorithm* **DirRDV** *solves the rendezvous of two agents placed initially according to χ_p, if (G, λ, χ_p) is covering minimal and otherwise detects that rendezvous is not solvable.*

Proof. The algorithm executed by an agent may terminate after round one only if the procedure CheckMAP returns true. In this case, we know that (G, λ, χ_p) is not covering minimal as proved in Lemma 7. In all other cases, both the agents reach round two of the algorithm. Thus, due to Lemma 6, the two agents have

distinct maps at the end of round one. The marker used by each agent consists of the round number and its map at the end of round one; This marker is distinct from the marker used by the other agent and also distinct from the markers used by both agents in the first round. Thus, due to Lemma 3 each agent successfully builds the map of (G, λ, χ_p) and if (G, λ, χ_p) is covering minimal, the agents can determine a unique location to meet. □

Theorem 3. *Algorithm Rendezvous requires $O(m.n)$ moves by the agents in total.*

Proof. The exploration part of the algorithm requires $O(m.n)$ moves by the agents (to explore each unexplored arc, $e = (u, v)$ the agent has to traverse at most n arcs to reach the source u of the arc). There are at most m calls to the procedure *Check-Path* and each execution of *Check-Path* requires $O(n)$ moves by an agent. The procedure Check-MAP is executed at most once during the algorithm and this requires $O(m.n)$ moves (since the size of the traversal path in the procedure is at most $m \cdot n$). Finally the second round of the algorithm has the same complexity as procedure *Simple-Explore*, which requires $O(m.n)$ moves as shown previously. □

4 Rendezvous without Incoming Arc Labels

In the previous section, we assumed the presence of local orientation on both the set of incoming arcs and the set of outgoing arcs incident to any node of the digraph. Having local orientation on the outgoing arcs is necessary for navigability in the digraph. In this section we consider digraphs G with simplex arc-labeling λ where incoming arcs at a node are not labelled. In fact, in this model, the agent can not even determine the in-degree of a node. The difference between the two models is the following. In the duplex arc-labeling model, any fibration on (G, λ, χ_p) was also an opfibration and vice versa. However, in the simplex arc-labeling model, there could exist an opfibration φ from (G, λ, χ_p) to a smaller labeled digraph even if (G, λ, χ_p) is covering minimal (and fibration-free). In this case there exists a node $v \in G$ such that two arcs e, f from distinct nodes u, u' lead to the node v such that $label(u) = label(u')$ and $\lambda(e) = \lambda(f)$. For example, see Figure 3 where the node labelled 4 has two similar incoming arcs (from two distinct nodes, both labelled with 2).

One consequence of the above observation is that Lemma 5 does not hold anymore. In other words, after executing procedure Check-Path, if the procedure returns true, then the agent may not have returned to a known node in its map. The agent is either back at a node u_0 in its own map or it is in some symmetric node v_0 in the map of the other agent. The agent was no means of determining which case is true. To solve this problem, we take the following approach. The agent continues with the exploration assuming that it is at node u_0. In this case the agent may have swapped places with the other agent or both agent may be in the same map. As the two subgraphs corresponding to the two partial maps are symmetric, it does not matter which part the agent is in. In fact we can

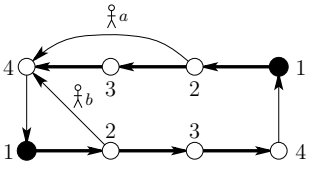


Fig. 3. An execution of the algorithm on a digraph G without incoming local orientation

show that the same algorithm from the previous section works also for the case of simplex arc labeling.

Theorem 4. *For any strongly connected digraph G, and a simplex arc-labeling λ on G, algorithm* **DirRDV** *solves the rendezvous of two agents placed initially according to χ_p, if (G, λ, χ_p) is covering minimal and otherwise detects that rendezvous is not solvable.*

Proof. During the exploration in round one of the algorithm, each agent maintains a map of the subgraph explored by it. Let M_a and M_b be the partial maps of the two agents a and b respectively. If the two maps are not symmetric, the agents will detect the asymmetry (during an execution of Check-Path or CheckMAP). Notice that Lemma 6 still holds. So, if the agents go to Round-2, they would have distinct maps and thus, they will successfully solve the rendezvous problem. Further Lemma 7 also continues to hold and if (G, λ, χ_p) not covering minimal, the agents detect that rendezvous is not possible.

The only difference from the previous section occurs when the maps of the two agents are identical but there exists a node v of G which has two incoming arcs e, f having the same label $x = \lambda(e) = \lambda(f)$. Suppose node $v \in M_a$, then there exists a node $v' \in M_b$ with identical label as node v. (For an illustration, see Figure 3 where the nodes labeled 4 correspond to v and v'). In this case, (G, λ, χ_p) is still covering minimal and we show below that the two agents indeed succeed in solving rendezvous for this instance.

The agent that is first to arrive at node v through either arc e or arc f, would add an arc labeled x incident to v in its map and write the new map at v. The other agent that arrives later to v through the other arc, would notice that the map at v already contains an arc labeled x incoming at the node v. This inconsistency would be detected when this agent executes Check-Path for the new arc that it traversed. Thus this agent would execute Round-2 of the algorithm using a map that does not contain the arcs e or f (such a map is distinct from the map of the other agent). Now consider the other agent which added the arc labeled x to its map. Note that the map written at v' does not contain such an arc. Thus, when the agent executes CheckMAP at the end of Round-1, it will arrive at node v' and detect the inconsistency in the maps. So, the procedure CheckMAP would return false and the agent would execute Round-2 of the algorithm using a map which contains one of the arcs e or f. Thus, the two agents execute round-2 using distinct maps (i.e. distinct markers) and they would succeed in solving rendezvous. □

5 Conclusions

In this paper, we considered the rendezvous of two agents in strongly connected digraphs. If the digraph G is not strongly connected, it is not always possible to explore the graph in general and thus, rendezvous is not possible. For any strongly connected digraph G, the algorithm we presented achieves the rendezvous of two agents whenever it is deterministically possible, i.e. whenever the labeled digraph (G, λ, χ_p) is covering minimal (or fibration-free). The rendezvous problem requires breaking the symmetry between two identical agents and a generalization of this problem, called the *Gathering* problem, is to gather together $k > 2$ agents at a single node of G. To solve the gathering of $k > 2$ agents we need to extend our solution by generalizing the technique of mapping paths in the graph using pseudo-arcs. A straightforward generalization of the algorithm would involve replacing each pseudo-arc traversal with the traversal of a cycle in the map, repeated $r = \gcd(1, 2, \ldots k)$ times (instead of just twice as in our algorithm), in order to ensure that the agent returns to the correct subgraph. However, this will blowup the cost of the algorithm exponentially. So, an interesting question to be answered by future research is whether $k > 2$ agents can gather in anonymous digraphs such that the number of moves is polynomial in the parameters n, m and k. For undirected graphs, it is already known to be possible using $O(m \cdot k)$ moves.

References

1. Albers, S., Henzinger, M.R.: Exploring Unknown Environments. SIAM Journal on Computing 29(4), 1164–1188 (2000)
2. Angluin, D.: Local and global properties in networks of processors. In: Proc. of 12th Symposium on Theory of Computing (STOC), pp. 82–93 (1980)
3. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Can we elect if we cannot compare? In: Proc. 15th ACM Symp. on Parallel Algorithms and Architectures (SPAA'03), pp. 200–209 (2003)
4. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Election and rendezvous in fully anonymous networks with sense of direction. Theory of Computing Systems 40(2), 143–162 (2007)
5. Baston, V., Gal, S.: Rendezvous search when marks are left at the starting points. Naval Research Logistics 48(8), 722–731 (2001)
6. Bender, M., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: Exploring and mapping directed graphs. In: Proc. 30th ACM Symp. on Theory of Computing (STOC), pp. 269–287 (1998)
7. Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 33–47. Springer, Heidelberg (2001)
8. Boldi, P., Vigna, S.: Fibrations of graphs. Discrete Math. 243, 21–66 (2002)
9. Czyzowicz, J., Dobrev, S., Kralovic, R., Miklík, S., Pardubská, D.: Black Hole Search in Directed Graphs. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 182–194. Springer, Heidelberg (2010)
10. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)

11. Dobrev, S., Flocchini, P., Prencipe, G., Santoro, N.: Multiple agents rendezvous in a ring in spite of a black hole. In: Papatriantafilou, M., Hunel, P. (eds.) OPODIS 2003. LNCS, vol. 3144, pp. 34–46. Springer, Heidelberg (2004)
12. Dobrev, S., Flocchini, P., Kralovic, R., Santoro, N.: Exploring a dangerous unknown graph using tokens. In: Proc. of 5th IFIP International Conference on Theoretical Computer Science, TCS (2006)
13. Fraigniaud, P., Ilcinkas, D.: Digraph exploration with little memory. In: Diekert, V., Habib, M. (eds.) STACS 2004. LNCS, vol. 2996, pp. 246–257. Springer, Heidelberg (2004)
14. Czyzowicz, J., Labourel, A., Pelc, A.: How to meet asynchronously (almost) everywhere. In: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA (2010)
15. Fraigniaud, P., Pelc, A.: Deterministic Rendezvous in Trees with Little Memory. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 242–256. Springer, Heidelberg (2008)
16. Kowalski, D.R., Malinowski, A.: How to meet in anonymous network. Theoretical Computer Science 399(1-2), 141–156 (2008)
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. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part I–Characterizing the solvable cases. IEEE Transactions on Parallel and Distributed Systems 7(1), 69–89 (1996)
19. Yu, X., Yung, M.: Agent rendezvous: A dynamic symmetry-breaking problem. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 610–621. Springer, Heidelberg (1996)

Almost Optimal Asynchronous Rendezvous in Infinite Multidimensional Grids

Evangelos Bampas[1,⋆], Jurek Czyzowicz[2], Leszek Gąsieniec[3], David Ilcinkas[1,⋆], and Arnaud Labourel[1,⋆,⋆⋆]

[1] LaBRI, CNRS / INRIA / Université de Bordeaux
{bampas,ilcinkas,labourel}@labri.fr
[2] Université du Québec
Jurek.Czyzowicz@uqo.ca
[3] University of Liverpool
L.A.Gasieniec@liverpool.ac.uk

Abstract. Two anonymous mobile agents (robots) moving in an asynchronous manner have to meet in an infinite grid of dimension $\delta > 0$, starting from two arbitrary positions at distance at most d. Since the problem is clearly infeasible in such general setting, we assume that the grid is embedded in a δ-dimensional Euclidean space and that each agent knows the Cartesian coordinates of its own initial position (but not the one of the other agent). We design an algorithm permitting the agents to meet after traversing a trajectory of length $O(d^\delta \text{polylog } d)$. This bound for the case of 2D -grids subsumes the main result of [12]. The algorithm is almost optimal, since the $\Omega(d^\delta)$ lower bound is straightforward.

Further, we apply our rendezvous method to the following network design problem. The ports of the δ-dimensional grid have to be set such that two anonymous agents starting at distance at most d from each other will always meet, moving in an asynchronous manner, after traversing a $O(d^\delta \text{polylog } d)$ length trajectory.

We can also apply our method to a version of the geometric rendezvous problem. Two anonymous agents move asynchronously in the δ-dimensional Euclidean space. The agents have the radii of visibility of r_1 and r_2, respectively. Each agent knows only its own initial position and its own radius of visibility. The agents meet when one agent is visible to the other one. We propose an algorithm designing the trajectory of each agent, so that they always meet after traveling a total distance of $O((\frac{d}{r})^\delta \text{polylog}(\frac{d}{r}))$, where $r = \min(r_1, r_2)$ and for $r \geq 1$.

1 Introduction

1.1 The Problem and the Model

Consider a Euclidean δ-dimensional space $\mathcal{F}$. We construct an infinite grid G_δ of dimension δ as follows. The set of nodes of G_δ are the points of $\mathcal{F}$ with

⋆ Partially supported by the ANR project ALADDIN, the INRIA project CEPAGE and by a France-Israel cooperation grant (Multi-Computing project).

⋆⋆ Corresponding author: LaBRI (bât A30), Université Bordeaux 1, 351 cours de la Libération, F-33405 Talence cedex, France. Tph. +33 (0)5 40 00 66 69.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 297–311, 2010.

integer coordinates. We link by an edge two nodes $u = (u_1, u_2, \ldots, u_\delta)$ and $v = (v_1, v_2, \ldots, v_\delta)$ iff there is $i \in \{1, 2, \ldots, \delta\}$, s.t., $\forall j \neq i, u_j = v_j$ and $u_i = v_i \pm 1$. At each node, the endpoints of edges incident to it are labeled by unique integers from the set $\{1, 2, \ldots, 2\delta\}$, called *port numbers*.

The *route* of each agent is a sequence of adjacent edges which are subsequently traversed during its movement. The actual timing of the *walk* of each agent along its route is *asynchronous*, i.e., it is controlled by an adversary. The adversary initially places both agents at any two nodes in the grid. Given the coordinates of its initial location v_0, the route chosen by an agent is a sequence of edges in the grid $(e_1, e_2, \ldots)$, s.t., in stage i the agent traverses the edge $e_i = [v_{i-1}, v_i]$, starting at v_{i-1} and finishing at v_i. Stages are repeated indefinitely until rendezvous is accomplished. We assume that each agent starts its walk at any time, but both agents are placed by the adversary at their respective initial positions at the same time, and since that moment any moving agent may encounter the other agent, even if the other agent didn't start its walk yet.

We describe the walk f of an agent on its route following definitions from [12, 15, 16]. Let $R = (e_1, e_2, \ldots)$ be the route of an agent. Let $(t_1, t_2, \ldots)$, where $t_1 = 0$, be an increasing sequence of reals, chosen by the adversary, that represent points in time. Let $f_i : [t_i, t_{i+1}] \to [v_i, v_{i+1}]$ be any monotonous, continuous function, chosen by the adversary, s.t., $f_i(t_i) = v_i$ and $f_i(t_{i+1}) = v_{i+1}$. For any $t \in [t_i, t_{i+1}]$, we define $f(t) = f_i(t)$. The interpretation of the walk f is as follows. At time t the agent is at point $f(t)$ of its route. The adversary may arbitrarily vary the speed of the agent for as long as the walk of the agent in each segment is continuous and monotonous.

Agents with routes R_1 and R_2 and with walks $f^{(1)}$ and $f^{(2)}$ meet at time t, if points $f^{(1)}(t)$ and $f^{(2)}(t)$ are identical. A rendezvous is guaranteed for routes R_1 and R_2, if the agents using these routes meet, regardless of the walks chosen by the adversary. The *cost* of the rendezvous algorithm is measured by the sum of the lengths of the trajectories of both agents from their starting locations until the time t of the rendezvous. Since the actual portions of these trajectories may vary depending on the adversary, we consider the maximum of this sum, i.e., the worst-case over all possible walks chosen for both agents by the adversary. In this paper we are looking for a rendezvous algorithm of the smallest possible cost with respect to the unknown original distance d between the agents.

1.2 Related Work

The rendezvous problem was first described in [38]. A detailed discussion of the large literature on rendezvous can be found in the excellent book [3]. Most of the results in this domain can be divided into two classes: those considering the geometric scenario (rendezvous in the line, see, e.g., [7, 8, 23], or in the plane, see, e.g., [5, 6]), and those discussing rendezvous in graphs, e.g., [2, 4]. A generalization of the rendezvous problem is that of gathering [21, 27–29, 35, 41], when more than two agents have to meet in one location.

If graphs are unlabeled, deterministic rendezvous requires breaking symmetry, which can be accomplished either by allowing marking nodes or by labeling

the agents. Deterministic rendezvous with anonymous agents working in unlabeled graphs but equipped with tokens used to mark nodes was considered e.g., in [33]. In [43] the authors studied gathering of many agents with unique labels. In [17, 31, 40] deterministic rendezvous in graphs with labeled agents was considered. However, in all the above papers, the synchronous setting was assumed. Asynchronous gathering under geometric scenarios has been studied, e.g., in [13, 21, 37] in different models than ours: agents could not remember past events, but they were assumed to have at least partial visibility of the scene. The first paper to consider deterministic asynchronous rendezvous in graphs was [16]. The authors concentrated on complexity of rendezvous in simple graphs, such as the ring and the infinite line. They also showed feasibility of deterministic asynchronous rendezvous in arbitrary finite connected graphs with *known* upper bound on the size. Further improvements of the above results for the infinite line were proposed in [39]. Gathering many robots in a graph, under a different asynchronous model and assuming that the whole graph is seen by each robot, has been studied in [28, 29].

The assumption that the distributed mobile entities know their initial location in the geometric environment was considered in the past, e.g., in the context of geometric routing, see, e.g., [1, 9, 32, 34], where it is typically assumed that the source node knows the position of the destination as well as its own position, or broadcasting [19, 20], where the position awareness of only the broadcasting node is admitted. Such assumption, partly fueled by the availability and the expansion of the Global Positioning System (GPS), is sometimes called *location awareness* of agents or nodes of the network, and it often leads to better bounds of the proposed solutions.

An alternative approach to location awareness is adopted in *network design* where the nodes of the network are preprocessed in order to enable or to speed up a certain distributed process. For example, in the context of graph exploration it was shown in [11] that a suitable *a priori* coloring of the nodes with 3 colors provides a graph environment that can be explored by an agent equipped with a constant size memory. In contrast, if the preprocessing is not allowed it is known [22] that an agent requires $\Omega(\log n)$ bits of memory to be able to explore all graphs of order n. Another approach to network design consists in graph preprocessing by setting the port numbers, so that the graph exploration is easy, i.e., with constant memory, e.g. see [24, 26] or memoryless agents, [14, 18, 30].

The efficient rendezvous solutions proposed in this paper rely directly on the use of *space-covering sequences* introduced recently in [12]. The space-covering sequences are close relatives to *space-filling curves* studied extensively in the literature, see, e.g., [10, 25, 36, 42]. The space-filling curves visit every point in an infinite grid exactly once. Gotsman and Lindenbaum pointed out in [25] that for any space-filling curve there always exist some close points in the grid that are arbitrarily far apart on the space-filling curve, i.e., these type of curves fail in preserving locality in the worst case. The deficiency of space-filling curves comes from the assumption that they must visit each point in a discrete 2D-space exactly once. The authors of [12] propose a novel structure of *space-covering*

sequences that traverse points in a discrete 2D-space repeatedly. They show that for any two points located at an arbitrary distance d in the 2D-space there are well defined and efficiently computable instances of these two points in the sequence at distance $O(d^{2+\varepsilon})$ apart, for any positive constant ε.

1.3 Our Results

In this paper, we design efficient rendezvous algorithms for anonymous agents asynchronously moving in multidimensional infinite grids. In section 2, we consider the location aware agents, i.e., agents knowing the Cartesian coordinates of their own initial positions in the integer grid. We give an almost optimal $O(d^{\delta}\text{polylog } d)$ rendezvous algorithm where δ is the space dimension of the grid. Our approach, applied to the 2-dimensional grid, results in an $O(d^2\text{polylog } d)$ algorithm, improving the recent main result from [12], i.e., a $O(d^{2+\varepsilon})$ upper bound. This may be seen as an exponential improvement on the deficiency factor that leads to an almost optimal solution since a straightforward lower bound on the worst case distance is $\Omega(d^2)$. The consequence of our approach is the design of the more efficient and simpler space-covering sequences introduced in [12]. In section 3 we show how this approach may be applied to the case of rendezvous of two agents with positive visibility radii, moving in a Euclidean space. Again each agent is aware of its original position (but not the position of the other agent). Finally, in section 4, we show that it is possible to set the port numbers of the integer grid in the δ-dimensional space to achieve an efficient rendezvous. In this case, the anonymous agents are not location aware.

2 Rendezvous Algorithm for Location Aware Agents in the Grid

In this section we assume that the grid belongs to the Euclidean δ-dimensional space $\mathcal{F}$ and each agent knows the integer Cartesian coordinates of its initial position in the space. We assume that the agents have coherent compasses and a common unit of length permitting to refer to the same system of Cartesian coordinates.

The general idea of our approach is the following. The rendezvous algorithm constructs the trajectory of the agents following the integer grid lines of the Euclidean space. Hence each such line contains points $(x_1, x_2, \ldots, x_\delta)$, where x_i, for some fixed $1 \leq i \leq \delta$ is any real value, and each x_j, for $j \neq i$, $1 \leq j \leq \delta$ is some integer. We will define an infinite sequence of grids, each grid inducing a partition of the space into δ-dimensional hypercubes. The size of these hypercubes increases when following this sequence of partitions. We then define a directed acyclic graph whose nodes are the hypercubes of all such partitions, with arcs between intersecting hypercubes belonging to consecutive partitions (the arcs are directed from the larger to the smaller hypercubes). For any initial position p of the agent, we consider an infinite tree $T(p)$ containing all hypercubes of the family containing point p, such that for each node in $T(p)$ its successors

are also in $T(p)$ and if the same hypercube is obtained this way more than once it is duplicated so that $T(p)$ remains a tree. The route produced for each agent will correspond to the upward movement along $T(p)$ interleaved with some depth-first-search type traversals of the siblings of the newly visited node. We show that the two agents will eventually visit some hypercube, belonging to both their respective trees, which contains the two initial positions of the agents. The route corresponding to the visit of such a hypercube will result in rendezvous.

2.1 Euclidean Space Partitions Induced by Multidimensional Grids

We consider an infinite sequence of partitions $\Pi = \pi_1, \pi_2, \ldots$ of the grid G_δ into hypercubes of dimension δ in $\mathcal{F}$. For the sake of simplicity, in the rest of the paper, we will use the term *hypercube* for a hypercube of dimension δ. The corners of hypercubes in π_i are points $u = (u_1, u_2, \ldots, u_\delta)$ such that $\forall j \in \{1, 2, \ldots, \delta\}, u_j = 2^{i-1} + k2^i$ for some integer k. Each grid π_i partitions space $\mathcal{F}$ into hypercubes of side length 2^i. To assure that each π_i forms an exact partition, we assume that each hypercube H contains, besides its interior points, the corner v having maximum coordinates, as well as all open f-faces containing v, for $f = 1, 2, \ldots, \delta - 1$. Such corner v is called the *reference point* of the hypercube containing it. For example, a 3-dimensional hypercube, having points $(0, 0, 0)$ and $(1, 1, 1)$ as its corners, is a union of its reference point $v = (1, 1, 1)$, the three open edges of the cube incident to v, the three open square faces incident to v, and the interior of the cube.

Lemma 1. *For positive integers i and k, such that $i \geq k + 1$, any hypercube located in partition π_i intersects $(2^k + 1)^\delta$ hypercubes belonging to partition π_{i-k}.*

Proof. Let $\mathcal{C}$ be a hypercube drawn from partition π_i. Consider any edge e of $\mathcal{C}$ along dimension $l \in \{1, \ldots, \delta\}$. This edge is of length 2^i. Due to the definition of Π the edge e cannot be aligned with edges in partition π_{i-k} and each endpoint of e is located in the centre of some hypercube in this partition. Since hypercubes in partition π_{i-k} are of size 2^{i-k}, the edge e must penetrate (including its endpoints) exactly $\frac{2^i}{2^{i-k}} + 1 = 2^k + 1$ hypercubes in π_{i-k}. Finally, since this phenomenon applies to every dimension, the number of hypercubes in partition π_{i-k} intersected by a hypercube in partition π_i is $(2^k + 1)^\delta$. □

By Γ we denote the subsequence $\gamma_1 = \pi_{i_1}, \gamma_2 = \pi_{i_2}, \ldots$, of the above sequence of partitions, such that the indices i_j are defined by the recurrence:

$$\begin{aligned} i_1 &= 1 \\ i_{j+1} &= i_j + \max\{1, \lceil \log i_j \rceil\} \end{aligned}$$

We consider the infinite, directed acyclic graph T, whose nodes are the hypercubes of the partitions $\gamma_1, \gamma_2, \ldots$ and there is a directed edge in T from hypercube P to Q, if

- P and Q are from two consecutive grids γ_k and γ_{k-1}, respectively, and
- P and Q have a nonempty intersection

For any point p of the Euclidean space, we call the *ascending path* $A_k(p)$ the path of T formed of the hypercubes $S_1(p), S_2(p), \ldots, S_k(p)$, such that each hypercube $S_i(p)$ belongs to grid γ_i and $p \in S_i(p)$. By $T_k(p)$ we denote the tree containing the ascending path $A_k(p)$, rooted at $S_k(p)$, obtained from T in such a way that, each time a hypercube has more than one predecessor in T, it is split (duplicated together with all its succession class) so eventually a tree is obtained. By $T(p)$ we denote the infinite tree obtained in the similar way.

Lemma 2. *For a given point p of the Euclidean space and for $s \geq 4$, any hypercube of size $s \cdot 2^{\lceil \log \log s \rceil}$ belonging to $T(p)$ has $(2^{\lceil \log \log s \rceil} + 1)^\delta$ children.*

Proof. The proof follows directly from the definition of $T(p)$ and from Lemma 1. □

Lemma 3. *For any pair of points p_1, p_2 at distance d in the Euclidean space and any $\delta + 1$ partitions $\gamma_{j_1}, \gamma_{j_2}, \ldots, \gamma_{j_{\delta+1}}, \in \Pi$, each one composed of hypercubes of size at least $4d$, there exists a hypercube of one of the partitions which contains both points p_1, p_2.*

Proof. First, we have to show the following claim.

Claim. Let H_1 and H_2 be two hyperplanes (of dimension $\delta - 1$) in $\mathcal{F}$ separating hypercubes in distinct partitions of $\gamma_{j_1}, \gamma_{j_2}, \ldots, \gamma_{j_{\delta+1}}$. If H_1 and H_2 are parallel then they are at distance at least $2d$ of each other.

Proof of the claim. Let H_1 and H_2 be hyperplanes separating hypercubes in γ_i and γ_j respectively. H_1 is defined by $x_l = 2^{i-1} + k2^i$ for some $l \in \{1, 2, \ldots, \delta\}$ and $k \in \mathbb{Z}$. Similarly, H_2 is defined by $x_{l'} = 2^{j-1} + k'2^j$ for some $l' \in \{1, 2, \ldots, \delta\}$ and $k' \in \mathbb{Z}$). Notice that the normal vector of H_1 is parallel to the x_l-axis and that the normal vector of H_2 is parallel to the $x_{l'}$-axis. H_1 and H_2 are parallel if and only if $l = l'$. Assume without loss of generality that $i < j$. The distance between the hyperplanes is equal to :

$$\begin{aligned} dist(H_1, H_2) &= |(2^{i-1} + k2^i) - (2^{j-1} + k'2^j)| \\ &= 2^{i-1}|(1 + 2k) - 2^{j-i}(1 + 2k')| \end{aligned}$$

Notice that $|(1+2k)-2^{j-i}(1+k'2k')|$ is an integer since k, k' and 2^{j-i} are integers. Moreover, $|(1 + 2k) - 2^{j-i}(1 + k'2k')|$ is odd and so it is greater than zero, since $(1 + 2k)$ is odd and $2^{j-i}(1 + 2k')$ is even. It follows that $dist(H_1, H_2) \geq 2^{i-1}$. Finally, since the size of the hypercubes in γ_i is $2^i \geq 4d$, the hyperplanes are at distance at least $2d$. This ends the proof of the claim.

Assume, by contradiction, that there are two points p_1, p_2 at distance d that are not in the same hypercube in any partition in $\gamma_{j_1}, \gamma_{j_2}, \ldots, \gamma_{j_{\delta+1}}$. Since p_1 and p_2 are not in the same hypercube in each γ_{j_i}, there is at least one hyperplane H_i separating p_1 and p_2, i.e., the segment joining p_1 and p_2 intersects H_i. Let $\mathcal{H}$ be the set $\{H_1, H_2, \ldots, H_{\delta+1}\}$. There are at least two parallel hyperplanes H and H'

in $\mathcal{H}$ since the normal vector of each H_i is parallel to an axis of the space $\mathcal{F}$. The segment s joining p_1 and p_2 intersects H in u and H' in v. The distance between p_1 and p_2 is at least the distance between u and v, hence, at least the distance between H and H'. However, by the Claim, H and H' are at distance at least $2d$, a contradiction. It follows, that there exists a hypercube in one of the partitions containing both points p_1 and p_2. □

2.2 The Algorithm

In the beginning the agent is at its original position p. The agent calls procedure `Initialize` (line 2) to move to the reference point of the hypercube H_1 from γ_1 (the lowest level grid of the construction) containing p. H_1 is the starting point in the ascending path $A(p)$ of the infinite tree $T(p)$, represented also by the single-node tree $T_1(p)$. The execution of the algorithm corresponds to the traversal of the ascending path $A(p)$, constructing iteratively the sequence of trees $T_1(p), T_2(p), \ldots$. We say that, when an agent advances on the ascending path to a new hypercube H_{i+1}, which is the root of the tree $T_{i+1}(p)$, the agent *explores* the new hypercube H_{i+1}. The exploration of a new hypercube is an attempt to meet the other agent, which, at this time, may be exploring the same hypercube or one of its ancestors.

The exploration of the hypercube H_{i+1} is made during the i-th iteration of the main loop of the `RV` algorithm. At the beginning of the i-th iteration the agent is placed in the reference point of hypercube H_i – the root of $T_i(p)$. The first phase of the loop consists in moving to the reference point of H_{i+1}. Iteratively, each preceding sibling of H_i is traversed in the backward sense by `Traverse` procedure and the previous sibling is reached by the corresponding connector (lines 7-9). Eventually, when the smallest sibling is reached and traversed, the connector to its parent is traversed (line 11), leading to the reference point of H_{i+1}. Finally, H_{i+1} is traversed in the forward sense by `Traverse` procedure (line 13).

The entire route of the agent is the concatenation of a sequence of connectors between different hypercubes of the construction (most of them repeated many times). Each such connector either joins consecutive siblings or it joins the first child (the smallest among all siblings) and its parent. In order to use efficient (i.e., short) connectors, the children of each node are arranged in such a way that the first child contains the reference point of its parent, and the consecutive siblings are adjacent hypercubes, i.e., they share a $(\delta - 1)$-face, cf. Lemma 4 below. Then, when children are of diameter r, all connectors between them or joining them to parents are of length $O(r)$.

The algorithm uses a variable *height* which is the height of the hypercube in the tree $T(p)$, which is currently being explored. The variable H corresponds to the hypercube at the reference point of which the agent currently is. The operation `Connect`(H, C) generates a connector from the reference point of the current hypercube H to the reference point of the hypercube C. Notice

Algorithm RV(point p)
1 $height \leftarrow 1$;
2 $H \leftarrow$ Initialize(v);
3 Traverse($H, 1, forward$);
4 **repeat**
5 $P \leftarrow$ Parent(H, p)
6 Traverse($H, height, backward$);
7 **while** PrevSib(H, P) exists **do**
8 Connect(H, PrevSib(H, P)); $H \leftarrow$ PrevSib(H, P);
9 Traverse($H, height, backward$);
10 $height \leftarrow height + 1$;
11 Connect(H, P);
12 $H \leftarrow P$;
13 Traverse($H, height, forward$);
14 **until** rendezvous

that C must be either the first child, parent, next sibling or previous sibling of H. The next sibling and previous sibling of a node can be obtained by procedure NextSib(H, P) and PrevSib(H, P) respectively. Observe that these procedures use a parent P of the hypercube H as a parameter since a hypercube can have one of several parents and so one of several potential next or previous siblings. The procedure Parent(H, p) returns the parent of H containing point p.

The procedure Traverse($H, height, sense$) performs essentially the depth-first-search traversal of the subtree of $T(p)$ rooted at hypercube H at height $height$. The parameter $sense$ permits to indicate an orientation for the traversal of $T(p)$, either $backward$ or $forward$. The traversal of a hypercube performed with parameter $sense$ equal to $backward$ is the reverse of the traversal done with $sense$ equal to $forward$. The procedure TraverseUnit performs the traversal of a hypercube of size 2 in γ_1.

procedure Traverse($H, height, sense$)
1 **if** ($height = 1$) **then**
2 TraverseUnit($H, 1, sense$)
3 **else**
4 $C \leftarrow$ FirstChild(H);
5 Connect(H, C);
6 **while** NextSib(C, H) exists **do**
7 **if**($sense = forward$) **then**
8 Traverse($C, height - 1, forward$);
9 Connect(C, NextSib(C, H)); $C \leftarrow$ NextSib(C, H);
10 Traverse($C, height - 1, sense$);
11 **while** PrevSib(C, H) exists **do**
12 Connect(C, PrevSib(C, H)); $C \leftarrow$ PrevSib(C, H);
13 **if**($sense = backward$) **then**
14 Traverse($C, height - 1, backward$);
15 Connect(C, H);

Lemma 4. *The total length of the connectors linking the space-covering of the children of a hypercube H of size $s \cdot 2^{\lceil \log \log s \rceil}$ is $O(s \cdot 2^{\delta(\log \log s+1)})$.*

Proof. Note first that the children of H form also a hypercube of size $s(2^{\lceil \log \log s \rceil} + 1)$ composed of smaller hypercubes of size s (this is not a hypercube which is an element of any of the grid partitions considered). In each hypercube of size s, the corner with the largest coordinates is chosen as its *representative*. The representatives can be interpreted as nodes of another hypercube H_r of size $s \cdot 2^{\lceil \log \log s \rceil}$ in which edges are of length s. It is possible to find a Hamiltonian path in H_r connecting the representatives, sharing the $(\delta-1)$-faces, and further use the edges of this tour as connectors. Since the number of representatives is $2^{\delta(\log \log s+1)}$ and the edges of the tour are of length s, the length of the Hamiltonian path and in turn the total length of the connectors is bounded by $O(s \cdot 2^{\delta(\log \log s+1)})$. □

Lemma 5. *The length of the trajectory of the agent corresponding to the exploration of a hypercube of size s is $O(s^\delta \log s)$.*

Proof. A hypercube of size $s \cdot 2^{\lceil \log \log s \rceil}$ has $(2^{\lceil \log \log s \rceil}+1)^\delta$ children by Lemma 2 and the total length of the connectors of its children is $O(s \cdot 2^{\delta(\log \log s+1)})$ by Lemma 4. Hence, the function $\lambda(s)$ defining the length of the portion of the trajectory corresponding to the exploration of a hypercube of size s is given by the following recurrence :

$$\begin{aligned} \lambda(1) &= c_1 \\ \lambda(s \cdot 2^{\lceil \log \log s \rceil}) &= (2^{\lceil \log \log s \rceil} + 1)^\delta \lambda(s) + O(s \cdot 2^{\delta(\log \log s+1)}) \end{aligned}$$

Let $c_2 \geq 1$ be a constant whose exact value will be defined later. We prove by induction that there is a constant c_3, such that for $s \geq c_2$, we have the following property:

$$\mathcal{P}_s : \lambda(s) \leq c_3 s^\delta \log s$$

We assume that the constant c_3 is large enough such that $c_3 \geq \frac{\lambda(c_2)}{c_2^\delta \log c_2}$. Hence, $\mathcal{P}_{c_2}$ is true. Now, we assume by induction that $\mathcal{P}_s$ is true. We have:

$$\begin{aligned} \lambda(s \cdot 2^{\lceil \log \log s \rceil}) &\leq (2^{\lceil \log \log s \rceil} + 1)^\delta \lambda(s) + c_4 s \cdot 2^{\delta(\log \log s+1)} \\ &\leq (2^{\lceil \log \log s \rceil} + 1)^\delta c_3 s^\delta \log s + c_4 s \cdot 2^{\delta(\log \log s+1)} \end{aligned}$$

Hence to prove that $\mathcal{P}_{s2^{\lceil \log \log s \rceil}}$ is true, it is sufficient to prove that:

$$\begin{aligned} (2^{\lceil \log \log s \rceil} + 1)^\delta c_3 s^\delta \log s + c_4 s \cdot 2^{\delta(\log \log s+1)} &\leq c_3 s^\delta 2^{\delta \lceil \log \log s \rceil} \log(s \cdot 2^{\lceil \log \log s \rceil}) \\ \frac{(2^{\lceil \log \log s \rceil} + 1)^\delta}{2^{\delta \lceil \log \log s \rceil}} \log s + \frac{c_4 2^\delta}{c_3 s^{\delta-1}} &\leq \log s + \log \log s \end{aligned}$$

Notice that:

$$\begin{aligned} (2^{\lceil \log \log s \rceil} + 1)^\delta &= 2^{\delta \lceil \log \log s \rceil} + \sum_{k=0}^{\delta-1} \binom{\delta}{k} 2^{k \lceil \log \log s \rceil} \\ &\leq 2^{\delta \lceil \log \log s \rceil} + 2^\delta 2^{(\delta-1)\lceil \log \log s \rceil} \end{aligned}$$

Hence it is sufficient to prove that:

$$\frac{2^{\delta\lceil\log\log s\rceil}+2^{\delta}2^{(\delta-1)\lceil\log\log s\rceil}}{2^{\delta\lceil\log\log s\rceil}}\log s+\frac{c_4 2^{\delta}}{c_3 s^{\delta-1}}\leq \log s+\log\log s$$

$$\frac{2^{\delta}}{2^{\lceil\log\log s\rceil}}\log s+\frac{c_4 2^{\delta}}{c_3 s^{\delta-1}}\leq \log\log s$$

$$2^{\delta}+\frac{c_4 2^{\delta}}{c_3 s^{\delta-1}}\leq \log\log s \quad \text{since } \log s\leq 2^{\lceil\log\log s\rceil}$$

$$2^{\delta}+\frac{2^{\delta}}{s^{\delta-1}}\leq \log\log s \quad \text{assuming that } c_3\geq c_4$$

The left term tends to 2^{δ} for s going to the infinity. Since the right term tends to the infinity and is positive for s going to the infinity, there must exist a constant c_5 such that the inequality is satisfied for all $s \geq c_5$. By taking $c_2 = c_5$ and $c_3 = \max\left\{c_4, \frac{\lambda(c_2)}{c_2^{\delta}\log c_2}\right\}$, all the assumptions made during the proof are fulfilled. Finally, $\mathcal{P}_s$ is true for every s by induction. □

Theorem 1. *Suppose that a pair of agents is originally placed at any two points p_1, p_2 at distance at most d in the δ-dimensional infinite integer grid. Consider the trajectories computed by* `RV`*(p_1) and* `RV`*(p_2). Any asynchronous walk along these trajectories results in the rendezvous of the agents after traversing trajectories of length $O(d^{\delta}\log^{\delta^2+\delta+1} d)$.*

Proof. We show first that all the connectors may be correctly computed by the agent using `RV` algorithm. Consider first the connectors used in algorithm `RV`. Suppose that the agent stores in its memory the global variable *height* - the height of the hypercube actually explored, the coordinates of its initial position p as well as the number of iterations of the loop from lines 7-9 that have already been performed within the current iteration of the main loop from lines 4-14. From this information and from the knowledge how the siblings at each level have been arranged, the agent may compute in which direction (i.e., positive or negative direction of some of the δ axes of the grid) the connector to the next or the previous sibling goes. The length of this connector is a function of *height* solely. Similarly the connector between a parent and its first child is easily computed since the position of the reference point of the first child is easily computed from the reference point of the parent.

In order to compute correctly the connectors computed in the `Traverse` procedure, we suppose that for each active call of `Traverse` we keep on the recursive stack the local variable *height* as well as the number of the sibling in lines 8 and 14 for which the current recursive call from its parent node is made. The data stored on the recursive stack and the knowledge how the siblings at each level are arranged allow to compute the direction and the length of each of the connectors.

In order to consider the moment of termination of the algorithm take $(\delta+1)$ consecutive grids $\gamma_k, \gamma_{k+1}, \ldots, \gamma_{(k+\delta)}$ of Γ, such that γ_k is the first grid of Γ containing hypercubes of size at least $4d$. By Lemma 3, one of the grid partitions contains some hypercube H containing both points p_1 and p_2. Suppose, by

symmetry, that the agent originally placed at point p_1 is the first one to complete the traversal of H. Since the other agent is all the time situated inside the segment of the trajectory corresponding to H the agents must meet.

The largest such hypercube H (the one belonging to $\gamma_{k+\delta}$) has the size in $O(d \log^{\delta+1} d)$. By an application of Lemma 5 for $s = O(d \log^{\delta+1} d)$, we obtain the statement of the theorem. □

To show that the above result is almost optimal observe that there are $\Omega(d^\delta)$ integer grid points in the d-neighborhood of any point. Since the adversary may hold one of the agents for arbitrary long time close to its initial position, the second agent must eventually explore its entire d-neighborhood using a trajectory of length $\Omega(d^\delta)$.

3 Location Aware Agents in Euclidean Space with Non-zero Visibility

We now consider a model in which an agent at point p in the δ-dimensional Euclidean space $\mathcal{F}$ *sees* all points in the δ-dimensional ball of radius $r \geq 1$ centered at p. We say that the agent has *visibility range* r. In this model, rendezvous occurs when *one* of the agents sees the other. Agents may have different visibility radii, thus it is not necessary that they both see each other at the same time. Each agent is aware of the coordinates of its own location at any time.

The idea is to apply the algorithm developed in the previous section for location-aware agents with zero visibility, but choosing the size of the lowest-level hypercubes in such a way that it is as large as possible and whenever an agent traverses the portion of its trajectory that corresponds to the lowest level hypercubes, it sees all of their interior points.

Each agent, knowing its visibility radius r, determines the maximum integer k (possibly negative) such that $k \leq \log \frac{r}{\sqrt{\delta}}$. Note that this implies $r \geq 2^k\sqrt{\delta}$, therefore the visibility radius of the agent is at least the diameter of a hypercube of size 2^k. This implies that an agent visiting a node of this hypercube sees all the interior points. Having computed the value of k, the agent moves from its original position to the closest reference point of some hypercube of size 2^k of the appropriate layer in the hierarchy Π. From that point, it starts executing the algorithm of the previous section with a unit hypercube of size 2^k.

In a more general setting, the two agents have different visibility radii r_1 and r_2. As explained above, they compute their respective values k_1 and k_2 and they start executing the rendezvous algorithm with hypercubes of respective sizes 2^{k_1} and 2^{k_2}. Essentially, the two agents ascend the same tree-like structure T as before, except that they start at a higher level which is possibly not the same for the two agents. In any case, one of the agents will start exploring first a hypercube that is guaranteed to contain both starting points, according to Lemma 3, and thus it will be able to see the other agent effecting rendezvous.

Due to the modification of the size of the unit hypercubes of the agents, the algorithm behaves in the worst case as if the distance between the starting points

of the agents is scaled by $2^{-k} < \frac{2\sqrt{\delta}}{r}$, where $r = \min\{r_1, r_2\}$. This results in a trajectory of length $O((\frac{2d\sqrt{\delta}}{r})^{\delta} \log^{\delta^2+\delta+1}(\frac{2d\sqrt{\delta}}{r})) = O((\frac{d}{r})^{\delta} \log^{\delta^2+\delta+1}(\frac{d}{r}))$.

4 Setting the Grid Port Numbers for Efficient Rendezvous

We now sketch how to convert the algorithm we developed in Section 2 into an algorithm suitable for anonymous grids with local port numbering. The idea is to put labels on the nodes of the grid, and then using these labels simulate the rendezvous algorithm in the absence of location awareness on the part of the agents. We then discuss how to manipulate the local port numbers at each node so that the agent is able to extract each node's label by performing a short exploration of its neighborhood.

Omitting the details of the labeling process due to lack of space, we just mention that for dimension δ we need $O(2^{\delta})$ different labels. By choosing carefully the order of the children of each hypercube, we can make sure that the labels provide the agent with enough information to execute the rendezvous algorithm. We now elaborate on the port setting scheme that we use. Let $\mathbf{x} = (x_1, \ldots, x_\delta)$ be an arbitrary node in the δ-dimensional grid. For i in the range $1 \leq i \leq \delta$, we assign port number i of node $\mathbf{x}$ to the edge connecting node $\mathbf{x}$ to node $(x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_\delta)$. The remaining port numbers from $\delta + 1$ to 2δ may be assigned in any of $\delta!$ ways to the remaining edges outgoing from $\mathbf{x}$. This assignment is described by a permutation $\sigma_{\mathbf{x}}$ of $\{1, \ldots, \delta\}$, such that for all i in the range $1 \leq i \leq \delta$, port number $\delta + \sigma_{\mathbf{x}}(i)$ of node $\mathbf{x}$ is assigned to the edge connecting node $\mathbf{x}$ to node $(x_1, \ldots, x_{i-1}, x_i - 1, x_{i+1}, \ldots, x_\delta)$.

Now, suppose that the agent finds itself at node $\mathbf{x}$. It is possible to recover the permutation $\sigma_{\mathbf{x}}^{-1}$, thus also the permutation $\sigma_{\mathbf{x}}$, in the following way: for each i in the range $1 \leq i \leq \delta$, (a) follow port number $\delta + i$ of node $\mathbf{x}$, (b) observe the port number j $(1 \leq j \leq \delta)$ that corresponds to the edge just traversed in the local port order of the new node, (c) follow port number j of the new node to go back to the original node $\mathbf{x}$. Then, the agent can deduce that $\sigma_{\mathbf{x}}^{-1}(i) = j$.

Knowing $\sigma_{\mathbf{x}}$, the agent knows exactly which port number corresponds to each direction out of node $\mathbf{x}$ in the δ-dimensional grid. Furthermore, the port setting scheme we just described endows each node with one out of $\delta!$ distinct labels. Therefore, for large enough dimension δ, it is possible to encode the required node labels using the available port number permutations.

We modify the rendezvous algorithm so that whenever an agent arrives at a node $\mathbf{x}$, it performs the following trajectory: it follows port numbers $\delta+1, \ldots, 2\delta$, in that order, and then if follows port numbers $2\delta, \ldots, \delta + 1$, in that order, each time returning to node $\mathbf{x}$. During the first phase of this trajectory (ports $\delta + 1, \ldots, 2\delta$), the agent recovers the permutation $\sigma_{\mathbf{x}}$, and thus the label of $\mathbf{x}$. This modification increases the length of the trajectory of each agent by a factor of $O(\delta)$. Moreover, since the interjected trajectory is a palindrome and both agents perform it at each node they visit, in the worst case they will meet while

exploring the neighborhood of the same node as the one on which they would meet using the original algorithm.

Thus, we arrive at the following theorem:

Theorem 2. *For large enough δ, it is possible to preprocess the local port numbers in the δ-dimensional infinite grid so that two agents originally placed at distance at most d in the grid can achieve rendezvous after traversing trajectories of size $O(d^\delta \log^{\delta^2+\delta+1} d)$.*

For smaller number of dimensions δ, it is still possible to encode all the required labels in the grid, by coarsening the grid on which the algorithm is executed and using more than one node of the original grid to encode the label of a node in the coarsened grid. Details are omitted due to lack of space.

5 Final Remarks

Several interesting questions remain unanswered. E.g., is it possible to design an optimal $O(d^\delta)$ rendezvous algorithm in δ-dimensional grids? Is it possible to extend the location aware approach to some classes of graphs other than grids? In Section 4 there seems to be a lot of freedom when choosing possible port arrangements. Is it possible to use a more sophisticated port arrangements so that the mobile agent uses $o(\log d)$ or even a constant number of its memory bits?

References

1. Abraham, I., Dolev, D., Malkhi, D.: LLS: a locality aware location service for mobile ad hoc networks. In: Proc. DIALM-POMC 2004, pp. 75–84 (2004)
2. Alpern, S.: The rendezvous search problem. SIAM J. on Control and Optimization 33, 673–683 (1995)
3. Alpern, S., Gal, S.: The theory of search games and rendezvous. Int. Series in Operations research and Management Science, vol. 55. Kluwer Academic Publisher, Dordrecht (2002)
4. Alpern, J., Baston, V., Essegaier, S.: Rendezvous search on a graph. Journal of Applied Probability 36, 223–231 (1999)
5. Anderson, E., Fekete, S.: Asymmetric rendezvous on the plane. In: Proc. 14th Annual ACM Symp. on Computational Geometry, pp. 365–373 (1998)
6. Anderson, E., Fekete, S.: Two-dimensional rendezvous search. Operations Research 49, 107–118 (2001)
7. Baston, V., Gal, S.: Rendezvous on the line when the players' initial distance is given by an unknown probability distribution. SIAM J. on Control and Optimization 36, 1880–1889 (1998)
8. Baston, V., Gal, S.: Rendezvous search when marks are left at the starting points. Naval Res. Log. 48, 722–731 (2001)
9. Bose, P., Morin, P., Stojmenovic, I., Urrutia, J.: Routing with guaranteed delivery in ad hoc wireless networks. Wireless Networks 7(6), 609–616 (2001)

10. Buchin, K.: Constructing Delaunay Triangulations along Space-Filling Curves. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 119–130. Springer, Heidelberg (2009)
11. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. ACM Transactions on Algorithms 4(4), 1–18 (2008)
12. Collins, A., Czyzowicz, J., Gasieniec, L., Labourel, A.: Tell me where I am so I can meet you sooner: Asynchronous rendezvous with location information. In: Proc. of ICALP 2010 (2010)
13. 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)
14. Czyzowicz, J., Dobrev, S., Gasieniec, L., Ilcinkas, D., Jansson, J., Klasing, R., Lignos, I., Martin, R.A., Sadakane, K., Sung, W.-K.: More Efficient Periodic Traversal in Anonymous Undirected Graphs. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 167–181. Springer, Heidelberg (2010)
15. Czyzowicz, J., Labourel, A., Pelc, A.: How to meet asynchronously (almost) everywhere. In: Proc. of SODA 2010, pp. 22–30 (2010)
16. 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)
17. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)
18. Dobrev, S., Jansson, J., Sadakane, K., Sung, W.-K.: Finding short right-hand-on-the-wall walks in graphs. In: Pelc, A., Raynal, M. (eds.) SIROCCO 2005. LNCS, vol. 3499, pp. 127–139. Springer, Heidelberg (2005)
19. Emek, Y., Gasieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in UDG radio networks with unknown topology. Distributed Computing 21(5), 331–351 (2009)
20. Emek, Y., Kantor, E., Peleg, D.: On the effect of the deployment setting on broadcasting in Euclidean radio networks. In: Proc. PODC 2008, pp. 223–232 (2008)
21. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous oblivious robots with limited visibility. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 247–258. Springer, Heidelberg (2001)
22. 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)
23. Gal, S.: Rendezvous search on the line. Operations Research 47, 974–976 (1999)
24. Gasieniec, L., Klasing, R., Martin, R.A., Navarra, A., Zhang, X.: Fast periodic graph exploration with constant memory. J. on Computer Systems and Sciences 74(5), 808–822 (2008)
25. Gotsman, C., Lindenbaum, M.: On the metric properties of discrete space-filling curves. IEEE Transactions on Image Processing 5(5), 794–797 (1996)
26. Ilcinkas, D.: Setting Port Numbers for Fast Graph Exploration. Theor. Comput. Sci. 401(1-3), 236–242 (2008)
27. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self stabilizing mutual exclusion. In: Proc. PODC'90, pp. 119–131 (1990)
28. Klasing, R., Kosowski, A., 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)
29. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theoretical Computer Science 390, 27–39 (2008)

30. Kosowski, A., Navarra, A.: Graph Decomposition for Improving Memoryless Periodic Exploration. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 501–512. Springer, Heidelberg (2009)
31. Kowalski, D., Malinowski, A.: How to meet in anonymous network. Theoretical Computer Science 399, 141–156 (2008)
32. Kozma, G., Lotker, Z., Sharir, M., Stupp, G.: Geometrically aware communication in random wireless networks. In: Proc. PODC 2004, pp. 310–319 (2004)
33. Kranakis, E., Krizanc, D., Santoro, N., Sawchuk, C.: Mobile agent rendezvous in a ring. In: Proc. 23rd International Conference on Distributed Computing Systems (ICDCS 2003), pp. 592–599 (2003)
34. Kuhn, F., Wattenhofer, R., Zhang, Y., Zollinger, A.: Geometric ad-hoc routing: theory and practice. In: Proc. PODC 2003, pp. 63–72 (2003)
35. Lim, W., Alpern, S.: Minimax rendezvous on the line. SIAM J. on Control and Optimization 34, 1650–1665 (1996)
36. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. IEEE Transactions on Knowledge Data Engineering 14(1), 124–141 (2001)
37. Prencipe, G.: Impossibility of gathering by a set of autonomous mobile robots. Theoretical Computer Science 384, 222–231 (2007)
38. Schelling, T.: The strategy of conflict. Oxford University Press, Oxford (1960)
39. Stachowiak, G.: Asynchronous Deterministic Rendezvous on the Line. In: Nielsen, M., Kucera, A., Miltersen, P.B., Palamidessi, C., Tuma, P., Valencia, F.D. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 497–508. Springer, Heidelberg (2009)
40. Ta-Shma, A., Zwick, U.: Deterministic rendezvous, treasure hunts and strongly universal exploration sequences. In: Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), pp. 599–608 (2007)
41. Thomas, L.: Finding your kids when they are lost. Journal on Operational Res. Soc. 43, 637–639 (1992)
42. Xu, B., Chen, D.Z.: Density-Based Data Clustering Algorithms for Lower Dimensions Using Space-Filling Curves. In: Zhou, Z.-H., Li, H., Yang, Q. (eds.) PAKDD 2007. LNCS (LNAI), vol. 4426, pp. 997–1005. Springer, Heidelberg (2007)
43. Yu, X., Yung, M.: Agent rendezvous: a dynamic symmetry-breaking problem. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 610–621. Springer, Heidelberg (1996)

Exclusive Perpetual Ring Exploration without Chirality

Lélia Blin, Alessia Milani, Maria Potop-Butucaru, and Sébastien Tixeuil

Univ. Pierre et Marie Curie - Paris 6, LIP6-CNRS UMR 7606, France
{lelia.blin,alessia.milani,maria.gradinariu,sebastien.tixeuil}@lip6.fr

Abstract. In this paper, we study the exclusive perpetual exploration problem with mobile anonymous and oblivious robots in a discrete space. Our results hold for the most generic settings: robots are asynchronous and are not given any sense of direction, so the left and right sense (*i.e.* chirality) is decided by the adversary that schedules robots for execution, and may change between invocations of a particular robots (as robots are oblivious). We investigate both the minimal and the maximal number of robots that are necessary and sufficient to solve the exclusive perpetual exploration problem. On the minimal side, we prove that three deterministic robots are necessary and sufficient, provided that the size n of the ring is at least 10, and show that no protocol with three robots can exclusively perpetually explore a ring of size less than 10. On the maximal side, we prove that $k = n - 5$ robots are necessary and sufficient to exclusively perpetually explore a ring of size n when n is co-prime with k.

1 Introduction

We consider autonomous robots that are endowed with visibility sensors (but that are otherwise unable to communicate) and motion actuators. Those robots must collaborate to solve a collective task, namely *exclusive perpetual exploration*, despite being limited with respect to input from the environment, asymmetry, memory, etc.

The area to be explored is modeled as a graph and the exclusive perpetual exploration tasks requires every possible vertex to be visited infinitely often by every robot, with the additional constraint that no two robots may be present at the same node at the same time or may concurrently traverse the same edge of the graph.

Robots operate in *cycles* that comprise *look*, *compute*, and *move* phases. The look phase consists in taking a snapshot of the other robots positions using its visibility sensors. In the compute phase a robot computes a target destination based on the previous observation. The move phase simply consists in moving toward the computed destination using motion actuators. We consider an asynchronous computing model, i.e., there may be a finite but unbounded time between any two phases of a robot's cycle. Asynchrony makes the problem hard

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 312–327, 2010.

since a robot can decide to move according to an old snapshot of the system and different robots may be in different phases of their cycles at the same time.

Moreover, the robots that we consider here have weak capacities: they are *anonymous* (they execute the same protocol and have no mean to distinguish themselves from the others), *oblivious* (they have no memory that is persistent between two cycles), and have no compass whatsoever (they are unable to agree on a common direction or orientation in the ring).

Related work. While the vast majority of literature on coordinated distributed robots considers that those robots are evolving in a *continuous* two-dimensional Euclidian space and use visual sensors with perfect accuracy that permit to locate other robots with infinite precision, a recent trend was to shift from the classical continuous model to the *discrete* model. In the discrete model, space is partitioned into a *finite* number of locations. This setting is conveniently represented by a graph, where nodes represent locations that can be sensed, and where edges represent the possibility for a robot to move from one location to the other. Thus, the discrete model restricts both sensing and actuating capabilities of every robot. For each location, a robot is able to sense if the location is empty or if robots are positioned on it (instead of sensing the exact position of a robot). Also, a robot is not able to move from a position to another unless there is explicit indication to do so (*i.e.*, the two locations are connected by an edge in the representing graph). The discrete model permits to simplify many robot protocols by reasoning on finite structures (*i.e.*, graphs) rather than on infinite ones. However, as noted in most related papers [6,5,3,4,2], this simplicity comes with the cost of extra symmetry possibilities, especially when the authorized paths are also symmetric.

Assuming visibility capabilities, the two main problems that have been studied in the discrete robot model are gathering [6,5], exploration with stop [3,4,2], and exclusive perpetual exploration [1]. For exploration with stop, the fact that robots need to stop after exploring all locations requires robots to "remember" how much of the graph was explored, *i.e.*, be able to distinguish between various stages of the exploration process since robots have no persistent memory. As configurations can be distinguished only by robot positions, the main complexity measure is then the number of robots that are needed to explore a given graph. The vast number of symmetric situations induces a large number of required robots. For tree networks, [4] shows that $\Omega(n)$ robots are necessary for most n-sized tree, and that sublinear robot complexity is possible only if the maximum degree of the tree is 3. In uniform rings, [3] proves that the necessary and sufficient number of robots is $\Theta(\log n)$, although it is required that the number k of robots and the size n of the ring are coprime. Note that both approaches are *deterministic*, *i.e.*, if a robot is presented twice the same situation, its behavior is the same in both cases. In [2], the authors propose to adopt a *probabilistic* approach to lift constraints and to obtain tighter bounds. They show that *four* identical probabilistic robots are necessary and sufficient to solve the exploration problem in any anonymous unoriented ring of size $n > 8$, also removing the coprime constraint between the number of robots and the size of the ring.

Most related to our work is the constrained perpetual exploration of anonymous graph presented in [1]. While this paper considers perpetual exploration instead of exploration with stop, it introduces the additional constraint that no two robots should ever concurrently be located at the same node or cross the same edge (denoted in the following as the *exclusivity* property). Due to the mutual exclusion constraints and dependently on the graph to be visited, the authors show that there exists a maximum number of robots such having a larger number of robots on the graph, makes the problem impossible to be solved. They consider a synchronous model and, contrarily to [6,5,3,4,2], they assume the robots to be endowed with sense of direction, *i.e.* they agree on the four basic directions: north, south, east, and west. In this strong model, n robots can synchronize to explore a ring of size n without violating the mutual exclusion requirements. Assuming chirality, permits to break all cases of initial symmetry since a global total order can be inferred on nodes.

Our contribution. In this paper, we initiate research about constrained perpetual exploration with mobile asynchronous, anonymous and oblivious robots in the discrete model in ring-shaped networks. Moreover, our robots are not given any sense of direction, so the left and right sense (*i.e.* chirality) is decided by the adversary that schedules robots for execution, and may change between invocations of a particular robots (as robots are oblivious). This very weak assumption preserves all usual problems related to symmetry breaking. We investigate both the minimal and the maximal number of robots that are necessary and sufficient to solve the exclusive perpetual exploration problem on a ring. On the minimal side, we prove that three deterministic robots are necessary and sufficient, provided that the size n of the ring is at least 10, and show that no protocol with three robots can exclusively perpetually explore a ring of size less than 10. On the maximal side, we prove that $n - 5$ robots are necessary and sufficient to exclusively perpetually explore a ring of size n when n is coprime with k.

2 Model

We consider a distributed system of k mobile robots scattered on a ring of n nodes u_0,u_1,..., $u_{(n-1)}$ such as u_i is connected to both $u_{(i-1)}$ and $u_{(i+1)}$. The ring is assumed to be anonymous *i.e* no labeling is available to distinguish nodes or edges. In addition, the ring is unoriented *i.e* given two neighbors, it is impossible to determine which node is on the right or on the left.

The robots are identical *i.e.*, they cannot be distinguished using their appearance and all of them execute the same protocol. Additionally, the robots are oblivious *i.e.*, they have no memory of their past actions. We assume the robots do not communicate in a explicit way. However, they have the ability to sense their environment and see the position of the other robots. Robots operate in three phase cycles: Look, Compute and Move. During the Look phase robots take a snapshot of their environment. The collected information (position of the other robots) are used in the compute phase in which robots decide to move

or to stay idle. In the last phase (move phase) they may move to one of their adjacent nodes towards the target destination computed in the previous phase.

The computational model we consider is the *CORDA model* [7] in a discreete setting, *i.e.* when a robot takes a snapshot of the network, it sees the other robots on nodes only. On the other hand, the time between Look, Compute, and Move operations is finite yet unbounded, and is decided by the adversary for each action of each robot. Thus, because of the asynchrony, different robots can execute concurrently different phases (*e.g.*, a robot can perform a look operation while another robot is moving), and a robot can use an outdated snapshot of the network to compute where to move and whether to move.

In the following we assume that initially every node of the ring contains at most one robot. A robot whose execution of Look and Compute operations leads to an actual move is said *activatable*. The positions of all robots at time t defines the system configuration at t.

Since the ring and the robots are anonymous, a robot cannot distinguish between two configurations where the relative positions of robots is the same, *e.g.*, any two configurations where all robots occupy adjacent nodes of the ring (one robot for each node) are indistinguishable for any robot despite the actual nodes occupied. For that reason, we abstract all indistinguishable configurations in a single configuration represented as a sequence of 0 (no robot occupies a node) and 1 (otherwise) which denotes the relative position of nodes in any of the above said indistinguishable configurations.

Formally,

Definition 1. *For k robots in the n-node ring, a* configuration *is an alternative (circular and non oriented) sequence of symbols* R *and* F*, indexed by integers:* R_i *stands for i consecutive nodes, each of them occupied by a robot, and* F_j *stands for j consecutive nodes free of any robot.*

For instance, a configuration $C = (\mathrm{R}_{i_1}, \mathrm{F}_{j_1}, \ldots, \mathrm{R}_{i_\ell}, \mathrm{F}_{j_\ell})$ describes the case in which the k robots are split in ℓ groups where each group of robots occupies consecutive nodes on the ring, and two groups are separated by at least one free node.

3 On the Minimal and Maximal Number of Robots for Perpetual Exploration

In this section, we provide a set of impossibility results. For sake of space, the corresponding proofs can be found in [8].

Lemma 1. *For any $n \geq 3$, perpetual exploration of the n-node ring is impossible with one robot.*

Lemma 2. *For any $n \geq 3$, perpetual exploration of the n-node ring is impossible with an even number of robots.*

Lemma 3. (Flocchini et al. [3]) *Let $k < n$. If k divides n then the exploration of the n-node ring with k robots is not possible.*

Lemma 4. *Perpetual exploration of the n-node ring is impossible with three robots for all $n < 10$.*

Lemma 5. *For any ring of size n it is impossible to solve the perpetual exploration with $n - 2$ robots, where $n \geq 2$.*

Lemma 6. *For any ring of size n, it is impossible to solve the perpetual exploration with $n - k$ robots with $2 < k \leq n$ where $n \bmod k = 0$.*

Lemma 7. *For any ring of size n, it is impossible to solve the perpetual exploration with $k = n - 3$ robots.*

Lemma 8. *For any ring of size n, it is impossible to solve the perpetual exploration with $k = n - 4$ robots.*

4 Exclusive Exploration Algorithm Using a Minimum Number of Robots

In this section we propose a distributed algorithm that achieves an exclusive perpetual exploration of a ring of size $n \geq 10$ with 3 robots. As shown, three robots are the minimal number of robots that can solve the exclusive perpetual exploration problem. We identify two types of configurations: legitimate and non legitimate. We identify the following legitimate configurations $C_0 = (\mathrm{R}_2, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_z)$, $C_1 = (\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_z)$ or $C_2 = (\mathrm{R}_2, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_z)$ with $z \notin \{0, 1, 2, 3\}$. These configurations will be referred in the following as *two-gap*, *one-two-gap* and *three-gap* configurations respectively.

When started in a legitimate configuration the protocol always moves the system in a legitimate configuration. When started in a non-legitimate configuration the protocol ensures the convergence towards a legitimate configuration. For the sake of the presentation we divide the protocol into two phases: the first phase is executed whenever the system is in a legitimate configuration while the second phase works when the protocol is in a initial configuration.

The first phase of the algorithm makes the system cycle between the legitimate configurations. A two-gap configuration moves to a one-two-gap configuration (via Rule $\mathtt{RL1}_m$), a one-two-gap configuration moves to a three-gap configuration (via Rule $\mathtt{RL2}_m$) while a three-gap configuration moves to a two-gap configuration (via Rule $\mathtt{RL3}_m$).

Legitimate Phase: $z \neq \{0, 1, 2, 3\}$	
$\mathtt{RL1}_m$ ::	$(\mathrm{R}_2, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_{z-1})$
$\mathtt{RL2}_m$ ::	$(\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_2, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_z)$
$\mathtt{RL3}_m$ ::	$(\mathrm{R}_2, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_2, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_{z+1})$

Convergence Phase takes care of the execution of the system while the initial configuration is not a legitimate configuration.

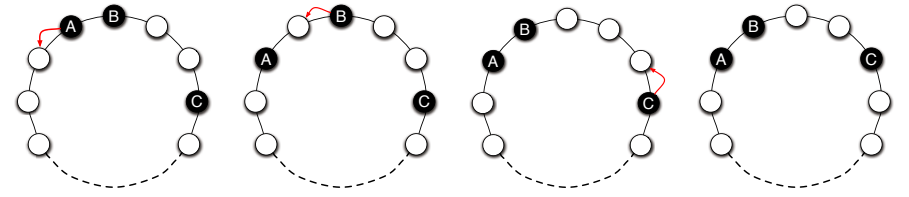


Fig. 1. Perpetual exploration with three robots

Convergence Phase. Execution starting from special configurations.			
RC1$_m$ ::	$(\mathrm{R}_2, \mathrm{F}_y, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_2, \mathrm{F}_{min(y,z)-1}, \mathrm{R}_1, \mathrm{F}_{max(y,z)+1})$	with $y \neq z \neq \{1,2,3\}$
RC2$_m$ ::	$(\mathrm{R}_1, \mathrm{F}_x, \mathrm{R}_1, \mathrm{F}_y, \mathrm{R}_1, \mathrm{F}_y) \rightarrow (\mathrm{R}_1, \mathrm{F}_x, \mathrm{R}_1, \mathrm{F}_{y-1}, \mathrm{R}_1, \mathrm{F}_{y+1})$	with $x \neq y \neq 0$
RC3$_m$ ::	$(\mathrm{R}_1, \mathrm{F}_x, \mathrm{R}_1, \mathrm{F}_y, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_1, \mathrm{F}_{x-1}, \mathrm{R}_1, \mathrm{F}_{y+1}, \mathrm{R}_1, \mathrm{F}_z)$	with $x < y < z$
RC4$_m$ ::	$(\mathrm{R}_3, \mathrm{F}_z) \rightarrow (\mathrm{R}_2, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_{z-1})$	when 1 robot executes
	$\rightarrow (\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_{z-2})$	when 2 robots execute
RC5$_m$ ::	$(\mathrm{R}_2, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_z) \rightarrow (\mathrm{R}_2, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_{z-1})$	

4.1 Correctness

In the following a round denotes the shortest fragment of execution where each robot executes at least once.

Lemma 9. *Starting in a legitimate configuration, after the execution of a round, the position of all the robots shift one location in the same direction.*

In the following we compute the *service time* of the algorithm (the number of steps necessary to all three robots to completely explore the ring at least once).

Lemma 10. *The service time of the algorithm is* $\Theta(kn)$.

Let C'_1 denote the configurations $(\mathrm{R}_1, \mathrm{F}_y, \mathrm{R}_2, \mathrm{F}_z)$ with $y \neq z \neq \{0,1,2,3\}$, C'_2 denote $(\mathrm{R}_1, \mathrm{F}_x, \mathrm{R}_1, \mathrm{F}_y, \mathrm{R}_1, \mathrm{F}_y)$ and C'_3 denote the configurations $(\mathrm{R}_1, \mathrm{F}_x, \mathrm{R}_1, \mathrm{F}_y, \mathrm{R}_1, \mathrm{F}_z)$.

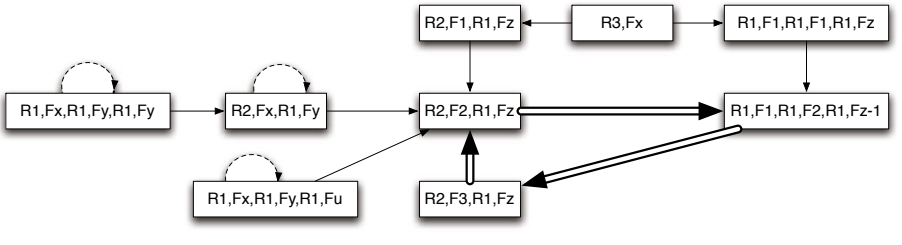
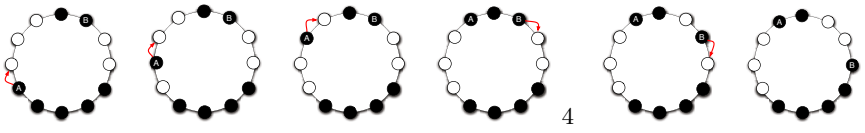


Fig. 2. Algorithm with three robots, convergence et legitimate phases



(a) C_0 (b) C_1 (c) C_2 (d) C_3 (e) C_4 (f) C_0

Fig. 3. Perpetual exploration with a maximum number of robots

Lemma 11. *Starting from a configuration C'_i the system converges to a legitimate configuration (C_i, i=1,3).*

Following the above lemmas we can state the correctness of the algorithm.

Theorem 1. *The algorithm implements the constrained perpetual exploration.*

5 Exploration Algorithm Using a Maximum Number of Robots

In this section we propose an algorithm that achieves a constrained perpetual exploration of a ring of size n with $k = n-5$ robots for $k > 3$ odd and n mod $k \neq 0$. As for the protocol with 3 robots, the algorithm works into two phases: the *Legitimate Phase* is to perpetually explore the ring and the *Convergence Phase* is to reach a legitimate configuration.

The main idea of the algorithm, once in a legitimate configuration is to maintain the invariant that if there is a block of 3 robots, than one robot in this block moves to join the other block (whose size is bigger) via the shortest path. Then, once the robot joins the biggest block, a robot in the opposite side of the block moves to join the block of size 2. A functioning scenario of the Legitimate Phase of the algorithm is presented in Figure 3.

Legitimate Phase: $\mathtt{LF}_M$	
$\mathtt{RL1}_M$::	$(\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_2) \rightarrow (\mathrm{F}_2, \mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_2)$
$\mathtt{RL2}_M$::	$(\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_2) \rightarrow (\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_2)$
$\mathtt{RL3}_M$::	$(\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_2) \rightarrow (\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_3)$
$\mathtt{RL4}_M$::	$(\mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_3) \rightarrow (\mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_2, \mathrm{F}_1, \mathrm{R}_1)$
$\mathtt{RL5}_M$::	$(\mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_2, \mathrm{F}_1, \mathrm{R}_1) \rightarrow (\mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_3, \mathrm{R}_3)$

We now describe how to reach a legitimate configuration starting from any configuration. We first introduce some notation that we will use later to describe one of the rules according to which robots move. Given a configuration $c = (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1, \mathrm{R}_k, \mathrm{F}_1)$, let S_c be the set of blocks with minimum size, i.e., $S_c = \{\mathrm{R}_i \; with \; i = \min\{x, y, z, w, k\}\}$. For a given R_i $i = \{x, y, z, w, k\}$, let $LN(\mathrm{R}_i)$ be the largest block at distance one free node from R_i in c. Finally, let $T_c = \{\mathrm{R}_i \in S :: \forall \mathrm{R}_j \in S, |LN(\mathrm{R}_i)| \geq |LN(\mathrm{R}_j)|\}$.

Convergence Phase (where ♠ denotes the sequence $\mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1$).

RC2-1$_M$::	$(\mathrm{R}_x, \mathrm{F}_5)$	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_4)$	if one robot moves
RC2-2$_M$::		$\to (\mathrm{R}_{x-2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_1)$	if both robots move
RC4-1$_M$::	$(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_2)$	$3 < x < y$
RC4-2$_M$::	$(\mathrm{R}_x, \mathrm{F}_4, \mathrm{R}_y, \mathrm{F}_1)$	$\to (\mathrm{R}_{x-1}, \mathrm{F}_4, \mathrm{R}_{y+1}, \mathrm{F}_1)$	$3 < x < y$
RC4-3$_M$::	$(\mathrm{R}_1, \mathrm{F}_4, \mathrm{R}_y, \mathrm{F}_1)$	$\to (\mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$	
RC4-4$_M$::	$(\mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$	$\to (\mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_{y-1}, \spadesuit)$	
RC4-5$_M$::	$(\mathrm{R}_x, \mathrm{F}_4, \mathrm{R}_y, \mathrm{F}_1)$	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1)$	$x \in \{2, 3\}$
RC6-1$_M$::	$(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$	$\to (\mathrm{R}_{x+1}, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{z-1}, \mathrm{F}_1)$	$z > 1$ and $x \geq y$
RC6-2$_M$::	$(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$	$\to (\mathrm{R}_{x+1}, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{z-1}, \mathrm{F}_1)$	$x = 1$ and $x \neq y$
RC6-3$_M$::	$(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \spadesuit)$	$\to (\mathrm{R}_{x+1}, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$	$2 < y \leq x$
RC6-4$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \spadesuit, \mathrm{R}_{z-1}, \mathrm{F}_2)$	$z > 1$ if one robot moves
RC6-5$_M$::		$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \spadesuit, \mathrm{R}_{z-2}, \spadesuit)$	$z > 1$ if both robots move
RC6-6$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$1 < x < y$, $x \neq y - 1$
RC6-7$_M$::	$(\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$\to (\mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_1)$	$2 < y < z$
RC6-8$_M$::	$(\mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$\to (\mathrm{R}_{y+1}, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_3)$	$2 < z < y$
RC6-9$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \spadesuit, \mathrm{R}_{z-1}, \mathrm{F}_2)$	$1 < x = y - 1$
RC6-10$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_2)$	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_1)$	$z = 1, 1 < x \leq y$

RC8-1$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_{w-1}, \mathrm{F}_1)$		$x = y$ and $w < z$
RC8-2$_M$:	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_w)$		
	$\to (\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_w)$		$x = y$ and $z = x + 1$ and $w = 1$
RC8-3$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_{w-1}, \mathrm{F}_1)$		$x > y$
RC8-4$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$		$x \geq y$ and $y = z$ and $w = 1$
RC8-5$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{w+1}, \mathrm{F}_1)$		$x \geq y$ and $y = z$ and $w = y + 1$
RC8-6$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$		$x > y$, $y \neq z$, $z = 1$ $w = x$
RC8-7$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_{w+1}, \mathrm{F}_1)$		$x > y$, $y \neq z$ $z = 1$ $w = x - 1$
RC8-8$_M$::	$(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$		$x > y$, $y = z - 1$ and $w = 1$
RC10-1$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		$z = w$ and $k = y$
	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		if one robot moves
RC10-2$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		$z = w$ and $k = y$
	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$		if two robots move
RC10-3$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$		$w = z + 1$ and $k = y - 1$
RC10-4$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$		
	$\to (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_{z+1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$		$y < z$ and $x \notin \{y, z\}$
RC10-5$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$		$x > z$
RC10-6$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		$x = y - 1$, if one robot moves
RC10-7$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x-2}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_1)$		$x = y - 1$, if two robots move
RC10-8$_M$::	$(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		
	$\to (\mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$		$x = y + 1$, if one robot moves

$\texttt{RC10-9}_M$:: $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_{x+2}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$ $x = y + 1$, if two robots move

$\texttt{RC10-10}_M$:: $(\mathrm{R}_{y+2}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_{y+3}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$ $y > 1$

$\texttt{RC10-11}_M$:: $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_1, \mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$ $y = x + 1$, if one robot moves

$\texttt{RC10-12}_M$:: $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_1, \mathrm{R}_{x-2}, \mathrm{F}_1, \mathrm{R}_{y+1}, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1)$ $y = x + 1$, if two robots move

$\texttt{RC10-13}_M$:: $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$ $x = y + 1$, if one robot moves

$\texttt{RC10-14}_M$:: $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$
$\rightarrow (\mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_{y-1}, \mathrm{F}_1)$ $x = y + 1$, if two robots move

$\texttt{RC10-15}_M$:: $c = (\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1, \mathrm{R}_w, \mathrm{F}_1, \mathrm{R}_k, \mathrm{F}_1)$
$\rightarrow$ a robot in $\mathrm{R}_i \in T_c$ moves towards $LN(\mathrm{R}_i)$

5.1 Correctness

First, we prove that a collision can never occur and that once in a legitimate configuration any robot that moves according to an old snapshot, it moves according to the algorithm Legitimate phase.

Lemma 12. *Robots never collide and in each legitimate configuration a robot that moves according to an old snapshot, moves as expected by the algorithm Legitimate phase.*

Proof. If one robot at time moves, then since a robot moves always towards a free node, it cannot create a collision. The only configurations when more than one robot move, are the symmetric configurations. Observe that in this configurations if two robots move at the same time, they never move towards the same free node. Thus, the only way to create a collision is the case when one of the two robots that was expected to move in a given symmetric configuration, moves later with respect to the other one. Thus, it moves according to an old snapshot in a configuration where some other robot may be scheduled to move. In the following we prove that, even though a robot moves according to an old snapshot, it never creates a collision. In the following we analyze all possible symmetric configurations:

1. First we consider all symmetric configurations where robots are divided into 5 groups. Observe that we cannot have all the blocks of the same size because this implies the number of robots is a multiple of the number of free nodes.
   - Consider all configurations where we have exactly two different sizes for the blocks: $c_1 = (R_x, F_1, R_y, F_1, R_y, F_1, R_y, F_1, R_y, F_1)$; $c_2 = (R_y, F_1, R_y, F_1, R_x, F_1, R_x, F_1, R_x, F_1)$; $c_3 = (R_x, F_1, R_y, F_1, R_x, F_1, R_y, F_1, R_x, F_1)$.
     When at configuration $c_1 = (R_x, F_1, R_y, F_1, R_y, F_1, R_y, F_1, R_y, F_1)$, if $x \notin \{y, 1, y + 1\}$, we recursively apply $\texttt{RC10-2}_M$(if both robots move) or $\texttt{RC10-1}_M$and then $\texttt{RC10-3}_M$(if one robot moves). Either we reach a

configuration $(R_x, F_2, R_z, F_1, R_z, F_2)$ where no robot is pending to move, or we reach a configuration $(R_x, F_2, R_z, F_1, R_{z-1}, F_1, R_1, F_1)$, where the single robot r is pending to move to join R_{z-1}. If $x > z$ we apply $\texttt{RC8-3}_M$, if $x < z$ we apply $\texttt{RC8-7}_M$. In both cases, r is the robot scheduled to move. Since $z = 2y$ we have that $z \neq x$, otherwise we have an even number of robots.

It remains to consider the case where $x \in \{y-1, y+1\}$. If $x = y-1$, we apply $\texttt{RC10-6}_M$if one robot moves and we reach a configuration $(R_{y-2}, F_1, R_{y+1}, F_1, R_y, F_1, R_y, F_1, R_y, F_1)$, where there is a robot r in R_{y-2} pending to move to join R_y. We recursively apply $\texttt{RC10-15}_M$and r is eventually scheduled to move alone.

If $x = y+1$ and only one robot moves ($\texttt{RC10-8}_M$), we reach a configuration where we apply $\texttt{RC10-10}_M$, where the robot pending to move is the one scheduled to move.

Consider configuration $c_3 = (R_x, F_1, R_y, F_1, R_x, F_1, R_y, F_1, R_x, F_1)$. If $x \neq y+1$ we apply $\texttt{RC10-2}_M$, or $\texttt{RC10-1}_M$and then $\texttt{RC10-3}_M$, to reach a configuration where no robot is pending to move. Otherwise, we apply $\texttt{RC10-11}_M$or $\texttt{RC10-12}_M$. If only one robot moves, we reach a configuration $(R_x, F_1, R_{y+1}, F_1, R_{x-1}, F_1, R_y, F_1, R_x, F_1)$ where a robot r in R_{x-1} is pending to move towards R_y. Then, we recursively apply $\texttt{RC10-15}_M$and we eventually reach a configuration where r is the only robot scheduled to move. If it moves before this configuration it moves in the opposite direction of the robot scheduled to move, so no collision happens.

Consider configuration $c_2 = (R_x, F_1, R_x, F_1, R_y, F_1, R_y, F_1, R_y, F_1)$. If $x \neq y+1$, we apply $\texttt{RC10-2}_M$(two robots move) or $\texttt{RC10-1}_M$(one robot moves) and then $\texttt{RC10-3}_M$, to reach a configuration where no robot is pending to move. If $x = y+1$ and only one robot moves, according to $\texttt{RC10-13}_M$, we reach a configuration $(R_x, F_1, R_{x+1}, F_1, R_{y-1}, F_1, R_y, F_1, R_y, F_1)$ where a robot r is pending to move from R_y to join R_x. We recursively apply $\texttt{RC10-15}_M$. And if r does not move, we reach a configuration $(R_x, F_1, R_z, F_1, R_1, F_1, R_y, F_1, R_y, F_1)$ where $z > x$ where there is another robot r' pending to move from R_1 to R_z. If r' moves before r, we apply $\texttt{RC8-5}_M$and r is the robot scheduled to move. If r moves before, r' is the robot scheduled to move according to $\texttt{RC10-15}_M$.

– Consider all configurations where we have exactly three different sizes for the blocks. Either we have three blocks of the same size: $c_1 = (R_x, F_1, R_x, F_1, R_x, F_1, R_y, F_1, R_z, F_1)$; $c_2 = (R_x, F_1, R_x, F_1, R_y, F_1, R_x, F_1, R_z, F_1)$; or we have two pairs of blocks of the same size: $c_3 = (R_x, F_1, R_x, F_1, R_y, F_1, R_y, F_1, R_z, F_1)$; $c_4 = (R_x, F_1, R_y, F_1, R_x, F_1, R_y, F_1, R_z, F_1)$; $c_5 = (R_x, F_1, R_y, F_1, R_y, F_1, R_x, F_1, R_z, F_1)$.

  When at configuration c_4 we apply $\texttt{RC10-4}_M$where only one robot moves and we reach a not symmetric configuration. When at c_5 we apply $\texttt{RC10-2}_M$if both robots move, or $\texttt{RC10-1}_M$and then $\texttt{RC10-3}_M$to reach a configuration of type c_5 where the size of x decreases by one and the

size of y increases by one w.r.t. the previous configuration. Thus, by recursively applying the above rules we reach either a quiescent configuration $(R_z, F_2, R_y, F_1, R_y, F_2)$ or a configuration $(R_z, F_2, R_y, F_1, R_{y-1}, F_1, R_1, F_1)$ where the single robot r in R_1 will eventually move towards R_{y-1}. In this latter configuration, if $z \geq y$ we apply $\texttt{RC8-1}_M$; if $y > z$ we apply $\texttt{RC8-8}_M$. In all the cases r is the robot scheduled to move. Hence, we reach a quiescent configuration and no collision happens.
When in configuration c_2 we apply $\texttt{RC10-5}_M$ and we reach a configuration
$(R_x, F_1, R_x, F_1, R_y, F_1, R_{x-1}, F_1, R_{z+1}, F_1)$ where $z > y$. Then, we apply $\texttt{RC10-15}_M$. Observe either the robots in R_{x-1} continue to join R_{z+1}, or the robots in R_y move to join the robot in R_x. Hence, no collision can happen. Finally, when at configuration c_1 or c_3, we apply $\texttt{RC10-15}_M$. Since in both cases there is only one robot that is scheduled to move, we cannot create a collision.

– Finally, consider all configurations where we have exactly four different sizes for the blocks. Then, we have at most two blocks of robots of the same size. We apply $\texttt{RC10-15}_M$.
Two robots at time may move only if we are at $(R_x, F_1, R_y, F_1, R_z, F_1, R_w,\ F_1, R_z, F_1)$ where either z is the minimum size and $w > x, y$ or w is the minimum size. In the first case, the robots in both R_z move to join R_w: if both robots move no collision happen; if one robot moves than we reach a configuration $(R_x, F_1, R_y, F_1, R_{z-1}, F_1, R_{w+1}, F_1, R_z, F_1)$ where there is a robot in R_z pending to move to join R_{w+1}. We apply $\texttt{RC10-15}_M$ recursively and we obtain a configuration $(R_x, F_1, R_y, F_2, R_{w'}, F_1, R_z, F_1)$ where $w' > y$ and $x \neq y$. Hence, for the $\texttt{RC8-3}_M r$ is the robot scheduled to move. In the second case, where w is the minimum, we have a single block of robots that has to move in two opposite direction. But in case there is a robot that at some time is pending to move, it will be the last in this block to move. Hence, no collision can happen.

2. Consider, the configuration $(R_x, F_1, R_x, F_2, R_z, F_2)$. Because of $\texttt{RC6-4}_M$if only one robot moves we reach configuration $(R_x, F_1, R_x, F_1, R_1, F_1, R_{z-1}, F_2)$, and there is a robot r that eventually will move from R_{z-1} to join R_x. If $z - 1 > x$, we apply $\texttt{RC8-4}_M$, and if $x > z - 1$, we apply $\texttt{RC8-6}_M$. In both cases, and r is the only robot scheduled to move. Finally consider $z - 1 = x$, we apply $\texttt{RC8-3}_M$and r is the robot scheduled to move.
3. for the symmetric configuration $(R_x, F_1, R_y, F_3, R_y, F_1)$. If two robots move they both move towards a block R_y, so no collision can happen. If only one robot moves, (according to $\texttt{RC6-1}_M$) we reach a configuration $(R_{x-1}, F_1, R_{y+1}, F_3, R_y, F_1)$ and a robot r is pending to move from R_{x-1} to join R_y. We recursively apply $\texttt{RC6-1}_M$and eventually we reach a configuration $(R_1, F_1, R_{y+x-1}, F_3, R_y, F_1)$ if r is still pending to move. Then, r is the only robot scheduled to move, because either in a legitimate configuration or by $\texttt{RC6-3}_M$.

4. Finally, consider the configuration where all robots are collected to form a single block. The two robots at the extreme of the block are expected to move. If they both move, it is simple to see that no collision may happen. If only one robot moves we reach a configuration (R_x, F_4, R_1, F_1). Then, we apply $\texttt{RC4-3}_M$and the pending robot is the one scheduled to move.

Then, we prove that starting from any configuration we always reach a legitimate configuration,i.e.; one of the configurations we reach in the execution of algorithm $\texttt{LF}_M$.

Lemma 13. *Starting from a non legitimate configuration the system converges to a legitimate configuration in finite time.*

Proof. 1. When in a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$, if $x \in \{2, 3\}$ we are in a legitimate configuration. Thus, first consider $x = 1$. Applying $\texttt{RC4-4}_M$and then $\texttt{RC6-2}_M$(being at configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_{y-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$), we move a robot from R_y to R_x and we reach a legitimate configuration.
Then consider $3 < x < y$. In this latter case, we recursively apply in sequence the three rules $\texttt{RC4-1}_M$, $\texttt{RC6-7}_M$and $\texttt{RC6-8}_M$to move one robot at the time from R_x to R_y up to the time we are in a legitimate configuration because $x = 3$. In particular, in each step of the recursion, we apply $\texttt{RC4-1}_M$to reach a configuration $(\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_y, \mathrm{F}_2)$. Then, we apply $\texttt{RC6-7}_M$to move to $(\mathrm{R}_{x-1}, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2)$, and then $\texttt{RC6-8}_M$to reach configuration $(\mathrm{R}_{x-1}, \mathrm{F}_3, \mathrm{R}_{y+1}, \mathrm{F}_2)$.

2. Consider a configuration $(\mathrm{R}_x, \mathrm{F}_4, \mathrm{R}_y, \mathrm{F}_1)$. If $x = 1$ we apply $\texttt{RC4-3}_M$and we reach a configuration $(\mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$. We apply $\texttt{RC6-1}_M$and $\texttt{RC6-2}_M$(if only one robot moved) and we reach a configuration $(\mathrm{R}_2, \mathrm{F}_3, \mathrm{R}_2, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_1)$. We recursively apply $\texttt{RC6-1}_M$and when $z = 1$ we apply $\texttt{RC6-3}_M$to reach a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$ (above proved). If $x \in \{2, 3\}$, we apply $\texttt{RC4-5}_M$ and we reach a configuration $(\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1)$. Either we are in the case previously proved or we apply $\texttt{RC6-2}_M$to reach configuration $(\mathrm{R}_2, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$ and we apply the algorithm $\texttt{RL1}_m$.
Finally, if $x > 3$ we recursively apply $\texttt{RC4-2}_M$and we reach a configuration $(\mathrm{R}_x, \mathrm{F}_4, \mathrm{R}_y, \mathrm{F}_1)$ where $x = 3$. The claim follows.

3. When in a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$ we apply $\texttt{RC6-1}_M$ to $\texttt{RC6-3}_M$ and eventually the set R_z joins the biggest set between R_x and R_y, say y.
If $z > 1$ and $x > 1$, we repeatedly apply $\texttt{RC6-1}_M$ eventually lead to a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$, proved above. If $x = 1$, we apply $\texttt{RC6-2}_M$ and we reach a configuration that is either a legitimate configuration, or such that we have to recursively apply $\texttt{RC6-1}_M$ and then we reach a legitimate configuration. If $z = 1$ and $x = 2$ we are in a legitimate configuration and we execute $\texttt{RL5}_M$. Otherwise, $x > 2$ and we apply $\texttt{RC6-3}_M$ to reach a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_2)$, proved above.

4. When in a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$, for $z = 1$, we apply $\texttt{RC6-10}_M$ and reach a configuration $(\mathrm{R}_x, \mathrm{F}_3, \mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_1)$. Then, for what above proved we reach a legitimate configuration in finite time. Then, consider the same configuration for $z > 1$. We have the following cases:

- if $x = 1$, we apply RC6-7$_M$and RC6-8$_M$and in at most two steps we reach a configuration $(\mathrm{R}_{y+1}, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_3)$
- If $1 < x < y$ and $x \neq y - 1$, we repeatedly apply RC6-6$_M$ to reach configuration $(\mathrm{R}_{x+y}, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_3)$ and for what above stated we reach a legitimate configuration.
- If $1 < x = y - 1$, we apply RC6-9$_M$ and we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{z-1}, \mathrm{F}_2)$. Then, if $z-1 > x$ we apply RC8-8$_M$and we reach a configuration $(\mathrm{R}_{z-1}, \mathrm{F}_2, \mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{x+2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$. then, because of RC8-3$_M$we reach a configuration $(\mathrm{R}_z, \mathrm{F}_2, \mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{x+2}, \mathrm{F}_2)$. The claim follows for what proved in the previous point.
  If $x > z - 1$, we applyRC8-3$_M$ and we eventually reach a configuration $(\mathrm{R}_{x+y}, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_z, \mathrm{F}_2)$. Then, we apply consecutively apply RC6-7$_M$and RC6-8$_M$and we reach a configuration $(\mathrm{R}_{x+y+1}, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_3)$.
  Finally, consider $x = z - 1$. We start from a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_2)$. We apply RC6-9$_M$and we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2)$. Then, we apply RC8-2$_M$and we reach a configuration $(\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{x+2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2)$. Finally, by applying RC8-2$_M$we reach a configuration $(\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_{x+2}, \mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_2)$ (above proved).
- If $x = y$, we apply RC6-4$_M$orRC6-5$_M$ (respectively if one robot moves or two robots move). First, consider that two robots move.
  Then we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{z-2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$. We are in a new symmetric configuration where two robots may move. If both robots move, according to the rule RC10-2$_M$we reach a configuration $(\mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{z-2}, \mathrm{F}_2)$. Then it is simple to see that by repeating the above rules we reach a configuration $(\mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_2)$ and then, by applying RC6-10$_M$, we reach a configuration $(\mathrm{R}_y, \mathrm{F}_1, \mathrm{R}_y, \mathrm{F}_3, \mathrm{R}_1, \mathrm{F}_1)$.
  If at $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{z-2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$ only one robot moves (i.e., we apply RC10-1$_M$), we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{z-2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$:
  If $z - 2 > x + 1$, we apply RC8-2$_M$and we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{z-1}, \mathrm{F}_2)$ (above proved).
  If $z-2 < x+1$, we apply RC8-6$_M$and we reach a configuration $(\mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{z-2}, \mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_1)$. Then we pass from an initial configuration $(\mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2, \mathrm{R}_x, \mathrm{F}_1)$ to $(\mathrm{R}_{x+1}, \mathrm{F}_2, \mathrm{R}_{z-2}, \mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_1)$. This means that eventually we reach a configuration $(\mathrm{R}_{x'}, \mathrm{F}_2, \mathrm{R}_1, \mathrm{F}_2, \mathrm{R}_{x'}, \mathrm{F}_1)$, where we apply RC6-10$_M$to reach $(\mathrm{R}_{x'}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_3, \mathrm{R}_{x'}, \mathrm{F}_1)$ (above proved).
  If $z-2 = x+1$, we apply RC8-1$_M$and we reach a configuration $(\mathrm{R}_{x+2}, \mathrm{F}_2, \mathrm{R}_{x+1}, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2)$ (above proved).
  It remains to prove that we converge to a legitimate configuration even if starting at configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2, \mathrm{R}_z, \mathrm{F}_2)$ only one robot moves, and we reach a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_{z-1}, \mathrm{F}_2)$.
  If $z - 1 > x$ we apply RC8-3$_M$, if $x > z - 1$ we apply RC8-6$_M$. In both cases, we reach a configuration $(\mathrm{R}_{z-2}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1)$. We reduce to the case where two robots move, previously proved.
  Finally, if $x = z-1$ we are in a configuration $(\mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_2)$. We apply RC8-3$_M$to reach a configuration $(\mathrm{R}_{x-1}, \mathrm{F}_1, \mathrm{R}_1, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1, \mathrm{R}_x, \mathrm{F}_1,$

R_1, F_1). We apply RC10-1$_M$, RC10-2$_M$if respectively one robot moves or two robots move, and we repeat the above reasoning.

5. Consider any configuration with 4 blocks, i.e., $(R_x, F_2, R_y, F_1, R_z, F_1, R_w, F_1)$. It is simple to see that rule RC8-3$_M$ converges to a configuration with three blocks. Then, we prove that every other rule either converges to a configuration where we have to apply RC8-3$_M$or to a configuration with three blocks.
   Consider RC8-4$_M$, then from configuration $(R_x, F_2, R_y, F_1, R_y, F_1, R_1, F_1)$ we pass to configuration $(R_{x-1}, F_1, R_1, F_1, R_y, F_1, R_y, F_1, R_1, F_1)$. If $x = 2$ and $y = x + 1$, we apply RC10-13$_M$if only one robot moves or RC10-14$_M$if two robots move. In this latter case we reach a configuration $(R_{y+1}, F_1, R_{y+1}, F_2, R_1, F_2)$ and the claim follows. In the first case, we reach a configuration $(R_y, F_1, R_{y+1}, F_2, R_1, F_1, R_1, F_1)$, where we have to apply RC8-3$_M$. Otherwise we apply RC10-1$_M$if only one robot moves or RC10-2$_M$if two robots move. In the first case, we reach a configuration $(R_{x-1}, F_2, R_{y+1}, F_1, R_y, F_1, R_1, F_1)$ and we can apply only RC8-3$_M$. In the second case we reach a configuration $(R_{x-1}, F_2, R_{y+1}, F_1, R_{y+1}, F_2)$.

   Consider RC8-6$_M$, then from configuration $(R_x, F_2, R_y, F_1, R_1, F_1, R_x, F_1)$ we pass to configuration $(R_x, F_1, R_1, F_1, R_{y-1}, F_1, R_1, F_1, R_x, F_1)$. We apply RC10-2$_M$ if two robots move to reach a configuration $(R_{y-1}, F_2, R_{x+1}, F_1, R_{x+1}, F_2)$. Otherwise, we apply RC10-1$_M$and we reach configuration $(R_{x+1}, F_2, R_{y-1}, F_1, R_1, F_1, R_x, F_1)$. Then, we apply RC8-7$_M$to reach a configuration with three blocks.
   Consider RC8-8$_M$, after the application of the rule at configuration $(R_x, F_2, R_y, F_1, R_{y+1}, F_1, R_1, F_1)$ to reach a configuration $(R_x, F_2, R_{y-1}, F_1, R_{y+2}, F_1, R_1, F_1)$. Either we have to apply RC8-3$_M$or $x = y - 1$. In this latter case, we apply RC8-1$_M$and we reach a configuration with three blocks.
   It is simple to see that the claim follows for RC8-5$_M$and RC8-7$_M$. Finally, with RC8-1$_M$and RC8-2$_M$we move to a configuration where $x \neq y$.
   Hence, the claim follows.
6. In a configuration with 5 blocks with rules RC10-4$_M$- RC10-14$_M$we arrive in a configuration where we have either to apply RC10-15$_M$or RC10-1$_M$, RC10-2$_M$. With RC10-15$_M$we converge to a configuration with less than 5 blocks, because we consider the minimum blocks and among them we select the one(s), say R_x, with the biggest block at distance one free node. This invariant is maintained while R_x move one robot at time to join the block at distance one free node with the biggest size. Finally, it is simple to see that if we recursively apply the sequence RC10-1$_M$, RC10-3$_M$or RC10-2$_M$we converge to a configuration with less than 5 blocks.
7. When in a configuration with one block of robots, because of RC2-1$_M$ or RC2-2$_M$, we reach a configuration either with 2 or 3 blocks. The claim follows.

It remains to prove that once the robots reach a legitimate configuration, each robot will explore all the ring without colliding with any other robot.

Theorem 2. *The Algorithm implements the constrained perpetual exploration.*

Proof. Following Lemma 13, the system in a finite number of steps converges to a legitimate configuration. By Lemma 12 no collision can happen and once in a legitimate configuration we move to another legitimate configuration.

Then, consider that at time t the system is in a quiescent legitimate configuration c, i.e., there is no robot that at that time should move because of a Look phase executed before t. It is simple to see that once we reach a quiescent legitimate configuration the system will move to another quiescent legitimate configuration. This is because in any legitimate configuration, each robot has a different view of the system and thus, at a given time, only one robot moves.

If $x = 2$, even though one robot moves from $\mathtt{R}_y$ to $\mathtt{R}_x$, we maintain the invariant that $y' = y - 1 > x$. When $x = 3$ we move one robot from $\mathtt{R}_x$ to $\mathtt{R}_y$. Then we return to the initial configuration. Note that once a robot moves from $\mathtt{R}_y$ to $\mathtt{R}_x$ via the longest path, it will eventually come back to $\mathtt{R}_y$ through the shortest path and shift in $\mathtt{R}_y$ up to reach the initial position and repeat all the above steps. Then, each robot visits all the nodes in the ring. Hence, the claim follows.

6 Conclusions

In this paper we extend the results in [1] related to the perpetual ring exploration. We consider the most generic model: our robots are asynchronous and are not given any sense of direction, so the left and right sense (*i.e.* chirality) is decided by the adversary that schedules robots for execution, and may change between invocations of a particular robots (as robots are oblivious). This very weak assumption preserves all usual problems related to symmetry breaking. We investigate both the minimal and the maximal number of robots that are necessary and sufficient to solve the exclusive perpetual exploration problem. On the minimal side, we prove that three deterministic robots are necessary and sufficient, provided that the size n of the ring is at least 10, and show that no protocol with three robots can exclusively perpetually explore a ring of size less than 10. On the maximal side, we prove that $k = n - 5$ robots are necessary and sufficient to exclusively perpetually explore a ring of size n when n is co-prime with k.

References

1. Baldoni, R., Bonnet, F., Milani, A., Raynal, M.: On the solvability of anonymous partial grids exploration by mobile robots. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 428–445. Springer, Heidelberg (2008)
2. Devismes, S., Petit, F., Tixeuil, S.: Optimal probabilistic ring exploration by asynchronous oblivious robots. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 195–208. Springer, Heidelberg (2010)
3. 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)

4. 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)
5. Klasing, R., Kosowski, A., 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)
6. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theor. Comput. Sci. 390(1), 27–39 (2008)
7. Prencipe, G.: Instantaneous actions vs. full asynchronicity: Controlling and coordinating a set of autonomous mobile robots. In: Restivo, A., Rocca, S.R.D., Roversi, L. (eds.) ICTCS. LNCS, vol. 2202, pp. 154–171. Springer, Heidelberg (2001)
8. Blin, L., Milani, A., Potop-Butucaru, M., Tixeuil, S.: Exclusive Perpetual ring exploration without chirality. Technical report inria-00464206 (2010)

Drawing Maps with Advice

Dariusz Dereniowski[1,⋆] and Andrzej Pelc[2,⋆⋆]

[1] Department of Algorithms and System Modeling,
Gdansk University of Technology, ul. Narutowicza 11/12, 80-233 Gdańsk, Poland
deren@eti.pg.gda.pl
[2] Département d'informatique, Université du Québec en Outaouais,
Gatineau, Québec J8X 3X7, Canada
pelc@uqo.ca

Abstract. We study the problem of the amount of information required to draw a complete or a partial map of a graph with unlabeled nodes and arbitrarily labeled ports. A mobile agent, starting at any node of an unknown connected graph and walking in it, has to accomplish one of the following tasks: draw a complete map of the graph, i.e., find an isomorphic copy of it including port numbering, or draw a partial map, i.e., a spanning tree, again with port numbering. The agent executes a deterministic algorithm and cannot mark visited nodes in any way. None of these map drawing tasks is feasible without any additional information, unless the graph is a tree. This is due to the impossibility of recognizing already visited nodes. Hence we investigate the minimum number of bits of information (minimum size of *advice*) that has to be given to the agent to complete these tasks. It turns out that this minimum size of advice depends on the numbers n of nodes or the number m of edges of the graph, and on a crucial parameter μ, called the *multiplicity* of the graph, which measures the number of nodes that have an identical *view* of the graph.

We give bounds on the minimum size of advice for both above tasks. For $\mu = 1$ our bounds are asymptotically tight for both tasks and show that the minimum size of advice is very small: for an arbitrary function $\varphi = \omega(1)$ it suffices to give $\varphi(n)$ bits of advice to accomplish both tasks for n-node graphs, and $\Theta(1)$ bits are not enough. For $\mu > 1$ the minimum size of advice increases abruptly. In this case our bounds are asymptotically tight for topology recognition and asymptotically almost tight for spanning tree construction. We show that $\Theta(m \log \mu)$ bits of advice are enough and necessary to recognize topology in the class of graphs with m edges and multiplicity $\mu > 1$. For the second task we show that $\Omega(\mu \log(n/\mu))$ bits of advice are necessary and $O(\mu \log n)$ bits of advice are enough to construct a spanning tree in the class of graphs with n nodes and multiplicity $\mu > 1$. Thus in this case the gap between our bounds is always at most logarithmic, and the bounds are asymptotically tight for multiplicity $\mu = O(n^{\alpha})$, where α is any constant smaller than 1.

⋆ This work was done during the visit of Dariusz Dereniowski at the Research Chair in Distributed Computing of the Université du Québec en Outaouais. This author was partially supported by the MNiSW grant N N206 379337.

⋆⋆ Partially supported by NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 328–342, 2010.

1 Introduction

The background and the problem. Knowing the topology of a network or at least a spanning tree of it is of significant help in organizing communication among nodes of the network and in accomplishing distributed tasks. It is well known that such tasks as, e.g., broadcasting or gossiping (information exchange) can be performed more efficiently when the topology of the network or a spanning tree of it are available to nodes, than when they have to be executed in an unknown network. One way of supplying this vital information to nodes is by means of exploration of the network, where a mobile agent has to traverse all links of the network and visit all its nodes. In fact, one of the main reasons to perform the well-studied task of graph exploration, see, e.g., [2, 4, 5, 8, 9], is to draw a faithful map of the graph that models the explored network. Provided that the agent has enough memory and computation power and that the exploration has been performed, drawing a map of the graph is easy, if nodes have distinct identities that can be perceived by the agent, or if the agent can leave marks at nodes when it visits them. However, neither of these assumptions is always satisfied. On the one hand, nodes may refuse to reveal their identities, e.g., for security reasons, or limited sensory capabilities of the agent may prevent it from perceiving these identities; on the other hand, nodes may not have facilities (whiteboards) allowing to leave marks, or such marks may be destructed between visits of the agent and thus unreliable. Thus it is important for the agent to be able to recognize the topology of the graph (draw a full map of the network) or construct its spanning tree (which is an important partial map of it) without relying on identities of the nodes and without marking them. In contrast, in order to allow the agent to move in the network, we have to assume that ports at every node are distinguishable for the agent. If an agent were unable to locally distinguish ports at a node, it may have even been unable to visit all neighbors of a node of degree at least 3. Indeed, after visiting the second neighbor, the agent cannot distinguish the port leading to the first visited neighbor from the port leading to the unvisited one. Thus an adversary may always force an agent to avoid all but two edges incident to such a node, thus effectively precluding exploration. Hence we assume that a node of degree d has ports $1, \ldots, d$ corresponding to the incident edges. Ports at each node can be perceived by an agent visiting this node, but there is no coherence assumed between port labelings at different nodes.

We consider two map drawing tasks that have to be accomplished by a mobile agent executing a deterministic algorithm and walking in an unknown connected graph with unlabeled nodes and labeled ports. One is *topology recognition* which consists in returning an isomorphic copy of the graph with correctly numbered ports, and the other is *spanning tree construction* which consists in returning a spanning tree of the graph, again with correctly numbered ports. Both tasks have to be performed by an agent that starts in an arbitrary node of an unknown graph and is allowed to explore it. These tasks can be easily accomplished after exploration, if the graph is a tree (and in this case they are obviously equivalent): after performing an Eulerian tour of the tree, the agent realizes this fact and

can reconstruct the topology of the tree. However, it turns out that *unless* the graph is a tree, none of these tasks can be accomplished without any additional information given to the agent.

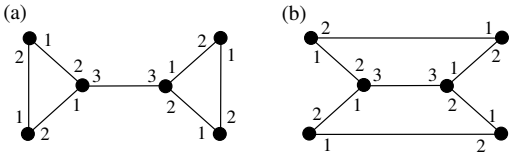


Fig. 1. Non-isomorphic graphs undistinguishable by the agent

Fig. 1 (cf. [25]) gives an example of two non-isomorphic graphs after whose exploration an agent will get identical information, and thus will not be able to distinguish them. Indeed, a stronger fact is true: for any graph that is not a tree, an agent that has explored the graph and has no additional information can neither recognize the topology of the graph, nor even construct its spanning tree. This yields the main problem that we study in this paper.

> What is the minimum number of bits of a priori information required by an agent exploring a graph, in order to recognize its topology or to construct its spanning tree?

This way of stating the problem follows the paradigm of network algorithms with *advice* that has become recently popular (cf. the subsection "Related work"), and can be described as follows. An oracle knowing the entire network can give a string of bits (advice) to the mobile agent. (In other settings strings of bits are given to nodes that subsequently exchange messages.) Then an algorithm is executed by the agent without knowing in which network it operates, using the provided advice. The total number of bits given by the oracle is the size of advice. Thus the framework of advice permits to quantify the amount of information needed to solve a network problem, regardless of the type of information that is provided.

Since for trees no additional information is needed, in the rest of the paper we assume that the explored graph is not a tree. It turns out that the size of advice needed to solve the two problems under investigation depends on three parameters: the number n of nodes, the number m of edges, and a crucial parameter μ called the *multiplicity* of the graph.[1] This parameter depends on the notion of the *view* from a node. Intuitively, this is the infinite tree with labeled ports, rooted at the given node, that would be obtained by a complete infinite exploration of the graph, if each visited node were attached as a new node in the tree (see the formal definition in Section 2). Since nodes of the graph are not labeled

[1] Our multiplicity parameter μ is related to the symmetricity σ of a graph, defined in [25]. We define multiplicity for a graph G with a particular port labeling, while $\sigma(G)$ is defined in [25] to be the maximum multiplicity over all port labelings of G.

and cannot be marked, and thus already visited nodes cannot be recognized on subsequent visits, the view from a node is the maximum information that can be obtained by an agent starting at this node and exploring the graph. It has been proved in [22] that if views of two nodes of a n-node graph are different, then their views truncated to level $n-1$ are also different. Hence, knowing any upper bound on n, the agent can learn in finite time which nodes have equal views and which do not. It is known (cf. [25]) that, for every node v of a graph, the number of nodes from which the view is identical as that from v is the same. This number is the multiplicity μ of the graph.

Our results. We give bounds on the minimum size of advice required by an agent both for the task of topology recognition and of spanning tree construction. (We focus on the feasibility of accomplishing these tasks and not on the complexity of their performance.) For $\mu = 1$ our bounds are asymptotically tight for both tasks and show that the minimum size of advice is very small: for an arbitrary function $\varphi = \omega(1)$ it suffices to give $\varphi(n)$ bits of advice to accomplish both tasks for n-node graphs, and $\Theta(1)$ bits are not enough. For $\mu > 1$ the minimum size of advice increases abruptly. In this case our bounds are asymptotically tight for topology recognition and asymptotically almost tight for spanning tree construction. We show that $\Theta(m \log \mu)$ bits of advice are enough and necessary to recognize topology in the class of graphs with m edges and multiplicity $\mu > 1$. For the second task we show that $\Omega(\mu \log(n/\mu))$ bits of advice are necessary and $O(\mu \log n)$ bits of advice are enough to construct a spanning tree in the class of graphs with n nodes and multiplicity $\mu > 1$. Thus in this case the gap between our bounds is always at most logarithmic, and the bounds are asymptotically tight for multiplicity $\mu = O(n^\alpha)$, where α is any constant smaller than 1.

Our results imply the following, somewhat surprising comparison of the importance of graph exploration in accomplishing the two considered tasks. For the task of topology recognition, the fact that the agent can explore the graph has a small impact on the required size of advice, if $\mu > 1$. Indeed, for any $\mu > 1$, there are $2^{O(m \log n)}$ port-labeled non-isomorphic graphs with n nodes, m edges and multiplicity μ. Hence without any exploration it would be enough to give $O(m \log n)$ bits of advice to recognize topology (by giving the index of the graph in some ordered list of all such graphs), and with the help of exploration the number of bits is $\Theta(m \log \mu)$. Thus the capability to explore the graph is "worth" at most a logarithmic factor in the size of advice (the largest difference occurring when μ is constant). By contrast, for the task of spanning tree construction, the possibility of exploring the graph may have a crucial impact on the required size of advice. Indeed, for any μ, there are at least $2^{\Omega(n/\mu)}$ graphs with n nodes and multiplicity μ, no pair of which has isomorphic spanning trees. Hence, without exploration, $\Omega(n/\mu)$ bits of advice would be necessary to solve the spanning tree construction problem. However, if the agent can explore the graph, only $O(\mu \log n)$ bits are sufficient to find a spanning tree. Thus, for any μ polylogarithmic in n, the capability of exploring the graph is "worth" an *exponential* decrease of the size of advice required for spanning tree construction.

Due to lack of space, proofs of several results (marked with ▲) are omitted.

Related work. Network algorithms with advice were studied, e.g., in [11–14, 18, 19, 23]. When advice is given by the oracle to the nodes, rather than to a mobile agent, the advice paradigm becomes closely related to that of *informative labeling schemes* [1, 7, 20, 21, 24]. In the advice paradigm the authors studied the minimum size of advice required for the solvability of the respective network problem or for its efficient solution. In [7] it was shown that giving appropriate 2-bit labels to nodes of a graph allows an agent to explore all graphs, and that with 1-bit labels an agent can explore all graphs of bounded degree. In [12] the authors compared the minimum size of advice required to solve two information dissemination problems using a linear number of messages. In [13] the authors established the size of advice given to a mobile agent, needed to break competitive ratio 2 of an exploration algorithm in trees. In [14] it was shown that advice of constant size permits to carry on the distributed construction of a minimum spanning tree in logarithmic time. In [11] the authors established lower bounds on the size of advice needed to beat time $\Theta(\log^* n)$ for 3-coloring of a cycle and to achieve time $\Theta(\log^* n)$ for 3-coloring of unoriented trees. It was also shown that, both for trees and for cycles, advice of size $\Omega(n)$ is needed to 3-color in constant time. In the case of [23] the issue was not efficiency but feasibility: it was shown that $\Theta(n \log n)$ is the minimum size of advice required to perform monotone connected graph clearing, using the minimum number of searchers. In [19] the authors studied radio networks for which it is possible to perform centralized broadcasting in constant time. They proved that $O(n)$ bits of advice allow to obtain constant time in such networks, while $o(n)$ bits are not enough. In [16] the trade-off between the size of advice and broadcasting time in trees was investigated, assuming that advice is given only to the source of broadcasting.

Computability in anonymous networks and feasibility of distributed tasks performed using message exchange in anonymous networks, without advice, have been studied, e.g., in [3, 6, 25]. Papers [10, 17] were devoted to investigating exploration in anonymous networks when agents have small memory. Rendezvous was considered in this context, e.g., in [15].

2 Terminology and Preliminaries

Networks are modeled as simple undirected connected graphs (without self-loops or multiple edges). Nodes of a graph are unlabeled and ports at a node of degree d are arbitrarily labeled $1, \ldots, d$. Thus each edge has two labels, one at each extremity. We do not assume any coherence between port labelings at different nodes. Port labels are visible to an agent walking in the graph.

Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic*, if there is a bijection $f : V_1 \longrightarrow V_2$, such that u is adjacent to v, if and only if $f(u)$ is adjacent to $f(v)$, and the port number corresponding to edge $\{u, v\}$ at node u is equal to the port number corresponding to edge $\{f(u), f(v)\}$ at node $f(u)$, for distinct $u, v \in V_1$.

An *oracle* is a function $\mathcal{O}$ from a class $\mathcal{G}$ of graphs to the set of finite binary strings. Given a graph $G \in \mathcal{G}$, the oracle gives to the agent the string $\mathcal{O}(G)$, called

the *advice*. The length of $\mathcal{O}(G)$ is called the *size* of advice. The agent (knowing $\mathcal{O}(G)$ but not G) is placed by an adversary in an arbitrary node of the graph G and executes its deterministic algorithm, for which $\mathcal{O}(G)$ can be a parameter. We say that an agent solves the topology recognition problem (respectively the spanning tree construction problem) in a class $\mathcal{G}$ of graphs, with advice of size s, if for any graph $G \in \mathcal{G}$, the agent, given advice of size s, outputs an isomorphic copy of G (respectively an isomorphic copy of a spanning tree of G) upon the execution of its algorithm.

We will use the following notion from [25]. Let G be a graph and v a node of G. The *view* from v is the infinite rooted tree $\mathcal{V}(v)$ with labeled ports, defined recursively as follows. $\mathcal{V}(v)$ has the root x_0 corresponding to v. For every node v_i, $i = 1, \ldots, k$, adjacent to v in G, there is a child x_i in $\mathcal{V}(v)$ such that the port number at v corresponding to edge $\{v, v_i\}$ is the same as the port number at x_0 corresponding to edge $\{x_0, x_i\}$, and the port number at v_i corresponding to edge $\{v, v_i\}$ is the same as the port number at x_i corresponding to edge $\{x_0, x_i\}$. Node x_i, for $i = 1, \ldots, k$, is now the root of the view from v_i. By $\mathcal{V}^t(v)$ we denote the view $\mathcal{V}(v)$ truncated to depth t. The following proposition is proved in [22].

Proposition 1. *For a n-node graph, $\mathcal{V}(u) = \mathcal{V}(v)$, if and only if $\mathcal{V}^{n-1}(u) = \mathcal{V}^{n-1}(v)$.*

The following notion was introduced in [25]. Let G be a graph. The *quotient graph* of G, denoted Q_G, is a (not necessarily simple) graph defined as follows. Nodes of Q_G correspond to sets of nodes of G which have the same view. For any (possibly equal) nodes x and y of Q_G, corresponding to sets U and V, respectively, there is an edge between x and y with labels p at x and q at y, if there exists an edge $\{u, v\}$ in G with $u \in U$, $v \in V$ and with ports p at u and q at v. Graphs G and H are called *equivalent*, if $Q_G = Q_H$.

If G has n nodes and Q_G has k nodes, then k divides n (cf. [25]) and the *multiplicity* μ of G is equal to n/k. It follows from [25] that computing views of all nodes of G is the maximum information that can be obtained from exploring G. More precisely we have the following proposition.

Proposition 2. *Let G and H be equivalent graphs with the quotient graph Q. Consider an agent starting from node v of G and from node w of H, where v and w correspond to the same node of Q. If the agent has the same advice for G and H, then the execution of its algorithm is identical in G and in H.*

The next proposition shows that any upper bound on the size of the graph G is sufficient to construct Q_G after exploring G.

Proposition 3. *Given any upper bound $\overline{n}$ on the number of nodes of a graph G, there exists an algorithm for an exploring agent that finds the quotient graph Q_G after exploring the graph G.* ▲

The next two propositions follow from the definition of the quotient graph.

Proposition 4. ([25]) *Let G be a graph with multiplicity μ. Let v_i be the node in Q_G corresponding to the set of nodes V_i of G having the same view. Consider*

an edge $e = \{v_i, v_j\}$ of Q_G with corresponding port numbers p and q. If E_e is the set of edges of G corresponding to e, then we have:

1. *if $i = j$ and $p = q$, then E_e forms a perfect matching in V_i,*
2. *if $i = j$ and $p \neq q$, then E_e is a set of pairwise disjoint cycles containing all the nodes in V_i,*
3. *if $i \neq j$, then E_e is a perfect matching in $V_i \cup V_j$ such that no edge has both endpoints either in V_i or in V_j.*

Proposition 5. ([25]) *Let G be any n-node graph of multiplicity μ and let T_{Q_G} be any spanning tree of the quotient graph Q_G. Then, there exist μ node-disjoint subtrees of G, each of which is isomorphic to T_{Q_G}.*

3 Graphs with Multiplicity 1

In this section we show that for graphs of multiplicity 1, $\omega(1)$ bits of advice are enough to accomplish both the topology recognition and the spanning tree construction tasks (an arbitrarily slowly growing function of the size of the graph will do), but $\Theta(1)$ bits are not enough for these tasks.

Lemma 1. *There exists no algorithm for an exploring agent that can find the number of nodes of the graph in the class of graphs of multiplicity* 1*, provided that $\Theta(1)$ bits of advice are given.* ▲

Corollary 1. *There exists no algorithm for an exploring agent that recognizes topology or constructs a spanning tree in the class of graphs with multiplicity* 1*, provided that $\Theta(1)$ bits of advice are given.* ▲

If $\mu = 1$, then G and Q_G are isomorphic. Hence Q_G contains all the information needed to recognize topology and to construct a spanning tree. Due to Proposition 3, it is sufficient to provide advice containing any upper bound on the number of nodes of G.

Let $\varphi\colon \mathbb{N} \to \mathbb{N}$ be any (computable) function diverging to infinity. (The divergence can be arbitrarily slow.) We construct an algorithm $\mathcal{A}_\varphi$ that uses advice of size at most $\varphi(n)$, where n is the number of nodes of the explored graph G, and determines the graph Q_G. There exists a non-decreasing (computable) function $\varphi^*\colon \mathbb{N} \to \mathbb{N}$ such that $\varphi^*(n) \leq \varphi(n)$ for all n, and φ^* diverges to infinity. Hence w.l.o.g. we may assume that φ itself is non-decreasing. For a n-node graph G the advice consists of the number $x = \varphi(n)$ and the information that $\mu = 1$. Then $\mathcal{A}_\varphi$ computes the smallest integer $\overline{n}$, such that $\varphi(\overline{n}) > x$. By the definition, $\overline{n}$ is an upper bound on n. By Proposition 3, using $\overline{n}$, the algorithm $\mathcal{A}_\varphi$ can find Q_G. It outputs this graph for the topology recognition task. The correctness of $\mathcal{A}_\varphi$ follows from the fact that Q_G is isomorphic to G. Since the agent using the algorithm $\mathcal{A}_\varphi$ recognizes the topology, it can also construct a spanning tree of G. Since the number $\varphi(n)$ can be coded on less than $\varphi(n)$ bits and one bit is enough to code $\mu = 1$, we have:

Theorem 1. *For any (computable) function* $\varphi\colon \mathbb{N} \to \mathbb{N}$ *such that* $\varphi = \omega(1)$, *there exists an algorithm for an exploring agent that uses advice of size at most* $\varphi(n)$ *and solves both the problem of topology recognition and that of the spanning tree construction in the class of* n*-node graphs of multiplicity* 1.

Together with Corollary 1, Theorem 1 gives the optimal size of advice to solve either of our two problems for graphs of multiplicity 1.

4 Topology Recognition for Graphs of Multiplicity > 1

In this section we establish asymptotically tight bounds on the size of advice needed for topology recognition in the class of graphs of any multiplicity $\mu > 1$.

Lemma 2. *For any* $\mu \geq 2$ *and* $m \geq 4\mu$ *there exist* $\mu^{\Omega(m)}$ *equivalent non-isomorphic graphs of multiplicity* μ. ▲

Theorem 2. *Every algorithm for an exploring agent that recognizes topology in the class of graphs with multiplicity* $\mu \geq 2$ *and* $m \geq 4\mu$ *edges requires* $\Omega(m \log \mu)$ *bits of advice.*

Proof. Let $\mathcal{G}$ be the class of equivalent non-isomorphic graphs constructed in the proof of Lemma 2. We have $|\mathcal{G}| \geq \mu^{cm} = 2^{cm \log \mu}$ for some constant $c > 0$. Suppose that an agent is able to recognize the topology of $G \in \mathcal{G}$ using $\lfloor cm \log \mu \rfloor - 1$ bits of advice. This advice partitions the class $\mathcal{G}$ into $k = 2^{\lfloor cm \log \mu \rfloor - 1} \leq \mu^{cm}/2$ disjoint classes $\mathcal{C}_1, \ldots, \mathcal{C}_k$ with the same advice for all graphs in the same class. Thus, there exists $i \in \{1, \ldots, k\}$ such that $|\mathcal{C}_i| > 1$. For different graphs $G, H \in \mathcal{C}_i$, graphs G and H are equivalent and, since the advice is the same for G and H, the agent produces the same output for both graphs, by Proposition 2. By Lemma 2, G and H are not isomorphic — a contradiction. □

Together with Theorem 2, the following result gives an asymptotically tight bound on the size of advice sufficient for topology recognition in the class of graphs with m edges and multiplicity $\mu \geq 2$.

Theorem 3. *For any* $\mu \geq 2$ *there exists an algorithm for an exploring agent that uses* $O(m \log \mu)$ *bits of advice and recognizes topology in the class of graphs with* m *edges and multiplicity* μ.

Proof. We prove the theorem by estimating the number of non-isomorphic equivalent graphs with m edges and of multiplicity μ. Consider one edge $e = \{v_i, v_j\}$ of the common quotient graph Q. Let V_i and V_j be sets of nodes of the equivalent graphs corresponding to v_i and v_j in Q. Sets V_i and V_j have size μ. By Proposition 4 we have the following possibilities:

1. $i = j$ and the port labels at the endpoints of e are identical. Then e corresponds to $O(n!)$ possible perfect matchings in the set V_i;
2. $i = j$ and the port labels at the endpoints of e are different. Then e corresponds to $\mu^{O(\mu)}$ possible collections of disjoint cycles with nodes in V_i;

3. $i \neq j$. In this case e corresponds to $O(n!)$ possible perfect matchings between the nodes in V_i and V_j.

There are at most $m/(\mu/2)$ edges in the quotient graph Q. Considering the worst case, there are at most $\mu^{O(m)}$ non-isomorphic graphs G that have the same quotient graph Q.

Given a graph G to explore, we provide the following advice to the agent: the number of nodes $n \leq m$ of G, and the index i of G in the list of all non-isomorphic n-node graphs with the quotient graph Q_G, ordered by using any fixed order provided with the algorithm. As shown above, this advice uses $O(m \log \mu)$ bits. By Proposition 3, the agent finds Q_G using the given n. Then, it reconstructs the ordered list of all n-node graphs with the quotient graph Q_G and returns the i-th graph in the list. □

The following observation shows that, for the problem of topology recognition, exploration is not worth much in terms of decreasing the size of advice. While $\Theta(m \log \mu)$ bits are needed with exploration, $O(m \log n)$ bits are enough without it, hence the ratio is at most logarithmic. This should be contrasted with the situation for spanning tree construction, where, as will be seen in Section 5, exploration is worth a lot.

Proposition 6. *Let G be a n-node graph with m edges. There exists an algorithm for an agent that returns the topology of G without performing any exploration, using $O(m \log n)$ bits of advice.* ▲

5 Spanning Tree Construction for Graphs of Multiplicity > 1

In this section we establish asymptotically almost tight bounds $\Omega(\mu \log(n/\mu))$ and $O(\mu \log n)$ on the size of advice needed for the spanning tree construction in the class of n-node graphs of any multiplicity $\mu > 1$. These bounds differ by at most a logarithmic factor and are asymptotically tight for multiplicity $\mu = O(n^\alpha)$, where α is any constant smaller than 1.

Lemma 3. *For every $\mu \geq 2$ and for every $n = r\mu$, where r is an integer, there exists a collection $\mathcal{G}$ of $(n/\mu)^{\Omega(\mu)}$ n-node equivalent graphs with multiplicity μ, such that no two graphs in $\mathcal{G}$ have isomorphic spanning trees.*

Proof. First we analyze the case when $\mu < 8$. Since the result is asymptotic, we can assume that n is sufficiently large, in particular $n > 5\mu$. Consider the quotient graph Q given in Fig. 2(b) (this example is for $\mu = 3$ and $r = 15$). (For even r take the quotient graph with $r-1$ nodes and add a pendant edge.) The edge e with port numbers 2 and 4 is unique in Q. The edge with port numbers 3 and 4 at distance i from e is called the i-th edge of Q, $i = 0, \ldots, (r-5)/2-1$. Define G_i by taking μ copies of Q and 'rearranging' the μ edges corresponding to the i-th edge of Q into a perfect matching between their left and right endpoints that makes G_i connected (see Fig. 2(a) for an example). Let $\mathcal{G} = \{G_0, \ldots, G_{(r-5)/2-1}\}$.

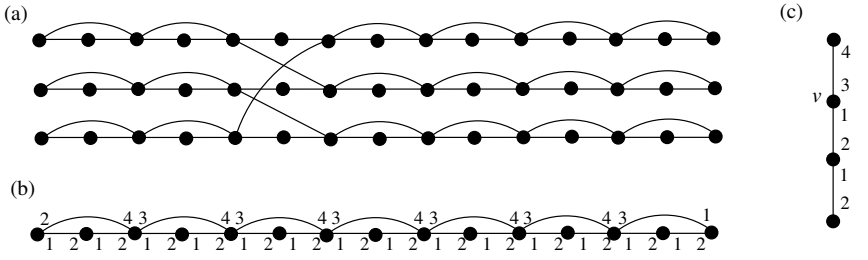


Fig. 2. (a) graph $G_1 \in \mathcal{G}$ with $\mu = 3$; (b) quotient graph Q; (c) a subpath of each spanning tree

If $i \neq j$, then G_i and G_j do not have isomorphic spanning trees. Indeed, a graph $G_k \in \mathcal{G}$ contains as a subgraph a cycle C with edges in the perfect matching and edges with port labels 1 and 2, connecting the endpoints of the edges in the matching. Moreover, exactly one edge of this cycle does not belong to any spanning tree of G, for otherwise the tree is not connected. Thus, in particular, each spanning tree of G_k contains a subgraph H of C depicted in Fig. 2(c). The node v of this subgraph uniquely identifies the length of the path in G_k with endpoint v and containing the edges which do not belong to C and have port labels 1 and 2. This determines the distance in the quotient graph between the edge corresponding to the perfect matching and the edge with port numbers $2, 4$. This distance equals k, which means that if the spanning trees of G_i and G_j are isomorphic, then $i = j$, which proves the claim. Moreover, $|\mathcal{G}| = (r-5)/2 = \Theta(n)$. This gives the bound from the lemma for each fixed μ, $2 \leq \mu < 8$.

Let now $8 \leq \mu \leq n$. For simplicity of presentation, assume that μ is divisible by 4. At the end of the proof we show how to handle the general case. Let $\mathcal{R}$ be any family of $(\mu/4)$-node paths rooted at an endpoint. Each edge e of R has a label $l(e) \in \{1, \ldots, r\}$. Two rooted paths R and R' are *similar*, if there exists a bijection f from the set of nodes of R into the set of nodes of R' such that $\{u, v\}$ is an edge of R if and only if $\{f(u), f(v)\}$ is an edge of R', and $l(\{u, v\}) = l(\{f(u), f(v)\})$ holds for all adjacent nodes u, v of R. (In other words, R and R' are similar, if they have the same length and identical edge labelings up to symmetry.) For a rooted path R, we use notions of parent and child, as in any rooted tree. For a given rooted path $R \in \mathcal{R}$ we construct a graph G_R of multiplicity μ.

Before defining G_R for $R \in \mathcal{R}$, we introduce the basic building block, called the *component*, used to construct the graphs G_R. Each component has $4r$ nodes. The component together with its quotient graph Q are given in Figs 3(a) and 3(b), respectively. The graph Q is also the quotient graph of each final graph G_R constructed below. We distinguish in the component an arbitrary path on r nodes connected by edges with port labels in $\{3, 4\}$. We will call this path the *leading* path of the component. The nodes of Q are denoted by $z_1, \ldots, z_r$ (see Fig. 3(b)). Each node v of R is represented by a copy of the component in G_R,

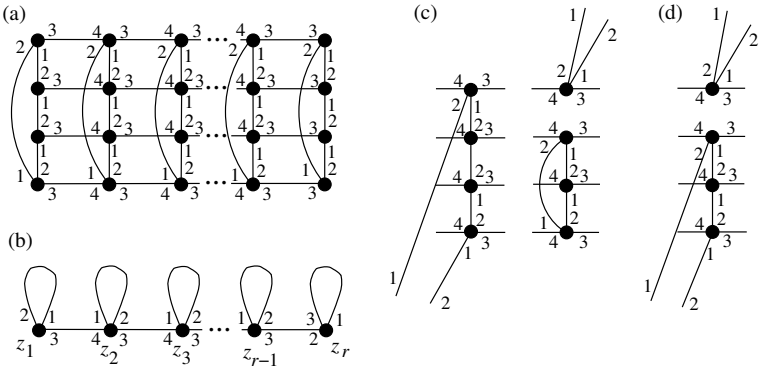


Fig. 3. (a) the component; (b) the quotient graph of the component and of G_R for each $R \in \mathcal{R}$; (c),(d) all possible connections between different components

denoted by $G_R(v)$. For any nodes u and v in R, such that u is the parent of v, we define $c(u, v)$ to be 3 if u has a parent w and $l(\{w, u\}) = l(\{u, v\})$, and to be 4 otherwise. If u is the parent of v in R, then we connect $G_R(u)$ with $G_R(v)$ in such a way that $c(u, v)$ nodes in $G_R(u)$ corresponding to $z_{l(\{u,v\})}$ in Q form a cycle together with the unique node corresponding to node $z_{l(\{u,v\})}$ in Q that belongs to the leading path of $G_R(v)$. Fig. 3(c) depicts the four nodes of a component $G_R(v)$, corresponding to a node z_i of Q, and connections representing the case when v has exactly one incident edge with label i and this edge connects v with its child, or its parent. Fig. 3(d) depicts these nodes and connections in the case when v has two incident edges with label i. By construction, the graph G_R is connected. Moreover, G_R has multiplicity μ, for each $\mu \neq n/2$. For $\mu = n/2$, a similar construction can be used, except that it has to be ensured that each "horizontal" edge in the component has different labels at the endpoints. The rest of the proof is for $\mu \neq n/2$, the case $\mu = n/2$ being analogous. See Fig. 4 for an example of the construction of G_R. Fig. 4(a) depicts a path R with 4 nodes. We take $\mu = 16$ and $n = 48$. The quotient graph Q given in Fig. 4(b) has $r = 3$ nodes. The graph G_R is shown in Fig. 4(c).

We define $\mathcal{G}(\mathcal{R}) = \{G_R \colon R \in \mathcal{R}\}$. In order to prove the lower bound stated in the lemma, we count the number of graphs in $\mathcal{G}(\mathcal{R})$ that do not have isomorphic spanning trees. The proof is in two steps. First we show that if R is not similar to R', then G_R and $G_{R'}$ cannot have isomorphic spanning trees. This will follow from the path-like structure of the graphs in $\mathcal{G}(\mathcal{R})$. In this way, we reduce our task to estimating the number of non-similar paths that can form a family $\mathcal{R}$. The proof of the following claim is in the Appendix.

Claim. If graphs G_R and $G_{R'}$ have isomorphic spanning trees, then R and R' are similar. ▲

Let $\mathcal{R}$ contain all $(\mu/4)$-node pairwise non-similar paths. Since $\mu \geq 8$, $\mathcal{R} \neq \emptyset$. There are $(n/\mu)^{\Omega(\mu)}$ such paths. By Claim 5, the family $\mathcal{G}(\mathcal{R})$ consists of

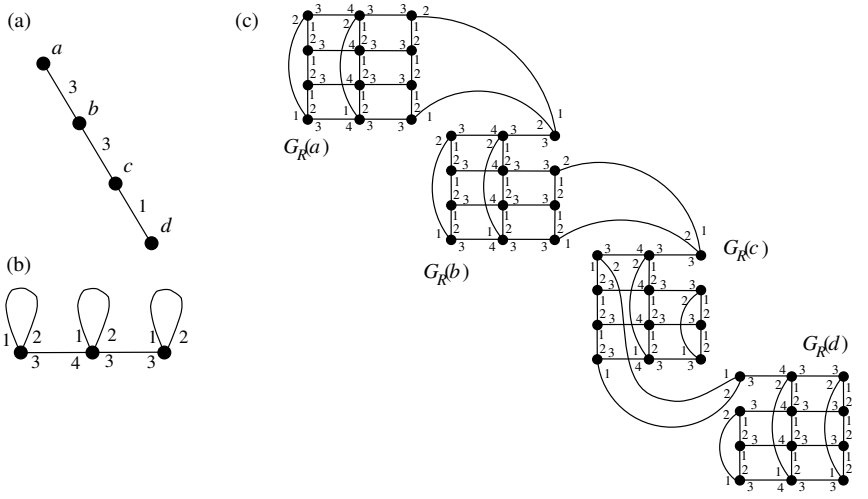


Fig. 4. (a) a rooted tree $R \in \mathcal{R}$; (b) the quotient graph Q; (c) the graph G_R

$(n/\mu)^{\Omega(\mu)}$ equivalent n-node graphs of multiplicity μ that do not have isomorphic spanning trees.

Note that if $4r$ does not divide n, then we obtain the bound from the lemma by making one component of size $4r + (n \mod (4r))$. □

Theorem 4. *For every $\mu \geq 2$ and every $n = r\mu$, every algorithm for an exploring agent that constructs a spanning tree in the class of n-node graphs of multiplicity μ requires $\Omega(\mu \log(n/\mu))$ bits of advice.* ▲

The following result gives an almost matching upper bound on the size of advice sufficient for spanning tree construction.

Theorem 5. *For any $\mu \geq 2$ there exists an algorithm for an exploring agent that uses $O(\mu \log n)$ bits of advice and constructs a spanning tree in the class of n-node graphs of multiplicity μ.*

Proof. The oracle provides advice consisting of the number n of nodes and of a sequence of $\mu - 1$ edges that the agent uses to obtain a spanning tree, where each edge is given as four integers in $\{1, \ldots, n\}$ (the endpoints and the port numbers at the endpoints). By Proposition 3, the agent finds Q_G using n. The quotient graph Q_G is connected, for otherwise G itself would not be connected. Let T_{Q_G} be the first spanning tree of Q_G in some fixed order. The agent finds T_{Q_G}. We need to assign a canonical order to nodes of this tree. If T_{Q_G} has a central node or a central edge but is non-symmetric (considering port labelings), nodes of T_{Q_G} can be canonically ordered starting from the central node of this tree, or from one of the endpoints of the central edge, following an Eulerian tour of the tree starting by the smallest port and leaving every node by the next port after the one by which the node was entered. If the tree has a central edge

and is symmetric, each of the halves of it can be ordered as above. Note that T_{Q_G} has n/μ nodes. By Proposition 5, the graph G contains μ node-disjoint subtrees T_i such that each T_i is isomorphic to T_{Q_G}, $i = 1, \ldots, \mu$. Thus, the total number of nodes (resp. edges) in all the subtrees T_i is n (resp. $n - \mu$). Since G is connected, there exist $\mu - 1$ edges e_j, $j = 1, \ldots, \mu - 1$, such that the subgraph of G containing exactly the edges of the subtrees T_i, $i = 1, \ldots, \mu$, and those edges e_j, is a spanning tree T of G. To finish the construction of the spanning tree T of G, the agent connects the subtrees T_i, $i = 1, \ldots, \mu$, using the edges e_j, $j = 1, \ldots, \mu - 1$, which are given as advice. Endpoints of the edges e_j are given to the agent using the canonical order described above. □

We illustrate the algorithm described in the proof of Theorem 5 in the following example. Let G be the graph given in Fig. 5(a), where $n = 16$, $\mu = 4$. This implies that the quotient graph must have $n/\mu = 4$ nodes. The agent finds the quotient graph Q_G and one of its spanning trees, given respectively in Figs 5(b) and 5(c). The advice, besides n, codes the three edges added in order to connect the copies of T_{Q_G} into a spanning tree T of G, shown in Fig. 5(d).

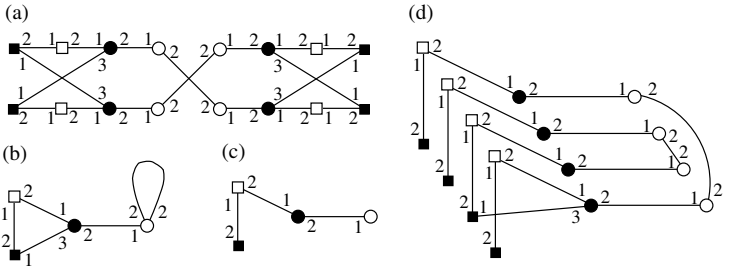


Fig. 5. (a) a graph G with $\mu = 4$ and $n = 16$; (b) Q_G; (c) T_{Q_G}; (d) T obtained from advice and $\mu = 4$ copies of T_{Q_G}

Theorems 4 and 5 imply that our bounds on the size of advice for spanning tree construction are asymptotically tight and equal $\Theta(\mu \log n)$, when $\mu = O(n^\alpha)$, for any constant $\alpha < 1$.

Our final observation shows that, as opposed to the task of topology recognition, in the spanning tree construction the ability of exploring the graph is worth a lot in terms of the size of advice. While $O(\mu \log n)$ bits are enough to accomplish this task with exploration, $\Omega(n/\mu)$ bits are needed without it. Hence, for μ polylogarithmic in n, the ability of exploring is 'worth' an exponential decrease in the size of advice required for constructing a spanning tree. As we have seen in Section 4, this is very different from what happens for the task of topology recognition.

Proposition 7. *An agent that returns a spanning tree for the class of n-node graphs of multiplicity μ without performing any exploration requires $\Omega(n/\mu)$ bits of advice.* ▲

6 Conclusion

We established asymptotically tight bounds on the minimum size of advice required to solve the topology recognition problem. For the spanning tree construction problem, our bounds are asymptotically almost tight: $O(\mu \log n)$ and $\Omega(\mu \log(n/\mu))$. Closing this (at most logarithmic) gap is a natural open problem. In this paper we focused only on the feasibility of these two map drawing tasks. It is natural to ask what are the trade-offs between the size of advice and the efficiency (time) of accomplishing each task. We also did not impose any restriction on the memory size of the agent. Another open problem is to study trade-offs between the size of advice and the size of the memory of the agent required to accomplish a particular network task.

References

1. Abiteboul, S., Kaplan, H., Milo, T.: Compact labeling schemes for ancestor queries. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 547–556 (2001)
2. Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM Journal on Computing 29, 1164–1188 (2000)
3. Angluin, D.: Local and global properties in networks of processors. In: Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980), pp. 82–93 (1980)
4. Bender, M.A., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: Exploring and mapping directed graphs. Information and Computation 176, 1–21 (2002)
5. Betke, M., Rivest, R., Singh, M.: Piecemeal learning of an unknown environment. Machine Learning 18, 231–254 (1995)
6. Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 33–47. Springer, Heidelberg (2001)
7. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 335–346. Springer, Heidelberg (2005)
8. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. Journal of Graph Theory 32, 265–297 (1999)
9. Dessmark, A., Pelc, A.: Optimal graph exploration without good maps. Theoretical Computer Science 326, 343–362 (2004)
10. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. Journal of Algorithms 51, 38–63 (2004)
11. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed computing with advice: information sensitivity of graph coloring. Distributed Computing 21, 395–403 (2009)
12. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Oracle size: a new measure of difficulty for communication problems. In: Proc. 25th Ann. ACM Symposium on Principles of Distributed Computing (PODC 2006), pp. 179–187 (2006)
13. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Tree exploration with an oracle. Information and Computation 206, 1276–1287 (2008)

14. Fraigniaud, P., Korman, A., Lebhar, E.: Local MST computation with short advice. In: Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2007), pp. 154–160 (2007)
15. Fraigniaud, P., Pelc, A.: Deterministic rendezvous in trees with little memory. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 242–256. Springer, Heidelberg (2008)
16. Fusco, E., Pelc, A.: Trade-offs between the size of advice and broadcasting time in trees. In: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), pp. 77–84 (2008)
17. Gasieniec, L., Pelc, A., Radzik, T., Zhang, X.: Tree exploration with logarithmic memory. In: Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007), pp. 585–594 (2007)
18. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. In: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), pp. 210–219 (2001)
19. Ilcinkas, D., Kowalski, D., Pelc, A.: Fast radio broadcasting with advice. In: Shvartsman, A.A., Felber, P. (eds.) SIROCCO 2008. LNCS, vol. 5058, pp. 291–305. Springer, Heidelberg (2008)
20. Katz, M., Katz, N., Korman, A., Peleg, D.: Labeling schemes for flow and connectivity. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), pp. 927–936 (2002)
21. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC 2005), pp. 9–18 (2005)
22. Norris, N.: Universal covers of graphs: Isomorphism to depth N-1 implies isomorphism to all depths. Discrete Applied Mathematics 56, 61–74 (1995)
23. Soguet, D., Nisse, N.: Graph searching with advice. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 51–65. Springer, Heidelberg (2007)
24. Thorup, M., Zwick, U.: Approximate distance oracles. J. ACM 52(1), 1–24 (2005)
25. Yamashita, M., Kameda, T.: Computing on anonymous networks: Part I - characterizing the solvable cases. IEEE Trans. Parallel and Distributed Systems 7, 69–89 (1996)

Network-Aware Distributed Algorithms: Challenges and Opportunities in Wireless Networks
(Invited Lecture Summary)

Nitin Vaidya

University of Illinois at Urbana-Champaign

Wireless networks differ from typical wired networks in several significant ways. Wireless channel is a broadcast medium, resulting in interference between simultaneous transmissions on the wireless channel. Wireless channel conditions vary over time and space, making it necessary to adapt to the time-varying conditions. The wireless signals can propagate along different paths from a transmitter to a receiver, resulting in channel fading.

These characteristics of the wireless medium can degrade performance. However, the same characteristics also offer potential performance benefits. For instance, the broadcast nature of the wireless medium may allow the nodes to overhear transmissions intended for other hosts. Fading may degrade instantaneous performance of a given user when the channel is in fade, however, multi-user diversity schemes can exploit fading to improve performance. Many wireless communications mechanisms have been designed over the years to realize such performance benefits.

Not surprisingly, being aware of the underlying network characteristics can help improve performance of distributed algorithms as well. This talk introduces some simple examples of distributed algorithms that can benefit by paying attention to the wireless network characteristics, and describes some topics for future research.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, p. 343, 2010.

Connectivity Problem in Wireless Networks*

Dariusz R. Kowalski and Mariusz A. Rokicki

University of Liverpool, United Kingdom
{D.Kowalski,M.A.Rokicki}@liverpool.ac.uk

Abstract. We study the complexity of the following connectivity problem in wireless networks: for a given placement of n nodes in the plane, the goal is to compute a channel and power assignment that forms strongly connected communication structure spanning all nodes. The complexity measure is the total number of assigned channels, and the goal is to minimize this number. We work with two signal inference models: *Geometric Radio Networks* (GRN) and *Signal to Interference Plus Noise Ratio* (SINR). We show a generic polynomial-time transformation from the wide class of separable assignments in GRN to assignments in the SIRN model. This transformation preserves asymptotic complexity, i.e., the number of channels used in the assignments. In this way we show an assignment, constructed in polynomial-time, guarantying connectivity in the SINR model by using only $O(\log n)$ channels, which is an improvement over the best previous result $O(\log^2 n)$ presented in [21].

1 Introduction

Wireless networks have become very popular for their numerous advantages from the user perspective. On the other hand, these properties, attractive to the users, pose several challenges to the designers of wireless network architectures and protocols. One of the main such problems is how to schedule interfering transmissions to accomplish specific communication tasks. In wireless networks where only one or a constant number of channels are used, the most popular way to resolve colliding transmissions is to use time/code division; this is often implemented using complex coding techniques, but in any case there are several provable limitations incurred in this setting, c.f., [7]. One of the solutions to this problem is to use a slightly larger number of transmission channels (though within a certain "reasonable" limit, e.g., logarithmic) and vary the transmission powers at the nodes.

Among the basic communication tasks there is a connectivity problem: how to schedule channels and power transmissions in order to achieve strong connectivity for a given placement of wireless nodes. More precisely, we would like to minimize the total number of channels needed to form a strongly connected network over a given set of n nodes located in the Euclidean plane. In order

* This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/G023018/1, EP/H018816/1].

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 344–358, 2010.

to achieve this goal, transmission powers need to be assigned properly. Note however that in this work we do not consider simultaneous optimization of energy consumption; assignment of transmission powers is used only to help in minimizing the number of used channels.

We consider two models of signal inference: Geometric Radio Networks model (GRN) and Signal to Interference plus Noise Ratio (SINR) model. In the GRN model we assume that each node has a certain transmission range. The transmission range of a node depends on its transmission power. In this model a transmission from a node v is successfully received at a node w if w is in v's transmission range and no other node that reaches w by its range transmits concurrently. In the SINR model the power of transmission fades with the distance. The transmission can be successfully received if its signal power is sufficiently strong against the total signal power of the concurrent transmissions and the background noise, all measured at the receiver. These two models seem to be quite different, in the sense that there are examples in which a successful transmission in the GRN model dose not imply a successful transmission in the SINR model and vice versa. However, we will show that in case of the connectivity problem on the plane, these two models have similar power, up to a logarithmic factor, in terms of the number of channels that are to be used.

Previous and related work. The SINR model was introduced by Gupta and Kumar [15], where the authors investigated the capacity of wireless networks. The connectivity problem considered in our work was abstracted and studied in the context of SINR model by Moscibroda and Wattenhofer [23]. The authors showed how to compute in polynomial time an assignment of transmission powers and $O(\log^4 n)$ channels to achieve strong connectivity of the resulting reachability graph. They also proved that allowing only uniform or linear power transmissions yields $\Omega(n)$ channels to be used for some node placement. The upper bound $O(\log^4 n)$ was later improved to $O(\log^3 n)$ by Moscibroda et al. [24] and to $O(\log^2 n)$ by Moscibroda [21]. In [4] the authors studied the connectivity problem in one- and two-dimensional grids with the assumption of the uniform power transmissions. In case of one dimensional grids, it was showed that a constant number of channels is sufficient to form a strongly connected network, provided $\alpha > 1$ (α is the path-loss exponent in the SINR model). The same result is proved for two dimensional grids for $\alpha > 2$. For small parameter α, the authors showed that $O(\log n)$ channels allow to form a strongly connected network, and at least $\Omega(\log n / \log\log n)$ channels are necessary in two dimensional grids.

In geometric radio networks, the complexity of the connectivity problem is $\Theta(\log n)$, as proved in [10]. This result holds for any placement of nodes in the plane.

The relations between the geometric radio model and the SINR model were studied in [20]. Assuming that nodes are distributed uniformly at random in the square unit, the authors showed that it is possible to emulate any UDG protocol on SINR with a polylogarithmic stretch factor, with very high probability.

A related problem of scheduling a given set of communication links (with or without a'priori specified power assignment) in the SINR model was also intensively studied, see e.g., [9,22]. The goal is to minimize the length of a transmission schedule in which there is a successful transmission along each of the requested links. In [14], the authors showed that link scheduling problem is NP-complete. A polynomial time algorithm with a constant approximation factor was given in [16]. It is an improvement over the $O(\log n)$-approximation given in [13]. Fanghanel et al. [9] showed that oblivious power assignment requires linear number of channels for some set of links to be realized. Note that the main difference between scheduling and connectivity problems is that in the former one, a set of links is given as a part of the input, while in the latter it needs to be computed as a part of the solution. In [3], the authors investigated reception zones under the SINR model. In particular, they showed that for $\alpha > 1$ and for uniform power assignment the reception zones have to be convex.

The related broadcast problem was also studied in the context of geometric radio networks, c.f., [8]. The goal of broadcasting is to find the shortest connected schedule that disseminates a message from one distinguished node to all other nodes. In the centralized setting, the best broadcasting protocol works in $O(D \log^2 \frac{r_{max}}{r_{min}})$ rounds [11], improving one of the results from [8]. Here $O(\log^2 \frac{r_{max}}{r_{min}})$ can be viewed as an average number of channels to perform broadcast without interference. In [18], the authors studied the convergecast problem in *ad-hoc* geometric radio networks, where the transmission powers can be modified during the execution.

In general graph-based model of radio networks (c.f., [5]), the fastest known broadcasting protocol in directed networks requires $O(D \log^2 n)$ rounds as shown in [6], where D is the diameter of the network (directed networks corresponds to non-uniform transmission powers). In [1] it was shown that there exists a bipartite graph that requires $\Omega(\log^2 n)$ rounds to guarantee that each node in one layer receives a message from at least one node from the other layer. Thus, in case of networks with constant diameter, the upper bound matches the lower bound. The fastest known broadcasting protocol in symmetric networks requires at most $O(D + \log^2 n)$ rounds [12,19].

Our results. The main result of this work is a generic polynomial-time transformation of a wide class of solutions for the connectivity problem, satisfying so called "separability" condition, from the model of geometric radio networks to the SINR model. The obtained assignment requires asymptotically the same number of channels as the original assignment for geometric radio networks. In particular, we show that the polynomial-time assignment based on [10], developed in the context of GRN model, is in the class that can be transformed to assignments in the SINR model (preserving the asymptotic number of channels). This gives a polynomial-time construction of an assignment using $O(\log n)$ channels in the SINR model, which is an improvement over the best previous $O(\log^2 n)$ bound obtained in [21].

We start by introducing the considered wireless models and the connectivity problem in Section 2. The properties of the assignment in the GRN model are

studied in Section 3. The generic transformation to the SINR model is given in Section 4. We discuss the obtained results and open directions in Section 5.

2 Models of Wireless Networks and the Connectivity Problem

We are given n nodes located in the Euclidean plane (i.e., with the Euclidean metric $d(\cdot,\cdot)$). Each node is equipped with a radio transmitter and receiver. Communication is scheduled in synchronous rounds (slots). If a node v decides to transmit in a round t, it chooses a transmission power $P_v(t)$ and a frequency channel $c_v(t)$ to be used. In this work, we consider static power and channel assignments, i.e., the power and the channel of node v do not depend on a round number t; therefore in the remainder we will use notation P_v, c_v without parameter t. Signal attenuation is proportional to the inverse of the distance to the power α, where α is typically assumed to be a constant in the range $(2, 6)$. Signals transmitted on different channels are not interfering, in the sense that receiving data transmitted on one channel does not depend on simultaneous transmissions on other channels. The feedback received from wireless transmissions in a round by node w depends on various physical constraints. In this work we consider two such interference models.

Geometric Radio Networks (GRN) model. In the *geometric radio network model*, also called the *radio model* or *GRN* for short, a node w receives a transmission from node v in round t only if

(i) node v transmits in round t and $P_v \geq d(v,w)^\alpha$, and
(ii) if a node $u \neq v$ transmits in round t then $P_u < d(u,w)^\alpha$.

In other words, a message sent by node v is received at node w if node v is the only transmitting node satisfying $(P_v)^{1/\alpha} \geq d(v,w)$. If there are at least two nodes v, u satisfying condition (i) in round t then we say that a collision occurs in node w, which implies that no message is successfully delivered to node w in this round.[1] Note that this model specification is equivalent to the classic geometric radio networks, c.f., [8], where the *transmission range* R_v of a node v corresponds to the threshold value $(P_v)^{1/\alpha}$ in our equivalent definition. In particular, the *Unit Disk Graphs (UDG)* model (c.f., [17]) is a geometric radio network with uniform ranges (i.e., uniform transmission powers in our formulation). Therefore, when considering GRN model, we will argue in terms of transmission ranges, and the corresponding reachability graph defined as follows. We say that v is w's in-neighbor, and w is v's out-neighbor, in the reachability graph defined by the set of nodes and their transmission powers, if the transmission range of v, i.e., $R_v = (P_v)^{1/\alpha}$, satisfies $R_v \geq d(v,w)$.

[1] In the setting considered in this work, the capability of distinguishing collisions from background noise does not influence the complexity, therefore we do not make any specific assumption about so called *collision detection*.

Signal to Interference plus Noise Ratio (SINR) model. In the *Signal to Interference plus Noise Ratio model*, also called the *SINR model* for short, a node w receives a transmission from node v in round t only if

$$\frac{\frac{P_v}{d(v,w)^\alpha}}{N + \sum_{u \neq v, w, u \in \mathcal{T}} \frac{P_u}{d(u,w)^\alpha}} \geq \beta \ ,$$

for some noise parameter $N > 0$ and threshold β, both being part of the model setting, and assuming that $\mathcal{T}$ is the set of nodes transmitting in round t. We assume that $\beta \geq 1$; if it is smaller than 1, then it is enough to make our construction for threshold $\beta' = 1$ and the result, i.e., all SINR equations, will also hold for original β.

Connectivity problem. Consider n nodes located in the plane. A *CP-assignment* σ is a function assigning to each node v a channel c_v and a transmission power P_v. A channel is a non-negative integer, while a transmission power is a real non-negative number. Note that we may assume that transmission powers are in fact rational non-negative numbers, since there is a finite number of stations and thus their initial powers could be slightly re-scalled to make the values rational, without violating properties of the radio or the SINR system.

We say that a CP-assignment σ *realizes link* (v, w) in a given model, for any nodes $v \neq w$, if a message transmitted by v with transmission power P_v and channel c_v defined by σ is always successfully received by node w, regardless of the behavior of other nodes (i.e., whether some of them simultaneously transmit using channels and transmission powers as specified by σ). Here we consider only two models: GRN and SINR.

We say that a CP-assignment σ solves the CP-connectivity problem in a given model if the set of links realized by σ in this model forms a strongly connected directed graph spanning all the nodes. We call such CP-assignment *admissible* for this model. The measure of quality of the outputted CP-assignment is the number of different channels used by σ, and the goal is to minimize this number. In this work we consider centralized setting, i.e., where each node knows exact locations of all other nodes and all nodes perform the same algorithm. Our goal is to give a polynomial-time centralized deterministic algorithm, in terms of the size of the input of the problem, finding a CP-assignment with a small number of channels.

Separability and other useful definitions. We call a CP-assignment *separable* if it satisfies the following condition:

> for any two nodes u, v with the same assigned channel, the distance between them is at least the maximum of their assigned powers to the power $1/\alpha$, i.e., $d(u, v) \geq \max\{(P_u)^{1/\alpha}, (P_v)^{1/\alpha}\}$.

We will use the notion of separability for CP-assignments admissible for the radio model. We show in Section 3 and an example of efficient separable assignment for the GRN model.

Throughout this work we distinguish objects used in the context of the radio model from the ones used for the SINR model by using a star symbol in the notation for SINR, i.e., σ^*, P_v^*, c_v^*, etc.

We denote by $[x]$ the set $\{0, 1, \ldots, x-1\}$. All logarithms without specified base are assumed to be to the base 2. In the analysis, we also assume that n is sufficiently large, as we focus on asymptotic notation.

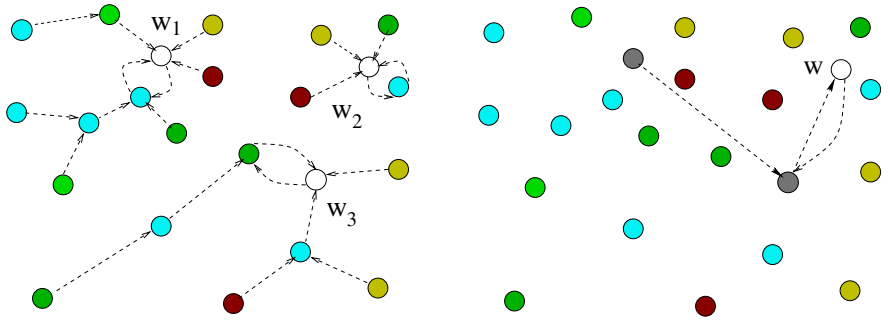
3 Connectivity and Separability in the GRN Model

In this section we show that the algorithmic method developed in [10] gives a separable CP-assignment for any given placement of n nodes, using at most $18 \log n + 1$ different channels. Moreover, it can be computed in polynomial time. We start with describing the algorithm from [10], and simultaneously we provide high-level justification of the correctness and estimation of the number of channels used. Finally, we give formal argument why the resulting assignment is separable. This will give a sample input to the transformation to the SINR model described in Section 4.

Consider the closest-neighbor network in which each node adjusts its transmission power so that its transmission range is equal to its closest neighbor (according to the Euclidean distance). It can be proved that the in-degree of each node in the closest-neighbor network is at most 6.

The algorithm works in phases. In each phase i only some subset $A_i \subseteq V$ of nodes is considered. Initially all the nodes are considered, i.e., $A_0 = V$. In the ith phase, each considered node $v \in A_i$ chooses its closest neighbor $u \in A_i$ and adjusts its transmission power so that its transmission range is exactly $d(v, u)$, i.e., P_v is set to $d(v, u)^\alpha$. We can see that this procedure creates a set of directed components in the reachability graph, c.f., Figure 1. Each directed component has a sink w that can be reached, via intermediate nodes, from each node of the component. (In case of more than one such node w in the component, we choose one arbitrarily and associate the name "sink" only with this node, with respect to the component.) Let us partition each component of the reachability graph into layers, where layer L_j contains nodes of j hops from the sink, i.e., nodes from which there is a directed path of length i to the sink, but not any shorter one. Recall that the in-degree of each node in a component is at most 6. Therefore, one can assign 6 channels so that each node in layer L_{j+1} can successfully transmit its message to some node in layer L_j, mainly, to the node in L_j within its transmission range (c.f., [12]). Note however that collision may still be caused by nodes located in any three consecutive layers. In order to avoid it, we need to use $3 \cdot 6 = 18$ different channels per phase. More precisely, we allocate channels to the nodes in layer L_j as follows: we use the first 6 available channels if $j = 1 \mod 3$, next 6 available channels for $j = 2 \mod 3$, and the last 6 available channels for $j = 0 \mod 3$.

This way our channel assignment guarantees that each node from each layer can reach its sink, without causing any collision. In phase $i + 1$, only sink nodes



(a) Phase 1. (b) Phase 2.

Fig. 1. Example of a radio network in which the root is found in two phases. In the first phase, Figure 1a, there are three directed components formed with sinks w_1, w_2, w_3 (white nodes). Only sinks are active in the second phase. In the second phase, Figure 1b, there is only one component (consisting of the sinks w_1, w_2, w_3) and the sink $w = w_2$ of this component becomes the root of the whole network. The root is assigned a unique channel and its transmission range reaches all the nodes in the network.

of the components from phase i are considered, i.e., the set A_{i+1} contains all the sink nodes from phase i. In a new phase, a new set of (at most 18) channels is used. The phases are iterated until there is only one sink node remaining. This node becomes the root of the whole network, and to guarantee its superiority we assign a new channel to it. We can see from the construction that the root is reachable by all the nodes in the network, via intermediate nodes. The root gets assigned one extra channel and sufficient power to reach all the nodes in the network. This way we achieve channel and power assignment with guarantees that there is a directed path between each pair of nodes in the graph consisting of realizable directed point-to-point links. Note that the algorithm is clearly polynomial.

We estimate the number of channels used in the construction. In each phase we reduce the number of active nodes by at least half. So by at most $\log n$ phases only the root remains. In each phase we use at most 18 channels. Thus, we use at most $18 \log n$ channels plus one extra channel for the root to reach all the nodes.

It remains to show that the obtained CP-assignment is separable. Since in each phase different channels are used, it is enough to consider channel assignment in a single phase. Consider two nodes v, u assigned the same channel. If one of them, say v, had range bigger than the distance between the two nodes, then node u would be closer to node v than the closest node (as the range assigned to v is the same as the distance to the closest node). This is however a contradiction, which means that the distance must be at least the maximum of both ranges (recall that ranges are defined as transmission powers to the power $1/\alpha$).

4 Connectivity in the SINR Model

We start with describing a general transformation from separable radio CP-assignments to SINR CP-assignments, then we analyze its properties and conclude about the resulting $O(\log n)$ channel assignment in the SINR model.

4.1 Transformation from Separable Radio CP-Assignments to SINR Assignments

We show how to find a CP-assignment σ^* admissible for the SINR model based on a given separable CP-assignment σ, preserving asymptotically the number of used channels.

The construction of CP-assignment σ^* admissible for the SINR model proceeds in four stages. First, if the maximum distance between any two nodes is bigger than 1, we scale down all distances to make them at most 1. Under this assumption, we deliver a formula on the power assignment, and next we construct channel allocation — all based on a given separable CP-assignment admissible for radio networks. Finally, if scaling was used in the first stage, we re-scale all distances up to the original ones and update the constructed power assignment, while keeping the channel assignment unchanged.

Let b be a constant depending on the system parameters α, β such that $b^{(\alpha-2)/2} \geq 16^{\alpha+3} \cdot \zeta(\alpha/2) \cdot 3\beta$, where $\zeta(\cdot)$ is the Riemann zeta function (c.f., [2]).

Scaling (optional). Let $d_{\max}$ be the maximum distance between any two nodes. If $d_{\max} \leq 1$ then we proceed to the next construction stage. Otherwise we scale distances by dividing each of them by $d_{\max}$. Parameter $d_{\max}$ is called a *scaling factor*. We also scale parameter N by multiplying it by $d_{\max}^{\alpha}$.

Obtaining a separable CP-assignment in the GRN model. Let σ denote a given separable CP-assignment in the GRN mode, obtained for the node placement satisfying $d_{\max} \leq 1$. Let P_u be the power and c_u be the channel assigned to node u by σ, and let $R_u = (P_u)^{1/\alpha}$ be the transmission range of node u in the radio model according to σ. We denote by R_{min} the minimum and by R_{max} the maximum of values R_u, over all nodes u. For the purpose of the presentation of the material in this section, it is more convenient to use ranges R_u defined by σ instead of actual transmission powers P_u assigned by σ.

Setting transmission powers in σ^*. We define node powers P_u^* in σ^* as follows. Let γ be equal to $3N\beta \cdot (R_{max})^{(\alpha-2)/2}$. We set $P_u^* = \gamma \cdot (R_u)^{(\alpha+2)/2}$.

Setting transmission channels in σ^*. Assume that all channels in σ are disjoint integers in $[\kappa] = \{0, 1, \ldots, \kappa - 1\}$. In order to assign channels to nodes in the CP-assignment σ^*, based on channel assignment from σ, we proceed in three steps. We first impose grid structures in the plane, then we color grids cells, and finally we assign channels to the nodes based on the channel assigned by σ, on the grid-cell coloring and on the transmission ranges in σ.

Grids $\mathcal{G}_{2^i r}$. Let r be equal to $R_{min}/\sqrt{2}$. Let $\mathcal{G}_r$ be a grid cutting the plane into $r \times r$ squares, also called *cells*, in such a way that each node is inside some square. Such grid exists since the number of points is finite. Note that by separability of CP-assignment σ and by the definition of r, there is at most one node in each cell of $\mathcal{G}_r$. We define grids $\mathcal{G}_{2^i r}$, for integers $i > 0$, by induction. Suppose we have already defined grid $\mathcal{G}_{2^{i-1} r}$ partitioning the plane into squares of size $(2^{i-1} r) \times (2^{i-1} r)$, for some $i > 0$. In order to construct grid $\mathcal{G}_{2^i r}$, we partition the set of squares in $\mathcal{G}_{2^{i-1} r}$ into disjoint groups, each containing four squares sharing the same corner point, and then each group is replaced by a different cell in $\mathcal{G}_{2^i r}$ being a union of the four squares in the group. It follows that the grid $\mathcal{G}_{2^i r}$, for every integer $i > 0$, contains squares of size $(2^i r) \times (2^i r)$ covering the whole plane, and such that every two intersecting squares have either a border side or a corner point in common. Observe also that each square in $\mathcal{G}_{2^i r}$ is a union of $2^{2(i-j)}$ different squares from grid $\mathcal{G}_{2^j r}$, for any $0 \leq j < i$.

Coloring cells in the grids. For any integer $i \geq 0$, we assign b^2 colors to the cells (squares) in grid $\mathcal{G}_{2^i r}$ as follows. We start with coloring cells in an arbitrary row in $\mathcal{G}_{2^i r}$ cyclically, using colors from 0 to $b^2 - 1$. Then, for any positive integer j, we color rows in $\mathcal{G}_{2^i r}$ as follows

- jth row above the first colored row in $\mathcal{G}_{2^i r}$: by shifting the colors right by $j \cdot b$ modulo b^2 (i.e., adding $j \cdot b$ modulo b^2 with respect to the color of the cell in the same column located in the first colored row), and
- jth row below the first colored row by shifting the colors left by $j \cdot b$ modulo b^2 (i.e., by subtracting $j \cdot b$ modulo b^2 with respect to the color of the corresponding cell in the first colored row).

The following fact holds by examining the grid coloring procedure.

Fact 1. *Any two cells of grid $\mathcal{G}_{2^i r}$, for any integer $i \geq 0$, colored by the same color have at least $b-1$ rows or at least $b-1$ columns of grid $\mathcal{G}_{2^i r}$ between them.*

Assigning channels in σ^*. For each node u, we define its channel c_u^* in σ^* as a triple (c_u, ℓ_u, q_u), where c_u is the number of the channel of u assigned by σ, while ℓ_u is an integer in $[b^2]$ and q_u is an integer in $[b]$ defined in the following way. Let $i \geq 0$ be the integer satisfying $2^i R_{min} \leq R_u < 2^{i+1} R_{min}$. We define ℓ_u to be the color of the cell in $\mathcal{G}_{2^i r}$ containing node u, and q_u is equal to $i \mod b$.

Re-scaling (optional). It is easy to see that the SINR ratios for the original distances and noise N are the same as the corresponding ones for the scaled values, for any transmission schedule. Therefore we keep the power and channel assignment computed in the scaled scenario unchanged.

4.2 Analysis of σ^*

We prove that if σ is a separable CP-assignment admissible for the radio model then the constructed σ^* is a CP-assignment admissible for the SINR model. We

also argue that the number of channels used by σ^* is bigger than the one used by σ by at most b^3 factor.

The main result about our transformation, proved in Section 4.3, is as follows.

Theorem 1. *If σ is a separable CP-assignment for a given placement of n nodes, admissible for the GRN model and using κ channels, then σ^* is a CP-assignment for this set of nodes which is admissible for the SINR model and it uses at most $\kappa \cdot b^3$ channels. Moreover, σ^* can be computed from σ in polynomial time.*

Applying Theorem 1 to the separable CP-assignment σ admissible for the radio model with at most $18 \log n + 1$ channels, as defined and analyzed in Section 3, we get the following result (recall that b is a constant and the considered assignment from Section 3 can be computed in polynomial time).

Corollary 1. *For any given set of n points on the plane there exists a CP-assignment admissible for the SINR model using $O(\log n)$ channels, and it can be computed in polynomial time.*

4.3 Proof of Theorem 1

Consider a placement of n nodes, satisfying $d_{\max} \leq 1$; otherwise it is easy to see that scaling and re-scaling procedures justify that the assignment computed in the scaled scenario gives the same connectivity graph as if we keep the same assignment in the original node placement. Also note that all computational steps made in the construction of σ^* are clearly polynomial (recall that α, β are constant parameters of the model).

Let σ be as assumed in the statement of the theorem. We may assume that the channels assigned by σ are from $[\kappa]$. Observe that the upper bound $\kappa \cdot b^3$ on the number of channels used in σ^* follows directly from the fact that there are κ channels used by σ, the range b^2 of the second coordinate and the range b of the third coordinate. In the remainder we argue that σ^* is admissible for the SINR model. We prove it by showing that each link realized by σ in the radio model is realized by σ^* in the SINR model, and thus the set of realized links forms a strongly directed spanning sub-graph.

Consider a link (v, w) realized by σ in the radio network model. To simplify the notation, denote the channel c_v assigned by σ to node v by c. First note that, according to the channel assignment in σ^*, only nodes with channel c allocated by σ may have the same channel assigned by σ^* as v does. We denote by T_c the set of nodes different from v, w that are assigned channel c by σ. Further, we partition set T_c into subsets $T_c(i)$, where $u \in T_c(i)$ if $u \in T_c$ and $2^i R_{min} \leq R_u < 2^{i+1} R_{min}$, for a non-negative integer i.

We first estimate the nominator of the SINR ratio at node w.

Lemma 1. *In the SINR model,*

$$\frac{P_v^*}{d(v,w)^\alpha} = \gamma \cdot \frac{R_v^{(2+\alpha)/2}}{d(v,w)^\alpha} \geq \gamma \cdot d(v,w)^{(2-\alpha)/2} \ .$$

Proof. First note that $R_v \geq d(v,w)$, by the facts that R_v is the transmission range assigned to v by σ and link (v,w) is realized by σ in the radio model. It follows that

$$\frac{P_v^*}{d(v,w)^\alpha} = \gamma \cdot \frac{R_v^{(2+\alpha)/2}}{d(v,w)^\alpha} \geq \gamma \cdot d(v,w)^{(2-\alpha)/2} \ ,$$

in the SINR model. □

We denote by c^* the channel assigned to node v by σ^*, that is, $c^* = (c_v, \ell_v, q_v) = (c, \ell, q)$, for some $\ell \in [b^2]$ and $q \in [b]$ (we drop sub-index v for simplicity, as v is fixed in the remainder of this proof).

It remains to estimate from above the denominator of the SINR ratio at node w, mainly the ratios $\frac{P_u^*}{d(u,w)^\alpha}$ for nodes $u \neq v$ with channel c^* assigned by σ^*. We denote the set of these nodes by $\mathcal{T}$. Note that $\mathcal{T} \subseteq \bigcup_{i \geq 0, i = q \mod b} T_c(i)$, by the fact that $c^* = (c, \ell, q)$. Observe also that links (u,w), for $u \in \mathcal{T}$, are not realized by σ in the radio model, since nodes in $\mathcal{T}$ are assigned the same channel c in σ as node v and we assumed that link (v,w) is realized by σ in the radio model.

For the purpose of the analysis we introduce a numbering of cells in $\mathcal{G}_r$ that is centered in the cell containing node w. More precisely, assume that the cell in $\mathcal{G}_r$ containing node w has cell-coordinates $(0,0)$, and the remaining cells are assigned pairs of integers accordingly to the directions of XY axis on the plane, i.e., cells in a jth row above (below) cell $(0,0)$ have the second coordinate j (resp., $-j$), and cells in a jth column on the right (left) side of cell $(0,0)$ have the first coordinate j (resp., $-j$), for any positive integer j.

The main tool in estimating the value of the interference at node w is a specific potential function associated with cells in grid $\mathcal{G}_r$. It "distributes" the value of the interference caused by a few points in the plane into wider square regions, in an almost uniform way, which later allows to estimate the total value of the interference by simple summing up the potential function over square regions. In order to define the potential function, we need to introduce an additional notation. Consider a node $u \in \mathcal{T}$. It follows that u has channel c^* allocated by σ^* and thus $u \in T_c(i)$, for some non-negative integer $i = q \mod b$. Let (x_u, y_u) be the coordinates of the cell in $\mathcal{G}_r$ containing node u. Recall that the location of node w was assumed to be in cell $(0,0)$ of the grid. Let C_u denote the cell in $\mathcal{G}_{2^{i-1}r}$ containing node u. We define S_u as a collection of cells (x,y) from grid $\mathcal{G}_r$ contained in C_u.

Lemma 2. *For any two different nodes $u, u' \in \mathcal{T}$, the sets S_u and $S_{u'}$ are disjoint.*

Proof. It is enough to show that $C_u = \bigcup S_u$ is contained in the ball of radius $R_u/2$ centered in u, for any node $u \in \mathcal{T}$.[2] Having this property proved, we could then apply separability of CP-assignment σ, saying that the distance between any $u \neq u'$ is at least the maximum of $R_u, R_{u'}$, which in turn is at least $R_u/2 + R_{u'}/2$.

[2] Notation $\bigcup S_u$, for a family of sets S_u, denotes the union of the sets in S_u; in our case, it is a union of squares on the plane.

This yields that the two balls—one centered in u and with radius $R_u/2$ and the other one centered in u' and with radius $R_{u'}/2$—are disjoint, and so the sets S_u and $S_{u'}$.

Consider a node $u \in \mathcal{T}$; we have $u \in T_c(i)$, for some non-negative integer $i = q \bmod b$. Recall that, by the definition of $T_c(i)$, we have $2^i R_{min} \leq R_u < 2^{i+1} R_{min}$. By the definition of S_u, the distance between any point in $C_u = \bigcup S_u$ and node u is smaller than

$$\sqrt{2} \cdot 2^{i-1} r = \sqrt{2} \cdot 2^{i-1} \cdot R_{min}/\sqrt{2} = 2^{i-1} R_{min} \leq R_u/2 \ ,$$

where in the equation above we used the definition of $r = R_{min}/\sqrt{2}$ and the property $2^i R_{min} \leq R_u$ of node $u \in T_c(i)$. This completes the proof of the lemma. □

For each cell (x, y) in S_u we associate its potential function $\psi_u(x, y)$ with respect to node $u \in \mathcal{T}$:

$$\psi_u(x, y) = \frac{a\gamma \cdot (R_u)^{(\alpha-2)/2}}{(r \cdot \max\{|x|, |y|\})^\alpha} \ ,$$

where $a = (16r)^\alpha$. The following fact links the potential function with the strength of the signal transmitted by $u \in \mathcal{T}$ and received by w in the SINR model.

Lemma 3. *Potential functions $\psi_u(x, y)$ of cell (x, y) in grid $\mathcal{G}_r$ satisfy the following inequality, for any $u \in \mathcal{T}$:*

$$\sum_{(x,y) \in S_u} \psi_u(x, y) \geq \frac{P_u^*}{d(u, w)^\alpha} \ .$$

Using Lemma 2, we can define a *general potential function* $\psi(x, y)$ of cell (x, y) in $\mathcal{G}_r$ as follows: since each cell (x, y) of $\mathcal{G}_r$ is in at most one set S_u, for $u \in \mathcal{T}$, we define $\psi(x, y)$ to be equal to $\psi_u(x, y)$ if there is $u \in \mathcal{T}$ such that $(x, y) \in S_u$, and $\psi(x, y) = 0$ otherwise.

Before making final estimation of $\sum_{u \in \mathcal{T}} \frac{P_u^*}{d(u,w)^\alpha}$ by using potential functions, as indicated by Lemma 3, we split this sum into two smaller sums. We denote by $\mathcal{T}^{(1)}$ the set of nodes $u \in \mathcal{T}$ such that $d(u, w) \geq b \cdot d(v, w)$. Note that if $R_u \geq b \cdot d(v, w)$, for $u \in \mathcal{T}$, then also $u \in \mathcal{T}^{(1)}$, but the reversed implication may not hold in general. Let $\mathcal{T}^{(2)} = \mathcal{T} \setminus \mathcal{T}^{(1)}$.

Lemma 4. *We have*

$$\sum_{u \in \mathcal{T}^{(1)}} \frac{P_u^*}{d(u, w)^\alpha} \leq \frac{P_v^*}{3\beta d(v, w)^\alpha} \ .$$

Lemma 5. *We have*

$$\sum_{u \in \mathcal{T}^{(2)}} \frac{P_u^*}{d(u, w)^\alpha} \leq \frac{P_v^*}{3\beta d(v, w)^\alpha} \ .$$

Observe also that, by the definition of γ and by the fact that link (v, w) is realized by σ in the radio model, we have

$$P_v^* = \gamma \cdot (R_v)^{(\alpha+2)/2} \geq 3N\beta \cdot (R_{max})^{(\alpha-2)/2} \cdot (R_v)^{(\alpha+2)/2} \geq 3N\beta \cdot (R_v)^{\alpha} \geq 3N\beta d(v,w)^{\alpha} \ ,$$

and consequently, the following fact holds:

Fact 2. *The inequality* $N \leq \frac{P_v^*}{3\beta d(v,w)^{\alpha}}$ *holds.*

By the inequalities from Lemmas 4 and 5 and from Fact 2, we finally get

$$\begin{aligned}
\frac{P_v^*}{d(v,w)^{\alpha}} &= \beta \cdot \left(3 \cdot \frac{P_v^*}{3\beta d(v,w)^{\alpha}} \right) \\
&\geq \beta \cdot \left(N + \sum_{u \in \mathcal{T}^{(1)}} \frac{P_u^*}{d(u,w)^{\alpha}} + \sum_{u \in \mathcal{T}^{(2)}} \frac{P_u^*}{d(u,w)^{\alpha}} \right) \\
&= \beta \cdot \left(N + \sum_{u \in \mathcal{T}} \frac{P_u^*}{d(u,w)^{\alpha}} \right) \ .
\end{aligned}$$

5 Concluding Remarks

We showed a generic (polynomial-time) transformation from separable CP-assignments admissible for the GRN model into CP-assignments admissible for the SINR model, preserving the number of used channels (asymptotically). For the purpose of analysis we introduced a potential function that may be a useful tool for analysis of transformations between GRN and SINR models for other communication problems. An interesting open question is whether there exists a dual transformation, i.e., a (polynomial-time) transformation from CP-assignments admissible for the SINR model into the ones admissible for the radio model, which does not substantially change the number of channels. Another perspective line of research is to consider dynamic and/or ad hoc scenarios in the context of the connectivity problem.

Our transformation could be also viewed as an algorithm solving the problem of scheduling communication along a given set of links (in our case, a set of links that forms a strongly connected spanning graph) with preliminary (radio) power and channel assignment satisfying separability condition. The requirement of being a separable radio assignment narrows the class of allowed inputs, and therefore some of negative results and lower bounds concerning general scheduling of *any* given set of links may not apply to our construction. For example, a linear lower bound on the number of channels proved in [9] for schedules that use so called "oblivious" power assignment (i.e., where the power assigned to the sender is a function of the distance between the sender and the receiver) cannot be applied to the class of inputs allowed by our transformation. More precisely, the lower bound in [9] was proved based on placements of nodes on the line with distances between consecutive points growing exponentially. It is easy to check

that such settings allow efficient radio assignments that form a line, and therefore some inputs for our transformation have $O(1)$ channels. The transformation assigns transmission powers in such a way than they are polynomially smaller that the corresponding values of the signal-decay factors (i.e., the distance to the power α). Hence, since the distances grow exponentially, nodes that are more than a few-hop distance away from the receiver contribute very little, in total, to the power of the interference at the receiver. This means that a constant number of channels is still enough to assure connectivity by our construction in the considered setting. This justifies that the lower bound for oblivious scheduling from [9] does not apply to our construction.

On the other hand, previously known solutions to the (general) scheduling problem cannot be directly applied to provide an upper bound to the connectivity problem for two reasons. First, they aim to find a solution only for a given set of links, while a solution to the connectivity problem needs to find a suitable set of links by itself. Second, solutions to the scheduling problem focus on comparing the computed assignment to the best possible one, while our solution establishes a global upper bound on the number of channels guarantying strong connectivity for any node placement.

In general, the proposed research direction of studying relations between different models of wireless communication can be applied to other models and communication problems. The GRN model appears to be simpler and more intuitive to deal with, and thus it may be more convenient for designing and analyzing wireless algorithms, while the SINR model seems to be closer to the real physical layer. Finding transformations between these two, and other related, models would help in better understanding of wireless communication and in transforming protocols and complexity results between different models.

Acknowledgments

The authors would like to thank the reviewers for pointing out an interesting relation between the connectivity and scheduling problems in the SINR model.

References

1. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A Lower Bound for Radio Broadcast. Journal of Computer and System Sciences 43, 290–298 (1991)
2. Apostol, T.M.: Introduction to Analytic Number Theory. Springer, New York (1995)
3. Avin, C., Emek, Y., Kantor, E., Lotker, Z., Peleg, D., Roditty, L.: Towards Algorithmically Usable SINR Models of Wireless Networks. In: Proc. 28th ACM Symposium on Principles of Distributed Computing (PODC), pp. 200–209 (2009)
4. Avin, C., Lotker, Z., Pasquale, F., Pignolet, Y.A.: A Note on Uniform Power Connectivity in the SINR Model. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 116–127. Springer, Heidelberg (2009)
5. Chlamtac, I., Kutten, S.: Tree-Based Broadcasting in Multihop Radio Networks. IEEE Transactions on Computers 36, 1209–1223 (1987)

6. Chlamtac, I., Weinstein, O.: The Wave Expansion Approach to Broadcasting in Multihop Radio Networks. IEEE Trans. on Communications 39, 426–433 (1991)
7. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective Families, Superimposed Codes, and Broadcasting on Unknown Radio Networks. In: Proc. 12th Annual Symposium on Discrete Algorithms (SODA), pp. 709–718 (2001)
8. Dessmark, A., Pelc, A.: Broadcasting in Geometric Radio Networks. J. Discrete Algorithms 5, 187–201 (2007)
9. Fanghanel, A., Kesselheim, T., Racke, H., Voecking, B.: Oblivious Interference Scheduling. In: Proc., 28th ACM Symposium on Principles of Distributed Computing (PODC), pp. 220–229 (2009)
10. Fussen, M., Wattenhofer, R., Zollinger, A.: Interference Arises at the Receiver. In: Proc. International Conference on Wireless Networks, Communications, and Mobile Computing, WIRELESSCOM (2005)
11. Gasieniec, L., Kowalski, D.R., Lingas, A., Wahlen, M.: Efficient Broadcasting in Known Geometric Radio Networks with Non-uniform Ranges. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 274–288. Springer, Heidelberg (2008)
12. Gasieniec, L., Peleg, D., Xin, Q.: Faster Communication in Known Topology Radio Networks. In: Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC), pp. 129–137 (2005)
13. Goussevskaia, O., Halldorsson, M., Wattenhofer, R., Welzl, E.: Capacity of Arbitrary Wireless Networks. In: Proc. 28th Ann. IEEE Conference on Computer Communications (INFOCOM), pp. 1872–1880 (2009)
14. Goussevskaia, O., Oswald, Y.A., Wattenhofer, R.: Complexity in Geometric SINR. In: Proc. 8th ACM Int. Symp. on Mobile Ad Hoc Networking and Computing (MobiHoc), pp. 100–109 (2007)
15. Gupta, P., Kumar, P.R.: The Capacity of Wireless Networks. IEEE Transactions on Information Theory 46, 388–404 (2000)
16. Halldorsson, M., Wattenhofer, R.: Wireless Communication is in APX. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 525–536. Springer, Heidelberg (2009)
17. Huson, M.L., Sen, A.: Broadcast Scheduling Algorithms for Radio Networks. In: Proc. IEEE Military Communications Conference (MILCOM), pp. 647–651 (1995)
18. Kesselman, A., Kowalski, D.R.: Fast Distributed Algorithm for Convergecast in Ad Hoc Geometric Radio Networks. J. Parallel and Distributed Comput. 66, 578–585 (2006)
19. Kowalski, D.R., Pelc, A.: Optimal Deterministic Broadcasting in Known Topology Radio Networks. Distributed Computing 19, 185–195 (2007)
20. Lebhar, E., Lotker, Z.: Unit Disk Graph and Physical Interference Model: Putting Pieces Together. In: Proc., 23rd IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS), pp. 1–8 (2009)
21. Moscibroda, T.: The Worst-case Capacity of Wireless Sensor Networks. In: Proc. 6th Int. Conf. on Information Processing in Sensor Networks (IPSN), pp. 1–10 (2007)
22. Moscibroda, T., Oswald, Y.A., Wattenhofer, R.: How Optimal are Wireless Scheduling Protocols? In: Proc. 26th Ann. IEEE Conference on Computer Communications (INFOCOM), pp. 1433–1441 (2007)
23. Moscibroda, T., Wattenhofer, R.: The complexity of Connectivity in Wireless Networks. In: Proc. 25th Ann. IEEE Conference on Computer Communications (INFOCOM), pp. 1–13 (2006)
24. Moscibroda, T., Wattenhofer, R., Zollinger, A.: Topology Control Meets SINR: the Scheduling Complexity of Arbitrary Topologies. In: Proc., 7th ACM Int. Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), pp. 310–321 (2006)

Trusted Computing for Fault-Prone Wireless Networks*

Seth Gilbert[1] and Dariusz R. Kowalski[2]

[1] National University of Singapore, Singapore
gilbert@comp.nus.edu.sg
[2] University of Liverpool, United Kingdom
D.Kowalski@liverpool.ac.uk

Abstract. We consider a fault-prone wireless network in which communication may be subject to wireless interference. There are many possible causes for such interference: other applications may be sharing the same bandwidth; malfunctioning devices may be creating spurious noise; or malicious devices may be actively jamming communication. In all such cases, communication may be rendered impossible.

In other areas of networking, the paradigm of "trusted computing" has proved an effective tool for reducing the power of unexpected attacks. In this paper, we ask the question: can some form of trusted computing enable devices to communicate reliably? In answering this question, we propose a simple "wireless trusted platform module" that limits the manner in which a process can access the airwaves by enabling and disabling the radio according to a pre-determined schedule. Unlike prior attempts to limit disruption via scheduling, the proposed "wireless trusted platform module" is general-purpose: it is independent of the application being executed and the topology of the network.

In the context of such a "wireless trusted platform module," we develop a communication protocol that will allow any subset of devices in a region to communicate, despite the presence of other disruptive (possibly malicious) devices: up to k processes can exchange information in the presence of t malicious attackers in $O(\max(t^3, k^2) \log^2 n)$ time. We also show a lower bound: when $t < k$, any such protocol requires $\Omega(\min(k^2, n) \log_k n)$ rounds; in general, at least $\Omega(\min(t^3, n^2))$ rounds are needed, when $k \geq 2$.

1 Introduction

Wireless networks are everywhere, enabling devices to communicate and exchange information without the need for physical infrastructure. Wireless networks rely on the open airwaves for communication, and the open airwaves are publicly accessible by anyone and everyone. This openness has advantages, allowing universal participation and creating a lower barrier to entry; it also has

* This publication was prepared while the first author was at EPFL, Switzerland. The work of the second author was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/G023018/1, EP/H018816/1].

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 359–373, 2010.

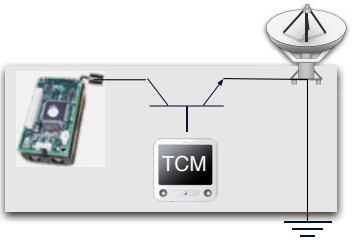


Fig. 1. Simple schematic of sensor device with a wireless "trusted computing module"(e.g., a wTPM) controlling power to the antenna

disadvantages: any user can join the network and cause disruption. Disruption may be caused intentionally, by malicious parties, or unintentionally, by other applications sharing the same bandwidth.

Trusted Computing. Recently, the paradigm of *Trusted Computing* has come to be seen as a powerful technique for reducing vulnerability to attack. (See, e.g., [29, 33].) The basic idea underlying *trusted computing* is that each computer (or networked device) will contain a tamper-proof component (often a special-purpose chip) known as a *trusted platform module* (TPM) that provides certain reliable guarantees. For example, the TPM may contain cryptographic authentication keys that securely identify the computer. The TPM may also contain a mechanism that protects data stored on a computer, or that may provide certain guarantees as to the software running on that computer. Elements of the trusted computing architecture are implemented today in Windows Vista, for example, in BitLocker Drive Encryption.

In this paper, we examine the application of trusted computing techniques to wireless networking, in particular to the problem of interference and disruption (either benign or malicious) on the wireless airwaves. Imagine that every wireless device has a "wireless" TPM (wTPM) that controls access to the radio[1]. (See Figure 1 for a simplified schematic representation.) When the radio is enabled by the wTPM, the software running on the wireless device can send and receive messages; conversely, when the radio is disabled by the wTPM, the software running on the wireless device cannot access the radio. Thus, even when a malicious attacker hacks or takes control of a wireless devices, (s)he can only create interference when the radio is enabled[2].

An important design criterion for a wTPM is that it be simple, and that it be as computation-agnostic as possible. The wTPM should not be aware of

[1] Note that use of the radio frequencies is already heavily regulated in most countries, and hence it might be feasible to require all legal devices to be equipped with such a wTPM; of course the "trusted computing" approach will be ineffective for a concerted attacker with access to illegal hardware.

[2] While "trusted computing" is sometimes criticized for its privacy implications, these problems are less severe in the wireless case where the wTPM only affects when the radio is enabled, while revealing no personal information.

the computation running on the wireless device, nor should it monitor the communication sent and received over the radio. Ideally, it should simply connect and disconnect the radio, irregardless of what the device is doing or whether the device is sending or receiving a message. The fundamental open question is whether it is possible to design such a simple, computation-agnostic wTPM that will still allow wireless devices to perform the communication and computation that they desire, without sacrificing efficiency.

Overview of Results. In this paper, we make some progress toward answering these questions. We focus on the basic problem of reliably exchanging information: there are at most k wireless devices—from some larger universe of n devices—that want to exchange information with each other. (We assume that k is fixed; however, we discuss in Sections 5.3 how to adapt to varying numbers of participants.) At the same time, there are at most t malicious devices that disrupt communication. These devices may broadcast corrupt messages, or they may "jam" the airwaves—when their radios are enabled by the wTPM—preventing any information form being exchanged.

Each device contains two pieces of software: (1) the wTPM, which is tamperproof, and (2) the communication protocol, which may be corrupt on malicious devices. *(1) wTPM*: The wTPM consists of a fixed binary sequence indicating whether the radio is enabled or disabled at any given time. When the radio is enabled, the device can communicate; when the radio is disabled, it can neither send nor receive. The behavior or the wTPM is fixed in advance, and is not affected by anything that occurs during an execution. *(2) Communication protocol*: The communication protocol determines whether the device broadcasts or receives in any given round, if the radio is enabled. We focus on *oblivious* protocols where the broadcast/receive schedule is also fixed in advance.

Our main result consists of the wTPM design, along with an efficient communication protocol that is compatible with the wTPM. Our protocol runs in $\Theta(\max(t^3, k^2) \log^2 n)$ rounds; surprisingly, this is almost as efficient as the best (oblivious) protocols for exchanging information, even when all the devices are honest: every such protocol requires $\Omega(\min(k^2, n) \log_k n)$ rounds [6, 11]. We provide a complementary lower bound, showing that exchanging information requires at least $\Omega(\min(t^3, n^2))$ rounds, for $k \geq 2$. (Note: there is a trivial $O(n^2)$ solution that selectively enables each pair of processes.) Together, these lower bounds indicate that our proposed protocol is near optimal in most cases.

Both the wTPM and the protocol are generated by a random process which is designed to activate only (approximately) k/t radios in each round. Even though communication among the honest devices is (effectively) limited to one-to-one communication, we still exchange information nearly as efficiently as protocols that rely on one-to-many communication, e.g., have each sender transmit his information while all the other processes listen. Of note, the resulting protocols are relatively simple to implement when provided with a good source of (pseudo)-random bits (or data structures for generating such bits, e.g., extractors).

Our approach to security also has a secondary benefit: it can significantly reduce the power usage of wireless protocols. Powering the radio is one of the

most energy-consuming operations for a small wireless device. If the wTPM disables the radio sufficiently often, then it forces every protocol running on the device to be much more energy efficient than might otherwise be the case. When $t = \Theta(k)$, the wTPM enforces a high-level of energy efficiency, enabling only $O(1)$ devices in each round. By contrast, prior protocols for information exchange activate $\Theta(k)$ processes per round.

Finally, in order to enable more general applications, we also briefly consider the *continuous* version of information exchange, where there are an unknown number of processes that occasionally have information to distribute. A natural generalization of our protocol ensures that every message injected in some round r will be delivered by round $r + O(\max(t^3, \ell^2) \log^3 n)$, as long as there are at most $\ell \leq n$ active messages during that interval.

Other Approaches to Tolerating Malicious Devices in a Radio Network. The idea of enforcing a fixed broadcast schedule for a wireless radio, in order to avoid malicious interference, has been previously proposed on several occasions. Koo [20], in one of the first papers studying the problem of reliable broadcast in a wireless network subject to Byzantine failures, suggested that devices be forced to follow a "round-robin" schedule, preventing malicious devices from broadcasting out of turn. Later papers (e.g., [4, 5]) followed this approach as well. In general, such an approach either assumes that only one device is enabled at a time—leading to $\Omega(n)$ or $\Omega(n^2)$ running times—or it relies on some geographic property to determine whether a device can broadcast, for example, enabling devices in specific regions to broadcast in a given round. By contrast, in this paper, we attempt to develop a *generic* wTPM that is computation-agnostic, geographically ignorant, and yet still achieves efficient performance.

An alternate approach for dealing with malicious disruption is to posit some limit on the *amount* of disruption (see, e.g., [1, 16, 21]), or on the *rate* at which the devices can cause disruption [2]. Such limits might arise from practical considerations, e.g., the size of the battery on a malicious device, or from hardware constraints, e.g., a device might not be allowed to broadcast at above some specified rate. The latter approach, in particular, has significant promise for a wTPM solution, as a wTPM might enforce a bounded rate of broadcast.

A third approach for coping with malicious disruption is to leverage the availability of more than one communication channel: while malicious devices may disrupt some subset of the available channels, reliable communication can proceed on the other channels. This has proved a popular approach, as it requires minimal limitations on the power or scope of the malicious devices. (See, for example, [12–15, 17, 26, 28, 31, 32].) One open question is whether such multi-channel solutions could be even more efficient if a wTPM were available.

Other Related Work. The problem of resolving contention among a set of (honest, fault-free) devices on a multiple-access channel has been extensively studied (see, e.g., [3, 19, 22, 34], among many others). Wireless networks with crash failures (but not Byzantine failures) have also been studied extensively (e.g., [8–10, 24]). In essence, the challenge in this paper is to solve the problem of

contention among honest processes, while simultaneously preventing the malicious processes from jamming.

Recently, there has been much interest in other models of interference, such as the SINR model [18, 27], and the *dual-graph model* [25]. These models capture interference in a somewhat more sophisticated manner, and hence it is an interesting open question how to cope with malicious interference in such models.

2 A Model for Wireless Trusted Computing

Model. Let Π be a set of n processes. Each process knows n and the set Π. Each process is either *active* or *passive*. An active process can send/receive messages and perform computations; a passive process cannot act in any way. At most k honest processes are activated. At most t dishonest (or *Byzantine*) processes may also be activated; such processes may act in an arbitrarily malicious fashion.

Processes communicate with a radio over a collision-prone wireless channel. In each round, each process (whether honest or dishonest) can either broadcast or listen. When exactly one process broadcasts, every other process receives the message; when more than one process broadcasts, no process receives anything.

Trusted Computing. Each device is equipped with a tamper-proof *wireless trusted-platform module (wTPM)*. The wTPM at each process is initialized with a binary string that indicates, for each round, whether the radio is enabled or disabled. When the radio is disabled, the process can neither send nor receive. The Byzantine devices cannot corrupt the wTPM, meaning that they cannot broadcast when the radio is disabled by the wTPM.

We define an *algorithm* $\langle T, B\rangle$ to be two binary (n, m)-matrices. We refer to matrix T as the *radio-enable* matrix and matrix B as the *broadcast-listen* matrix. Rows of each matrix correspond to processes, and columns corresponds to rounds. That is, row p of matrix T is the initialization string for the wTPM at process p: the radio at process p is enabled in round r if and only if $T[p, r] = 1$. Similarly, row p of matrix B indicates whether process p broadcasts or listens in each round: process p broadcasts in round r if $B[p, r] = 1$; otherwise it listens. We assume, for simplicity, that whenever a process is enabled and scheduled to broadcast, it transmits all available information. (There is no required relation between T and B.)

By definition, algorithms are *oblivious*: the behavior of each process is fixed; they do not adapt to adversarial behavior. Oblivious protocols have several advantages: they are often more robust, as they do not depend on accurately observing ongoing events. In the case of a wTPM, an oblivious wTPM would appear more plausible, as it can be constructed in a generic protocol-independent manner (as compared to attempting to adapt to circumstances).

Exchanging Information. We consider the problem of (k, t)-*information exchange*. Define $P \subseteq \Pi$ to be the set of at most k active honest processes. (Note that activations are local; a process knows only whether it is in set P or not.) Each process in P is initialized with a rumor. At the end of the execution, every

active, honest process should transmit its rumor to every other active, honest process, as long as there are at most t active dishonest processes.[3]

The primary metric is time complexity, i.e., the number of rounds that the protocol executes. In Section 5.3 we consider a continuous variant where we count from a rumor's injection until the rumor is received by all other honest processes. Another complexity measure of interest is energy consumption, defined as the sum, over all rounds, of the number of radio-enabled processes.

3 Lower Bound

Theorem 1. *For the problem of (k,t)-information exchange:(i) if $k \geq t$, then $\Omega(\min(k^2, n)\log_k n)$ rounds are required; (ii) if $k \geq 2$, then $\Omega(\min(t^3, n^2))$ rounds are required.*

Proof. When $k \geq t$, the lower bound of $\Omega(\min(k^2, n)\log_k n)$ follows from bounds on superimposed codes [6, 11], which holds even when all processes are honest.

Now assume $k \geq 2$. We show that there are two honest processes that fail to exchange rumors in the first $t^2(t-2)/32$ rounds. It is sufficient to consider the case when $t^2(t-2)/32 < n(n-1)/4$.

For $0 \leq i \leq n$, define R_i to be the set of rounds such that $A_r = \{p : T[p,r] = 1\}$ is of size i. We omit rounds in set $R_0 \cup R_1$ from the analysis, as at most one radio-enabled process cannot send a message to any other process. We focus on sets R_2 and $R_{\geq 3} = \bigcup_{i \geq 3} R_i$. Let S_2 be the set of all pairs that are radio-enabled in rounds in R_2, i.e., $S_2 = \{\{p,q\} : \exists_{r \in R_2}\ T[p,r] = T[q,r] = 1\}$. We have $|S_2| \leq |R_2| \leq t^2(t-2)/32 < n(n-1)/4$.

Let F_2 be a set of $t/2$ processes such that the number of pairs of elements from F_2 that are in S_2 is smaller than $\binom{t/2}{2}/2$. Such a set exists by a probabilistic argument: the expected number of pairs from S_2 included in the random set of $t/2$ processes is smaller than:

$$|S_2| \cdot \frac{\binom{n-2}{t/2-2}}{\binom{n}{t/2}} = |S_2| \cdot \frac{(t/2-1)(t/2)}{(n-1)n} < n(n-1)/4 \cdot \frac{(t/2-1)(t/2)}{(n-1)n} = \frac{1}{2} \cdot \binom{t/2}{2},$$

and therefore a set with this property exists.

Consider processes in F_2 and rounds in $R_{\geq 3}$. Let R be the subset of $R_{\geq 3}$ containing rounds $r \in R_{\geq 3}$ such that $|\{p \in F_2 : T[p,r] = 1\}| = 2$; let S be the set $\{\{p,q\} : p,q \in F_2\} \setminus S_2$. Since each round in R is associated with at most one pair in S, and there are at least $\binom{t/2}{2} - \binom{t/2}{2}/2 = \binom{t/2}{2}/2$ pairs in S, there is a pair $\{p^*, q^*\}$ in S associated with at most $\frac{|R|}{|S|} \leq \frac{t^2(t-2)/32}{\frac{1}{2}\cdot\binom{t/2}{2}} \leq t/2$ rounds in $R_{\geq 3}$.

Let $R^*_{\geq 3} = \{r \in R_{\geq 3} : T[p^*,r] = T[q^*,r] = 1\}$. By the choice of p^*, q^*, we have $|R^*_{\geq 3}| \leq t/2$. Since $R^*_{\geq 3} \subseteq R_{\geq 3}$, for every $r \in R^*_{\geq 3}$ there is a process $p(r)$ different from p^*, q^* such that $T[p(r),r] = 1$. Let F_3 be the set of processes

[3] Malicious processes may create their own rumors, which cannot be distinguished from honest rumors; it is unavoidable that processes may deliver such rumors.

$\{p(r) : r \in R^*_{\geq 3}\}$. An estimate $|F_3| \leq |R^*_{\geq 3}| \leq t/2$ holds. We define F as $(F_2 \setminus \{p^*, q^*\}) \cup F_3$. It follows that $|F| \leq |F_2| - 2 + |F_3| < t$.

Let $\{p^*, q^*\}$ be the set of honest processes, and F be the set of Byzantine processes. The adversary's strategy is as follows: whenever a Byzantine process is radio-enabled according to T, it transmits. It is easy to check that in each round where processes p^*, q^* are radio-enabled by T, there is also another process in F which is radio-enabled by T, and thus it interrupts any attempted transmission between p^* and q^*. Indeed, they cannot both be active in round $r \in R_2$, since $\{p^*, q^*\}$ in set S, and set S does not contain—by definition—any pair in set S_2, i.e., any pair that is active alone in some round in R_2. Therefore no communication can occur between p^* and q^* during rounds in R_2. (Recall that this is also impossible in rounds in R_0 and R_1, by definition.) Consider a round $r \in R_{\geq 3}$. If p, q are both active in round r, then, by definition of $R_{\geq 3}$, there must be at least one more process active in this round. Hence, round r satisfies the condition in the definition of set F_3, which means that at least one process $p \in F_3$ is such that $T[p, r] = T[p^*, r] = T[q^*, r] = 1$, and thus p jams the communication between p^*, q^* in round r. Therefore, rumors between p^*, q^* are not exchanged. Finally, note that set F of Byzantine processes is of size $|F_2| - 2 + |F_3| < t$ and there are only two honest processes p^*, q^*. □

4 Implementing Information Exchange

We now present an algorithm that performs (k, t)-*information exchange* in time $\Theta(\max(t^3, k^2) \log^2 n)$. Each process knows that there are at most k honest processes active, and at most t Byzantine processes active. We define $\langle T, B \rangle$ (i.e., the algorithm) using a random process, and argue that the resulting algorithm achieves the desired results with high probability. This both shows that there exists an efficient deterministic solution to the problem of (k, t)-*information-exchange*, via the probabilistic method, and shows how to find it efficiently.

We construct the algorithm out of sub-pieces. As we do not know how many honest processes are active, we define algorithm $A(\ell, \ell/2)$ which assumes that there are more than $\ell/2$ but at most ℓ honest processes active, and which runs in time $\Theta(\max(t^3/\ell, \ell^2) \log^2 n)$ rounds. The final protocol consists of concatenating the algorithms $A(\cdot, \cdot)$ for exponentially decreasing ranges, i.e., $A_k = A(k, k/2)$ & $A(k/2, k/4)$ & ... & $A(2, 1)$, where & represents concatenation. Summing the costs as ℓ decreases, the final running time is $\Theta(\max(t^3, k^2) \log^2 n)$.

4.1 Defining Algorithm $A(\ell, \ell/2)$

We now define $A(\ell, \ell/2) = \langle T, B \rangle$, for any $2 \leq \ell \leq k$, where ℓ is a power of 2. We divide $A(\ell, \ell/2)$ into three "sub-algorithms", $\langle T_1, B_1 \rangle, \langle T_2, B_2 \rangle, \langle T_3, B_3 \rangle$, which when concatenated, form $A(\ell, \ell/2)$. Let $m = c \cdot \max(t^3/\ell, \ell^2) \log n$, for a sufficiently large constant c, to be derived in the analysis. (The function of each of these stages is described in more detail in Section 4.2.)

- *Stage 1:* $\langle T_1, B_1 \rangle$ We define T_1 to be a binary $(n \times m)$-matrix where, for every p, r, each bit $T_1[p,r] = 1$ with probability $\min(1/t, 1/\ell)$; otherwise $T_1[p,r] = 0$. We define B_1 to be a binary $(n \times m)$-matrix where, for every p, r, each bit $B_1[p,r] = 1$ with probability $1/2$; otherwise $B_1[p,r] = 0$.
- *Stage 2:* $\langle T_2, B_2 \rangle$ Define the $(n \times 2m)$-matrix T' as follows, for all p : For each odd column $r = 1, 3, 5, \ldots$, define $T'[p,r] = 1$ with probability $\min(1/t, 1/\ell)$; otherwise $T'[p,r] = 0$. For each even column $r = 2, 4, 6, \ldots$, define $T'[p,r]$ to be identical to the preceding column, i.e., $T'[p,r] = T'[p,r-1]$. Define T_2 as $2 \log n$ repetitions of T'; T_2 is a $(n \times (4m \log n))$-matrix.
  Define B' as follows, for all p: For each odd column $r = 1, 3, 5, \ldots$, we define $B'[p,r] = 1$ with probability $1/2$; otherwise $B'[p,r] = 0$. For each even column $r = 2, 4, 6, \ldots$, we define $B'[p,r]$ to be the inverse of the preceding column, i.e., we define $B'[p,r] = (1 - B'[p,r-1])$. Define B_2 as $2 \log n$ repetitions of B'. Note that B_2 is a $(n \times (4m \log n))$-matrix.
- *Stage 3:* $\langle T_3, B_3 \rangle$ We define T_3 to be identical to T_1, i.e., $T_3 = T_1$. We define B_3 to be the inverse of B_1, i.e., for all p, r: $B_3[p,r] = (1 - B_1[p,r])$.

Thus, $A(\ell, \ell/2)$ is defined by the matrices T_1 & T_2 & T_3 and B_1 & B_2 & B_3. It follows that the length of algorithm $A(\ell, \ell/2)$ is $O(\max(t^3/\ell, \ell^2) \log^2 n)$.

4.2 Overview of the Analysis

We now analyze the protocol and show that it is correct and efficient. We consider $A(\ell, \ell/2)$, where $2 \leq \ell \leq k$. As we have already bounded the running time, we focus on showing that every honest process succeeds in transmitting its rumor to every other honest process. Fix ℓ such that there are more than $\ell/2$ and at most ℓ honest, active processes. Recall that P is the set of honest, active processes. We examine each of the three "sub-algorithms" separately.

- *Stage 1:* guarantees each rumor is delivered to $> (|P| - \ell/8)$ honest processes.

When this stage completes, each rumor is known to a large number of honest processes. However, there may be no one honest process that knows all the rumors. While it is relatively cheap to distribute each rumor to a large fraction of the participants, it is more expensive to deliver each rumor to *every* other active participant. In the first stage, rumors are delivered directly, in a pairwise fashion: each participant directly sends its rumor to a large fraction of the other participants. In order to deliver every rumor directly in a pairwise fashion to *every* process would require approximately $\Theta(k^2 t)$ rounds. The second stage avoids this by exchanging rumors indirectly.

- *Stage 2:* guarantees that there is a subset $P^* \subseteq P$ of size $\ell/8$ where every process in P^* has received all the rumors.

The second stage relies on a more careful examination of the communication graph defined by the protocol. Unlike in Stage 1, we do not rely on direct edges between pairs of processes, but instead expect rumors to be passed indirectly

over multiple "hops" in the induced communication graph. We show that the communication graph, when appropriately defined, has good *expansion* (see Definition 1), which immediately implies that the communication graph has a large component with small diameter (see Corollary 1). We can then conclude that every process in the large component learns every rumor.

– *Stage 3:* guarantees that each honest process receives at least one message from a process in P^*.

Processes in P^* cooperate to ensure that every process in P is notified of all the rumors. Notably, it turns out that Stage 3 is the symmetric opposite of Stage 1: whereas Stage 1 involved disseminating rumors, Stage 3 involves collecting them. We now proceed to analyze the three parts in more detail.

4.3 Stage 1: Spreading

The goal of the first stage, intuitively, is to distribute each rumor to more than $|P| - \ell/8$ honest participants. We show that the sub-protocol $\langle T_1, B_1 \rangle$ guarantees the following property: For every $P^* \subseteq P$, where $|P^*| = \ell/8$, and for every rumor ρ, there exists some process $q \in P^*$ such that q receives ρ during Part 1 of the protocol. This implies that we can choose any subset of P of size $\ell/8$ and be sure that every rumor is known by at least one member of that subset.

For the purpose of the next lemma, fix some set P of size bigger than $\ell/2$ and at most ℓ, some subset $P^* \subseteq P$ of size $\ell/8$, and some process $p \in P \setminus P^*$. (When $p \in P^*$, the property follows trivially.) Also, fix some set F of at most t Byzantine processes. We calculate the probability that the rumor from process p reaches some process in P^* without being disrupted by a process in F:

Lemma 1. *For given sets P, P^*, F and process $p \in P \setminus P^*$: there is some round r and some process $q \in P^*$ such that p successfully transmits its rumor to q in round r (i.e., p is the only process radio-enabled that transmits and q is the only process radio-enabled that listens in round r) with probability at least $1 - e^{-(c/128)\cdot\max(t,\ell)\log n}$.*

Proof (sketch). For any given round r, the probability that p is radio-enabled and set to broadcast, while exactly one process in P^* is radio-enabled and set to listen, while every other process in P and F is radio-disabled is at least $\min(\ell/(16t), 1/16)$. Thus, the probability that p fails to broadcast to q in all $c \cdot \max(t^3/\ell, \ell^2) \log n$ rounds is as desired, with high probability. □

By counting the number of possible configurations of subsets, we conclude, by a union bound, that the desired property is achieved by the end of the first stage:

Lemma 2. *The following event holds w.h.p., for sufficiently large constant c: For every set P of active processes where $\ell/2 < |P| \leq \ell$, for every subset $P^* \subseteq P$ of size $\ell/8$, for every set F of at most t Byzantine processes, every rumor in P is received by some process in P^* by the end of sub-algorithm $\langle T_1, B_1 \rangle$.* □

4.4 Stage 2: Exchanging

We now show that there is some subset of honest processes of size $\ell/8$ where every process in the set has received every rumor by the end of the second stage.

Recall that $\langle T_2, B_2 \rangle$ consists of $2 \log n$ repetitions of two matrices T' and B', respectively. Given a set of honest processes P and a set of Byzantine processes F, we define an undirected graph $G(P, F, T', B')$ based on T' and B'. Each vertex in G represents a process, i.e., there are n vertices. For each odd column $r = 1, 3, 5, \ldots$ we add an edge (p, q) to graph G if the following hold: (1) For every process $p' \in F$, $T'[p', r] = 0$, i.e., every Byzantine process is radio-disabled. (2) For every process $q' \in P \setminus \{p, q\}$, $T'[q', r] = 0$, i.e., every other process is radio-disabled. (3) For processes p and q, $T'[p, r] = T'[q, r] = 1$, i.e., processes p and q are radio-enabled. (4) For process p, $B'[p, r] = 1$; for process q, $B'[q, r] = 0$.

This implies that process p succeeds in sending a message to process q in the round based on column r. Since column $r+1$ is defined in terms of column r, we conclude that q succeeds in sending a message to process p in the round based on column $r + 1$. Thus, we consider the graph G to be undirected.

We argue that for all sets P and F, the graph $G(P, F, T', B')$ has a large subgraph with small diameter. We show this by examining the *expansion* of G. Following the definition from [30], we say that a graph G is an α-expander if it follows the following property:

Definition 1. *A graph $G = (V, E)$ is an α-expander if for every pair of subsets $W_1 \subseteq V$ and $W_2 \subseteq V$, where $|W_1| \geq \alpha$ and $|W_2| \geq \alpha$, there is some $p \in W_1$ and some $q \in W_2$ such that $(p, q) \in E$.*

We will argue that, with high probability, for every set P and set F, graph $G(P, F, T', B')$ is an $\ell/8$-expander:

Lemma 3. *With high probability, for sufficiently large c, for every P and F, graph $G(P, F, T', B')$ is an $\ell/8$-expander.*

Proof. Fix a set P of size $\ell/2 < |P| \leq \ell$ and a set F of size at most t. (We may assume, without loss of generality, that F is of size exactly t, as otherwise the adversary could add "silent" Byzantine processes without otherwise changing the execution.) We calculate the probability that $G(P, F, T', B')$ is a $\ell/8$-expander (after which we take a union bound over all possible sets P and F).

Fix arbitrary sets W_1 and W_2 of size at least $\ell/8$. We calculate the probability that there is some edge between W_1 and W_2 in $G(P, F, T', B')$. (We then take a union bound over all possible sets W_1 and W_2.) Specifically, for a given column of T' and B': (1) Every process in F is radio-disabled: with probability $\geq (1-\min(1/t, 1/\ell))^{|F|} \geq (1-1/t)^t \geq 1/4$. (2) Exactly one process in W_1 is radio-enabled: with probability at least $(\ell/8) \cdot \min(1/t, 1/\ell) \cdot (1 - \min(1/t, 1/\ell))^{\ell/8-1} \geq (1/32) \cdot \min(\ell/t, 1)$. (3) Exactly one process in W_2 is radio-enabled: with probability at least $(\ell/8) \cdot \min(1/t, 1/\ell) \cdot (1 - \min(1/t, 1/\ell))^{\ell/8-1} \geq (1/32) \cdot \min(\ell/t, 1)$. (4) The conditional event, under the assumption that one radio-enabled element in W_1 is chosen and one radio-enabled element in W_2 is chosen, that either the radio-enabled process in W_1 is set to broadcast and the radio-enabled process in

W_2 is set to receive, or the radio-enabled process in W_1 is set to receive and the radio-enabled process in W_2 is set to broadcast: with probability at least $1/2$.

Thus, for a given column, there is an edge between W_1 and W_2 with probability at least $\frac{1}{8} \cdot \left(\frac{\min(\ell/t,1)}{32}\right)^2$. Thus over $c \cdot \max(t^3/\ell, \ell^2) \log n$ odd columns (and their even counterparts corresponding to the edge in the reverse direction), the probability that there is no edge between W_1 and W_2 is bounded by:

$$\left(1 - \frac{1}{8} \cdot \left(\frac{\min(\ell/t,1)}{32}\right)^2\right)^{c \cdot \max(t^3/\ell,\ell^2) \log n} \leq \left(\frac{1}{e}\right)^{\frac{\min(\ell^2/t^2,1)}{8 \cdot 32^2} \cdot c \cdot \max(t^3/\ell,\ell^2) \log n}$$

$$= e^{-\frac{c}{8 \cdot 32^2} \max(t\ell,\ell^2) \log n}.$$

We now count the total number of sets W_1 and W_2, and also the total number of sets P and F. There are at most n^ℓ sets P with more than $\ell/2$ and at most ℓ elements. There are at most n^t sets F with (at most) t elements. There are at most 2^ℓ sets W_1, and similarly at most 2^ℓ sets W_2. In total, we can bound the number of sets P, F, W_1, and W_2 by: $n^\ell \cdot n^t \cdot 2^\ell \cdot 2^\ell = 2^{2\ell + (\ell+t)\log n} \leq 2^{3\max(t,\ell)\log n}$. By a union bound over all possible sets, the probability that there exists any sets P and F such that $G(P, F, T', B')$ is not an $\ell/8$-expander is no greater than: $2^{3\max(t,\ell)\log n} \cdot e^{-\frac{c}{8 \cdot 32^2} \max(t\ell,\ell^2) \log n} \leq e^{-\left(\frac{c}{8 \cdot 32^2} - 3\right) \cdot \max(t\ell,\ell^2) \log n}$. Thus, for sufficiently large c, w.h.p., graph $G(P, F, T', B')$ is a $(\ell/8)$-expander for every P and F. □

We now apply the following, proven in [7], to conclude that there is some subset of P with small diameter:

Theorem 2. *Let G be an α-expander. For every set Q of at least 4α nodes, there is a subset $Q^* \subseteq Q$ of at least α nodes such that the subgraph of G induced by set Q^* has diameter of at most $2 \log n$.* □

Corollary 1. *For every set P containing more than $\ell/2$ and at most ℓ processes, for every set F of size at most t, there is a subset $P^* \subseteq P$ containing $\ell/8$ processes such that P^* has diameter at most $2 \log n$ in $G(P, F, T', B')$.* □

We thus conclude that after executing $\langle T_2, B_2 \rangle$, there is some subset P^* of size $\ell/8$ such that every process in P^* knows every rumor:

Lemma 4. *The following event holds w.h.p., for sufficiently large constant c: For every set P with more than $\ell/2$ and at most ℓ processes, for every set F of at most t processes: after executing $\langle T_1, B_1 \rangle$ & $\langle T_2, B_2 \rangle$, there is some subset $P^* \subseteq P$ containing $\ell/8$ honest processes such that every rumor has been received by every process in P^*.*

Proof. Define P^* as per Corollary 1. Recall that P^* has diameter at most $2 \log n$. At the end of $\langle T_1, B_1 \rangle$, i.e., at the end of Stage 1, every rumor is known to some process in P^*, by Lemma 2. In every iteration of $\langle T', B' \rangle$ during Stage 2, rumors are propagated one hop through graph $G(P, F, T', B')$. Thus, during Stage 2, over $2 \log n$ iterations of $\langle T', B' \rangle$, every rumor stored in P^* at the end of Stage 1 is propagated to every other process in P^*. □

4.5 Stage 3: Dissemination

In the third stage, the identified subset P^* distributes the rumors gathered during Stage 2 to the remaining processes in P. We have already shown that in Stage 1, each process in $P \setminus P^*$ successfully sends a message to some process in P^*. As $T_3 = T_1$ and B_3 is the entry-by-entry binary inverse of B_1, each successful sender in Stage 1 becomes a successful receiver in Stage 3 and *vice versa*, in every round. (Note: in the analysis of Stage 1, we considered only events/rounds in which there was only one sender and one receiver.) Thus, each process in $P \setminus P^*$ receives a message from some process in P^*. Thus we conclude:

Lemma 5. *The following event holds w.h.p., for sufficiently large constant c: For every set P with more than $\ell/2$ and at most ℓ of honest processes, and for every set F of at most t processes, after executing $\langle T_1, B_1\rangle$ & $\langle T_2, B_2\rangle$ & $\langle T_3, B_3\rangle$, each process in P has received all rumors of other processes in P.* □

Combining the $\log k$ instances for exponentially decreasing ℓ, and applying the probabilistic argument to Lemma 5 for each instance $A(\ell, \ell/2)$, we conclude:

Theorem 3. *There exists a (k,t)-information exchange algorithm with running time $O(\max(t^3, k^2)\log^2 n)$.* □

5 Extensions

5.1 Energy Usage

An advantage of the wTPM is that it enforces energy efficiency: in each around of $A(\ell, \ell/2)$, only a $\min(1/t, 1/\ell)$ fraction of honest processes are radio-enabled; the remainder cannot access their radios, saving power. Thus, we can show:

Lemma 6. *There exists a (k,t)-information exchange protocol with running time $O(\max(t^3, k^2)\log^2 n)$, where there are an average of $O(\lceil k/t\rceil)$ processes radio-enabled in each round.*

Proof. In protocol $A(\ell, \ell/2)$, in expectation, there are $\leq \min((k+t)/t, (k+t)/\ell) \leq 2\lceil k/t\rceil$ processes radio-enabled in every round. Thus, w.h.p., there are $O(\max(t^3, k^2)\log^2 n \cdot \lceil k/t\rceil)$ processes radio-enabled in the execution. Combined with the time complexity result of Lemma 5, which holds with high probability, and using the probabilistic argument, this implies the claimed result. □

For $t = \Theta(k)$ the per round energy usage is $O(1)$, on average, which is optimal.

5.2 Self-verifying Rumors

We can somewhat improve the previous results when rumors are *self-verifying*, that is, when a process can distinguish a rumor that was initiated at an honest process from a rumor initiated at a malicious process (for example, via public keys or MACs). If a process can *stop early*, i.e., can cease executing the protocol when it believes it has received all available rumors, then we can obtain the following result:

Lemma 7. *There exists a (k,t)-information exchange protocol with running time $O(\max(t^3/k', k^2)\log^2 n)$ where an average of $O(1)$ honest processes are radio-enabled in each round, and k' is the actual number of active honest processes.*

Proof. Consider the protocol as before: $A(k,k/2)$ & $\cdots$ & $A(2,1)$. A process terminates when it completes protocol $A(\ell,\ell/2)$, having already received at least $\ell/2$ rumors: there are clearly at least $\ell/2$ honest processes, and there is no need to continue executing the protocol for smaller ℓ. Since the running time for each $A(\ell,\ell/2)$ is $O(\min(t^3/\ell, k^2)\log^2 n)$, the claimed running time follows.

For energy: assume that $k' \leq k$ honest process are activated. On average, there are $\min((k'+t)/t, (k'+t)/\ell)$ processes enabled in each round. Since the protocol terminates no later than where $\ell > k'/2$, on average there are no more than $O(1)$ processes radio-enabled in each round, by using the similar argument as in the proof of Lemma 6. □

We conjecture that by carefully ordering the $A(\cdot,\cdot)$ instances, and by detecting when to stop, it may be possible to adapt to $|P|$, independent from k and n.

5.3 Continuous Communication

To this point, we have assumed that honest processes are all enabled in the same round, and that they each have exactly one rumor to distribute. In some situations, processes may be activated—and rumors injected—in any round. Consider, then, the following straightforward strategy: instead of executing each instance of $A(\ell,\ell/2)$ sequentially, interleave the executions. That is, divide time into blocks of $\log n$ rounds, and in a round r where $r \mod \log n = k$, execute one round of $A(2^{k+1}, 2^k)$. When a rumor is injected at a process p, it begins participating for a given $A(\ell,\ell/2)$ each time a new instance is started. (Here, a global clock or additional synchronization mechanism must be used). From this we conclude that there exists a *continuous* information exchange protocol where if there are $k \leq n$ rumors active in some round r, for an unknown value k, then all such rumors will be delivered no later than time $r + O(\max(t^3, k^2)\log^3 n)$.

6 Conclusions

We have shown that it is possible to design a wTPM that, by selectively enabling and disabling the radio, facilitates reliable communication. Surprisingly, as long as $t < k^{2/3}$, the resulting protocol is nearly as efficient, in time complexity, as optimal *oblivious information exchange* protocols for networks with no malicious devices. We have also shown a new lower bound indicating that when $k \leq \sqrt{n}$ or when $t \geq k^{2/3}$, the resulting bound is near optimal. The new protocol also provides improved energy efficiency, as existing oblivious solutions (in the model without malicious devices) need $O(k \cdot \min(k^2, n)\log_k n)$ energy [6, 11].

An interesting open question is the performance of protocols such as the one in this paper in multi-hop networks. When there are no malicious devices,

the time complexity of all-to-all communication is $\Theta(n \min(D, \sqrt{n}))$ [23]. When there are Byzantine processes, the situation is more complex, as we need to guarantee that honest processes form a connected component. Another question is whether, by relaxing the restrictions on the wTPM, allowing randomization or some adaptivity, we may be able to achieve even better performance.

References

1. Alistarh, D., Gilbert, S., Guerraoui, R., Milosevic, Z., Newport, C.: Securing your every bit: Reliable broadcast in byzantine wireless networks. In: Proceedings of the Symp. on Parallel Algorithms and Architectures (SPAA). pp. 50–59 (2010)
2. Awerbuch, B., Richa, A.W., Scheideler, C.: A jamming-resistant mac protocol for single-hop wireless networks. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 45–54 (2008)
3. 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. of Computer and System Sciences 45(1), 104–126 (1992)
4. Bhandari, V., Vaidya, N.H.: On reliable broadcast in a radio network. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 138–147 (2005)
5. Bhandari, V., Vaidya, N.H.: On reliable broadcast in a radio network: A simplified characterization. Tech. rep., U. of Illinois at Urbana-Champaign (2005)
6. Bonis, A.D., Gasieniec, L., Vaccaro, U.: Optimal two-stage algorithms for group testing problems. SIAM J. on Computing 34(5), 1253–1270 (2005)
7. Chlebus, B., Kowalski, D.R., Shvartsman, A.A.: Collective asynchronous reading with polylogarithmic worst-case overhead. In: Proceedings of the Symp. on Theory of Computing (STOC). pp. 321–330 (2004)
8. Chlebus, B.S., Kowalski, D.R., Lingas, A.: The do-all problem in broadcast networks. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 117–127 (2001)
9. Clementi, A., Monti, A., Silvestri, R.: Optimal f-reliable protocols for the do-all problem on single-hop wireless networks. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 320–331. Springer, Heidelberg (2002)
10. Clementi, A., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. JPDC 64(1), 89–96 (2004)
11. Clementi, A.E.F., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: Proceedings of the twelfth annual ACM-SIAM Symp. on Discrete algorithms. pp. 709–718 (2001)
12. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a multi-channel radio network: An oblivious approach to coping with malicious interference. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 208–222. Springer, Heidelberg (2007)
13. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Secure communication over radio channels. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 105–114 (2008)
14. Dolev, S., Gilbert, S., Guerraoui, R., Kowalski, D.R., Newport, C., Kuhn, F., Lynch, N.: Reliable Distributed Computing on Unreliable Radio Channels. In: MobiHoc S^3 Workshop (2009)

15. Dolev, S., Gilbert, S., Guerraoui, R., Kuhn, F., Newport, C.: The Wireless Synchronization Problem. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 190–199 (2009)
16. Gilbert, S., Guerraoui, R., Newport, C.: Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 215–229. Springer, Heidelberg (2006)
17. Gilbert, S., Guerraoui, R., Kowalski, D., Newport, C.: Interference-Resilient Information Exchange. In: INFOCOM. pp. 2249–2257 (2009)
18. Goussevskaia, O., Moscibroda, T., Wattenhofer, R.: Local broadcasting in the physical interference model. In: DIALM-POMC. pp. 35–44 (2008)
19. Komlos, J., Greenberg, A.: An asymptotically fast non-adaptive algorithm for conflict resolution in multiple access channels. IEEE Trans. on Information Theory pp. 302–306 (1985)
20. Koo, C.Y.: Broadcast in radio networks tolerating byzantine adversarial behavior. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 275–282 (2004)
21. Koo, C.Y., Bhandari, V., Katz, J., Vaidya, N.H.: Reliable broadcast in radio networks: The bounded collision case. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 258–264 (2006)
22. Kowalski, D.R.: On selection problem in radio networks. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC). pp. 158–166 (2005)
23. Kowalski, D.R., Pelc, A.: Time complexity of radio broadcasting: adaptiveness vs. obliviousness and randomization vs. determinism. Theoretical Computer Science 333(3), 355–371 (2005)
24. Kranakis, E., Krizanc, D., Pelc, A.: Fault-tolerant broadcasting in radio networks. J. of Algorithms 39(1), 47–67 (2001)
25. Kuhn, F., Lynch, N., Newport, C., Oshman, R., Richa, A.: Broadcasting in radio networks with unreliable communication. In: Proceedings of the Symp. on Principles of Distributed Computing (PODC) (2010)
26. Meier, D., Pignolet, Y.A., Schmid, S., Wattenhofer, R.: Speed Dating Despite Jammers. In: Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS). pp. 1–14 (2009)
27. Moscibroda, T., Wattenhofer, R.: The complexity of connectivity in wireless networks. In: INFOCOM (2006)
28. Newport, C.: Distributed Computation on Unreliable Radio Channels. Ph.D. thesis, MIT (2009)
29. Pearson, S., Balacheff, B.: Trusted computing platforms: TCPA technology in context. Prentice Hall (2002)
30. Pippenger, N.: Sorting and selecting in rounds. SIAM J. of Computing 16, 1032–1038 (1987)
31. Strasser, M., Pöpper, C., Capkun, S.: Efficient Uncoordinated FHSS Anti-jamming Communication. In: Proceedings International Symp. on Mobile Ad Hoc Networking and Computing (MOBIHOC). pp. 207–218 (2009)
32. Strasser, M., Pöpper, C., Capkun, S., Cagalj, M.: Jamming-resistant Key Establishment using Uncoordinated Frequency Hopping. In: Proceedings of the Symp. on Security and Privacy. pp. 64–78 (2008)
33. Trusted Computing Group: Trusted platform module (tpm) specifications, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`
34. Willard, D.E.: Log-logarithmic selection resolution protocols in a multiple access channel. SIAM J. of Computing 15(2), 468–477 (1986)

Opportunistic Information Dissemination in Mobile Ad-hoc Networks: The Profit of Global Synchrony*

Antonio Fernández Anta[1,2], Alessia Milani[3], Miguel A. Mosteiro[4,2], and Shmuel Zaks[5]

[1] Institute IMDEA Networks, Leganés, Spain
[2] LADyR, GSyC, Universidad Rey Juan Carlos, Móstoles, Spain
anto@gsyc.es
[3] LIP6, Université Pierre et Marie Curie - Paris 6, Paris, France
alessia.milani@lip6.fr
[4] Department of Computer Science, Rutgers University, Piscataway, NJ, USA
mosteiro@cs.rutgers.edu
[5] Department of Computer Science, Technion, Haifa, Israel
zaks@cs.technion.ac.il

Abstract. The topic of this paper is the study of *Information Dissemination* in Mobile Ad-hoc Networks by means of deterministic protocols. We characterize the connectivity resulting from the movement, from failures and from the fact that nodes may join the computation at different times with two values, α and β, so that, within α time slots, some node that has the information must be connected to some node without it for at least β time slots. The protocols studied are classified into three classes: *oblivious* (the transmission schedule of a node is only a function of its ID), *quasi-oblivious* (the transmission schedule may also depend on a global time), and *adaptive.*

The main contribution of this work concerns negative results. Contrasting the lower and upper bounds derived, interesting complexity gaps among protocol-classes are observed. More precisely, in order to guarantee any progress towards solving the problem, it is shown that β must be at least $n-1$ in general, but that $\beta \in \Omega(n^2/\log n)$ if an oblivious protocol is used. Since quasi-oblivious protocols can guarantee progress with $\beta \in O(n)$, this represents a significant gap, almost linear in β, between oblivious and quasi-oblivious protocols. Regarding the time to complete the dissemination, a lower bound of $\Omega(n\alpha + n^3/\log n)$ is proved for oblivious protocols, which is tight up to a polylogarithmic factor because a constructive $O(n\alpha + n^3 \log n)$ upper bound exists for the same class. It is also proved that adaptive protocols require $\Omega(n\alpha + n^2)$, which is optimal given that a matching upper bound can be proved for quasi-oblivious protocols.

These results show that the gap in time complexity between oblivious and quasi-oblivious, and hence adaptive, protocols is almost linear. This gap is what we call the *profit of global synchrony,* since it represents the gain the network obtains from global synchrony with respect to not having it.

* This research was partially supported by Spanish MICINN grant no. TIN2008-06735-C02-01, Comunidad de Madrid grant no. S2009TIC-1692, EU Marie Curie International Reintegration Grant IRG 210021, NSF grant no. 0937829, ANR grant R-DISCOVER, and by the Israel Science Foundation, grant no. 1249/08.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 374–388, 2010.

1 Introduction

A Mobile Ad-hoc Network (aka MANET) is a set of mobile nodes which communicate over a multihop radio network, without relying on a stable infrastructure. In these networks, nodes are usually battery-operated devices that can communicate via radio with other devices that are in range. Due to unreliable power supply and mobility, nodes may have a continuously changing set of neighbors in that range. This dynamic nature makes it challenging to solve even the simplest communication problems in general. Hence, proposed protocols often have strong synchronization and stability requirements, like having a stable connected network for long enough time.

Current trends in networking-architecture developments, like *delay and disruption tolerant networks,* and *opportunistic networking* [8, 22], aim to deal with the disconnections that naturally and frequently arise in wireless environments. Their objective is to allow communication in dynamic networks, like a MANET, even if a route between sender and receiver never exists in the network. The result is that multi-hop communication is provided through *opportunistic communication,* in which the *online route* of a message is followed one link at a time, as links in the route become available. While the next link is not available, the message is held in a node. With opportunistic communication, strong connectivity requirements are no longer needed. Furthermore, in some cases mobility is the key to allow communication (e.g., consider two disconnected static nodes, where communication between them is provided by a device that, due to mobility, sometimes is in range of one and sometimes of the other).

In this paper, we formally define a particular class of MANET which is suited for opportunistic communication, and which we call *potentially epidemic*. A MANET is potentially epidemic if the changes in the communication topology are such that an online route exists among any two nodes that wish to communicate.

The network is *potentially* epidemic because the actual propagation of the information on the online routes, and then the possibility for a node to affect another node, depends on the stability of each communication links of the online route.

In this context, we define and study the deterministic solvability of a problem that we call *Dissemination.* In this problem, at a given time a source node holds an information that must be disseminated to a given set of nodes belonging to the MANET. The nodes elected to eventually receive the information are the ones that satisfy a given predicate. Depending on this predicate, the Dissemination problem can instantiate most of the common communication problems in distributed systems, such as Broadcast, Multicast, Geocast, Routing, etc.

In particular, we determine assumptions on link stability and speed of nodes under which a distributed deterministic protocol exists that solves Dissemination in potentially epidemic networks. Moreover, we relate the time complexity of the solution to the speed of movement and to the information that protocols may use.

1.1 The Dissemination Problem

We study the problem of disseminating a piece of information, initially held by a distinguished source node, to all nodes of a given set in the network. Formally,

Definition 1. *Given a MANET formed by a set V of n nodes, let $\mathcal{P}$ be a predicate on V and $s \in V$ a node that holds a piece of information I at time t_1 (s is the source of dissemination). The* Dissemination *problem consists of distributing I to the set of nodes $V_{\mathcal{P}} = \{x \in V :: \mathcal{P}(x)\}$. A node that has received I is termed* covered*, and otherwise it is* uncovered*. The Dissemination problem is solved at time slot $t_2 \geq t_1$ if, for every node $v \in V_{\mathcal{P}}$, v is covered by time slot t_2.*

The Dissemination problem abstracts several common problems in distributed systems. E.g. Broadcast, Multicast, Geocast, Routing etc., are all instances of this problem for a particular predicate $\mathcal{P}$. In order to prove lower bounds, we will use one of these instances: the Geocast problem. The predicate $\mathcal{P}$ for Geocast is $\mathcal{P}(x) = \text{true}$ if and only if, at time t_1, x is up and running, and it is located within a parametric distance $d > 0$ (called *eccentricity*) from the position of the source node at that time.

1.2 Model

We consider a MANET formed by a set V of n mobile nodes deployed in $\mathbb{R}^2$, where no pair of nodes can occupy the same point in the plane simultaneously. It is assumed that each node has data-processing and radio-communication capabilities, and a unique identificator number (ID) in $[n] \triangleq \{1, \ldots, n\}$.

Time. Each node is equipped with a clock that ticks at the same uniform rate ρ but, given the asynchronous activation, the clocks of different nodes may start at different times. A time interval of duration $1/\rho$ is long enough to transmit (resp. receive) a message. Computations in each node are assumed to take no time. Starting from a time instant used as reference, the global time is slotted as a sequence of time intervals or *time slots* $1, 2, \ldots$, where slot $i > 0$ corresponds to the time interval $[(i-1)/\rho, i/\rho)$. Without loss of generality [24] all node's ticks are assumed to be in phase with this global tick.

Node Activation. We say that a node is *active* if it is powered up, and *inactive* otherwise. It is assumed that, due to lack of power supply or other unwanted events that we call *failures*, active nodes may become inactive. Likewise, due also to arbitrary events such as replenishing their batteries, nodes may be re-activated. We call the temporal sequence of activation and failures of a node the *activation schedule*. The activation schedule for each node is assumed to be chosen by an adversary, in order to obtain worst-case bounds. Most of the lower-bound arguments included in this paper hold, even if all nodes are activated simultaneously and never fail (which readily provide a global time), making the results obtained stronger.

We assume that a node is activated in the boundary between two consecutive time slots. If a node is activated between slots $t-1$ and t we say that it is activated at slot t, and it is active in that slot. Upon activation, a node immediately starts running from scratch an algorithm previously stored in its hardware, but no other information or status is preserved while a node is inactive. Consequently, it is possible that a covered node does not hold the information I, because it has been inactive after receiving it. To distinguish a covered node that does not hold the information from one that holds it, we introduce the following additional terminology: we say that a node p is *informed* at a given time t if it holds the information I at time t, otherwise p is said to be *uninformed*.

Radio Communication. Nodes communicate via a collision-prone single radio channel. A node v can receive a transmission of another node u in time slot t only if their distance is at most the *range of transmission* r during the whole slot t. The range of transmission is assumed to be the same for all nodes and all time slots. If two nodes u and v are separated by a distance at most r, we say that they are *neighbors*. In this paper, no collision detection mechanism is assumed, and a node cannot receive and transmit at the same time slot. Therefore, an active node u receives a transmission from a neighboring node v at time slot j if and only if v is the only node in u's neighborhood transmitting at time slot j. Also, a node cannot distinguish between a collision and no transmission. In general, we say that a node $v \in V'$ *transmits uniquely* among the nodes of set $V' \subseteq V$ in a slot t if it is the only node in V' that transmits in t.

Link stability. We assume that nodes may move on the plane. Thus, the topology of the network is time dependent. For simplicity, we assume that the topology only changes in the boundaries between time slots. Then, at time slot t nodes u and v are connected by a link in the network topology iff they are neighbors during the whole slot t. An online route between two nodes u and v is a path $u = w_0, w_1, \ldots, w_k = v$ and a sequence of time slots $t_1 < t_2 < \cdots < t_k$ such that the network has a link between w_{i-1} and w_i at time slot t_i. Observe that in order to be able to solve an instance of Dissemination, it is necessary that the network is potentially epidemic. I.e. after the initial time t_1, there is an online route from the source s to every node in $V_{\mathcal{P}}$. However, as argued in [6], worst-case adversarial choice of topologies for a dynamic network precludes any deterministic protocol from completing Broadcast, even if connectivity is guaranteed. Note that Broadcast is an instance of Dissemination, and that if there is connectivity then there are online routes between all nodes. Thus, the property that the network is potentially epidemic as described is not sufficient to solve Dissemination, and further limitations to the adversarial movement and activation schedule are in order. While respecting a bound on the maximum speed $v_{\max}$, which is a parameter, the adversarial movement and activation schedule is limited by the following connectivity property:

Definition 2. *Given a Mobile Ad-hoc Network, an instance of the Dissemination problem that starts at time t_1, and two integers $\alpha \geq 0$ and $\beta \geq 0$, the network is* (α, β)-connected *if, for every time slot $t \geq t_1$ at which the problem has not yet been solved, there is a time slot t' such that the following conditions hold:*

- *the intersection of time intervals $[t, t + \alpha)$ and $[t', t' + \beta)$ is not empty, and*
- *there is a pair of nodes p, p', such that at t' p is informed and p' is uncovered, and they are active and neighbors the whole time interval $[t', t' + \beta)$.*

It is of the utmost importance to notice that (α, β)-connectivity is a characterization that applies to *any* model of dynamic network, given that for any mobility and activation schedule, and any pair of nodes, there is a minimum time they are connected (even if that time is 0) and a maximum time they are disconnected (even if that time is very large). Thus, any dynamic network model used to study the Dissemination problem has its own α and β values.

Due to the same argument, (α, β)-connectivity does not guarantee by itself that the network is epidemic (i.e. that the information is eventually disseminated); instead, an

(α, β)-connected network is only *potentially* epidemic. Consider for instance the source node. Thanks to the (α, β)-connectivity, at most every α slots, the source s is connected to other nodes of the network for at least β time slots. But, we have progress only if the protocol to solve Dissemination is able to use the β slots of connectivity to cover some uncovered node. As a consequence of the above discussed, impossibility results only restrict β, whereas α only constrains the running time, as it is shown in this paper.

1.3 Protocols for Dissemination

We consider distributed deterministic protocols, i.e., we assume that each node in the network is preloaded with its own and possibly different deterministic algorithm that defines a schedule of transmissions for it. Even if a transmission is scheduled for a given node at a given time, that node will not transmit if it is uninformed.

Following the literature on various communication primitives [17,16], a protocol is called *oblivious* if, at each node, the algorithm's decision on whether or not to schedule a transmission at a given time slot depends only on the identifier of the node, and on the number of time slots that the node has been active. Whereas, if no restriction is put on the information that a node may use to decide its communication schedule, the protocol is called *adaptive*. Additionally, in this paper, we distinguish a third class of protocols that we call *quasi-oblivious*. In a quasi-oblivious protocol the sequence of scheduled transmissions of a node depends only on its ID and a global time. Quasi-oblivious protocols have sometimes been called oblivious, since the model assumed simultaneous activation, and hence a global time was readily available. However we prefer to make the difference explicit, as done in [21], because we found a drastic gap between this class and fully oblivious protocols.

1.4 Previous Work

A survey of the vast literature related to Dissemination is beyond the scope of this article. We overview in this section the most relevant previous work. Additionally, a review of relevant related work for static and dynamic networks beyond MANETs can be found in [11].

The Dissemination problem abstracts several common problems in Radio Networks. When some number $1 \leq k \leq n$ of active nodes hold an information that must be disseminated to all nodes in the network, the problem is called *k-Selection* [16] or *Many-to-all* [5]. If $k = 1$ the problem is called *Broadcast* [2,18], whereas if $k = n$ the problem is known as *Gossiping* [9,4]. Upper bounds for these problems in mobile networks may be used for Dissemination, and even those for static networks may apply if the movement of nodes does not preclude the algorithm from completing the task (e.g., round-robin). On the other hand, if only the subset of k nodes have to receive the information, the problem is known as *Multicast* [5,13], and if only nodes initially located at a parametric distance from the source node must receive the information the problem is called Geocast [15], defined in Section 1.1.

Deterministic solutions for the problems above have been studied for MANETs. Their correctness rely on strong synchronization or stability assumptions. In [19], deterministic Broadcast in MANETs was studied under the assumption that nodes move

in a one-dimensional grid knowing their position. Two deterministic Multicast protocols for MANETs are presented in [14,20]. The solutions provided require the network topology to globally stabilize for long enough periods to ensure delivery of messages, and they assume a fixed number of nodes arranged in some logical or physical structure. Leaving aside channel contention, a lower bound of $\Omega(n)$ rounds of communication was proved in [23] for Broadcast in MANETs, even if nodes are allowed to move only in a two-dimensional grid, improving over the $\Omega(D \log n)$ bound of [3], where D is the diameter of the network. This bound was improved to $\Omega(n \log n)$ in [7] without using the movement of nodes, but the diameter of the network in the latter is linear. Recently, deterministic solutions for Geocast were proposed in [1] for a one-dimensional setting and in [10] for the plane. In the latter work, the authors concentrate in the structure of the Geocast problem itself, leaving aside communication issues such as the contention for the communication channel.

1.5 Our Results

The main contribution of this work concerns negative results. Contrasting the lower bounds obtained with upper bounds derived by careful combination of previous techniques, interesting complexity gaps among protocol-classes are observed.

For a model where nodes may fail, there is no global clock, and nodes may be activated at different times, we show in Theorem 6 that any oblivious protocol takes, in the worst case, $\Omega((\alpha + n^2/\ln n)n)$ steps to solve the Geocast problem if $v_{max} > \pi r/6(\alpha + \lfloor (n/3)(n/3-1)/\ln(n/3(n/3-1))\rfloor - 2)$. Given the upper bound of $n(\alpha + 4n(n-1)\ln(2n))$ for Dissemination established in Theorem 8 by means of an oblivious deterministic protocol based on Primed Selection [12], this lower bound is tight up to a poly-logarithmic factor.

Moreover, for the same model, Theorem 5 shows that, even if nodes are activated simultaneously and do not fail, and an adaptive protocol is used, any Geocast protocol takes, in the worst case, $\Omega(n(\alpha+n))$ if $v_{max} > \pi r/(3(2\alpha+n-4))$. This result should be contrasted with the quasi-oblivious protocol based on *Round-Robin* that solves Dissemination in at most $n(\alpha + n)$ steps as established in Theorem 7.

The latter results are asymptotically tight and show that full adaptiveness does not help with respect to quasi-obliviousness. The first lower bound and the last upper bound, show an asymptotic separation almost linear between oblivious and quasi-oblivious protocols. In a more restrictive model, where nodes are activated simultaneously, there exists an oblivious protocol (e.g. Round Robin) that solves Dissemination in at most $n(\alpha + n)$ steps. Hence, the lower bound proved in Theorem 6 shows the additional cost of obliviousness when nodes are not simultaneously activated. This gap is what we call the *profit of global synchrony,* since it represents the gain the network obtains from global synchrony with respect to not having it. Moreover, the quasi-oblivious protocol derived shows that for the Dissemination problem, the simultaneous activation performance can be achieved by distributing the time elapsed since the source started the dissemination. For a discussion of the importance of node-activation schedule in distributed computing refer to [11].

Additionally, it is shown in Theorem 1 that no protocol can solve the Geocast problem (and hence Dissemination) in all (α, β)-connected networks unless $\beta \geq n - 1$.

Interestingly, it is shown in Theorem 2 that this bound becomes $\beta > \lfloor (n-1)(n-3)/4 \ln((n-1)(n-3)/4) \rfloor$ if the protocol is oblivious. Comparing these bounds with the requirements of the protocols presented above, the quasi-oblivious protocol required $\beta \geq n$, which is almost optimal, while the oblivious protocol required $\beta \in \Omega(n^2 \log n)$, which is only a polylogarithmic factor larger than the lower bound. These results also expose another aspect of the profit of global synchrony mentioned before: while $\beta = n$ is enough for quasi-oblivious protocols to solve Dissemination, oblivious protocols require a value of β almost a linear factor larger.

Finally, for an arbitrary small bound on node speed, we show in Theorem 3 the existence of an (α, β)-connected network where Geocast takes at least $\alpha(n-1)$ steps, even using randomization; and the existence of an (α, β)-connected network where any deterministic protocol that transmits periodically takes at least $n(n-1)/2$ steps, even if nodes do not move, in Theorem 4.

1.6 Paper Organization

The rest of the paper is organized as follows. In Section 2 we introduce some technical lemmas that will be used to prove our main results; in Section 3 we prove the lower bounds on link stability and on the time complexity to solve the Dissemination problem with respect to some important aspects of the system (e.g. speed of movement of nodes and their activation schedule) and of the protocols (e.g., obliviousness versus adaptiveness). We finally present the corresponding upper bounds in Section 4.

2 Auxiliary Lemmas

The following lemmas will be used throughout the analysis. A straightforward consequence of the pigeonhole principle is established in the following lemma.

Lemma 1. *For any time step t of the execution of a Dissemination protocol, where a subset V' of k informed nodes do not fail during the interval $[t, t+k-2]$, there exists some node $v \in V'$ such that v does not transmit uniquely among the nodes in V' during the interval $[t, t+k-2]$.*

In the following lemma, we show the existence of an activation schedule such that, for any *oblivious* deterministic protocol, within any subset of at least 3 nodes, there is one that does not have a unique transmission scheduled within a period roughly quadratic in the size of the subset. The proof, based on the probabilistic method, is omitted for brevity and can be found in [11].

Lemma 2. *For any deterministic oblivious protocol that solves Dissemination in a MANET of n nodes, where nodes are activated possibly at different times, and for any subset of k nodes, $k \geq 3$, there exists a node-activation schedule such that, for any time slot t and letting $m = \lfloor k(k-1)/\ln(k(k-1)) \rfloor$, each of the k nodes is activated during the interval $[t-m+1, t]$, and there is one of the k nodes that is not scheduled to transmit uniquely among those k nodes during the interval $[t, t+m-1]$.*

3 Solvability of the Dissemination Problem

If there is at least one node in $V_{\mathcal{P}} - \{s\}$ at least one time slot is needed to solve Dissemination, since the source node has to transmit at least once to pass the information. Furthermore, if all nodes in $V_{\mathcal{P}}$ are neighbors of s, one time slot may also be enough if the source node transmits before neighboring nodes are able to move out of its range. On the other hand, if the latter is not possible, more than one time slot may be needed. Let us consider the Geocast problem. Given that the specific technological details of the radio communication devices used determine the minimum running time when the eccentricity is $d \leq r$, all efficiency lower bounds are shown for $d > r$ unless otherwise stated.

3.1 Link Stability Lower Bounds

The following theorem shows a lower bound on the value of β for the Geocast problem.

Theorem 1. *For any $V_{max} > 0$, $d > r$, $\alpha > 0$, and any deterministic Geocast protocol Π, if $\beta < n - 1$, there exists an (α, β)-connected MANET of n nodes such that Π does not terminate, even if all nodes are activated simultaneously and do not fail.*

Proof. Consider three sets of nodes A, B, and C deployed in the plane, each set deployed in an area of size ε arbitrarily small, such that $0 < \varepsilon < r$ and $d \geq r + \varepsilon$. The invariant in this configuration is that nodes in each set form a clique, every node in A is placed within distance r from every node in B, every node in B is placed at most at distance ε from every node in C, and every node in A is placed at some distance $r < \delta \leq r + \varepsilon$ from every node in C. At the beginning of the first time slot, the adversary places $n - 1$ nodes, including the source node s, in the set C, the remaining node x in set A, and activates all nodes. The set B is initially empty. Given that $d \geq r + \varepsilon$, x must become informed to solve the problem. Also, ε is set appropriately so that a node can move ε distance in one time slot without exceeding V_{max}.

For any protocol Π for Geocast, let t be the first time slot where the source node is the only node to transmit in the set C. Adversarially, let t be the first time slot when the source is informed. Thus, (α, β)-connectivity is preserved up to time slot t for any α. At time slot t, all nodes placed in C are informed.

After time slot t, the adversary moves the nodes as follows. Given that the problem was not solved yet and nodes in C do not fail, according to Lemma 1, there exists a node $y \in C$ that does not transmit uniquely among the nodes in C during the interval $[t + 1, t + n - 2]$. Given that Π is a deterministic protocol, and the adversary knows the protocol and defines the movement of all nodes, the adversary knows which is the node y.

Assume, for the sake of contradiction, that $\beta \leq n - 2$. Then, the adversary places y in B for all time slots in the interval $[t + 1, t + \beta]$. Additionally, for each time slot $t' \in [t + 1, t + \beta]$ where y transmits, the adversary moves to B some node $z \in C$ that transmits at t' to produce a collision. At the end of each time slot t' the adversary moves z back to C. Such a node z exists since y does not transmit uniquely during the interval $[t+1, t+n-2]$ and $n-2 \geq \beta$. At the end of time slot $t+\beta$, the adversary moves y back

to C and the above argument can be repeated forever preserving the (α, β)-connectivity and precluding Π from solving the problem. Therefore, β must be at least $n-1$.

Building upon the argument used in the previous theorem, but additionally exploiting the adversarial node activation, the following theorem shows a lower bound on the value of β for the Geocast problem if the protocol used is oblivious. The idea of the proof is to split evenly the nodes of set C in the proof of Theorem 1 in two groups, so that alternately the nodes in one group are activated while the nodes in the other group produce collisions. The details are omitted for brevity and can be found in [11].

Theorem 2. *For any $V_{max} > 0$, $d > r$, $n \geq 8$, $\alpha > 0$, and any deterministic oblivious protocol for Geocast Π, if $\beta \leq m = \lfloor (n-1)(n-3)/4 \ln((n-1)(n-3)/4) \rfloor$, there exists an (α, β)-connected MANET of n nodes such that Π does not terminate.*

3.2 Time Complexity Lower Bounds versus Speed, Activation and Obliviousness

Exploiting the maximum time α that a partition can be disconnected, a lower bound on the time efficiency of any protocol for Geocast, even regardless of the use of randomization and even for arbitrarily slow node-movement, can be proved. The following theorem establishes that bound. The proof is omitted for brevity and can be found in [11].

Theorem 3. *For any $V_{max} > 0$, $d > r$, $\alpha > 0$, and $\beta > 0$, there exists an (α, β)-connected MANET of n nodes, for which any Geocast protocol takes at least $\alpha(n-1)$ time slots, even if all nodes are activated simultaneously and do not fail.*

The linear lower bound for Geocast proved in Theorem 3 was shown exploiting the maximum time of disconnection between partitions. Exploiting the adversarial schedule of node activation, even if nodes do not move nor fail, the same bound can be simply proved for arbitrary Geocast protocols, while a quadratic bound can be shown for the important class of *equiperiodic* protocols. The protocol definition and the theorem for the latter follows. The proof is omitted for brevity and can be found in [11].

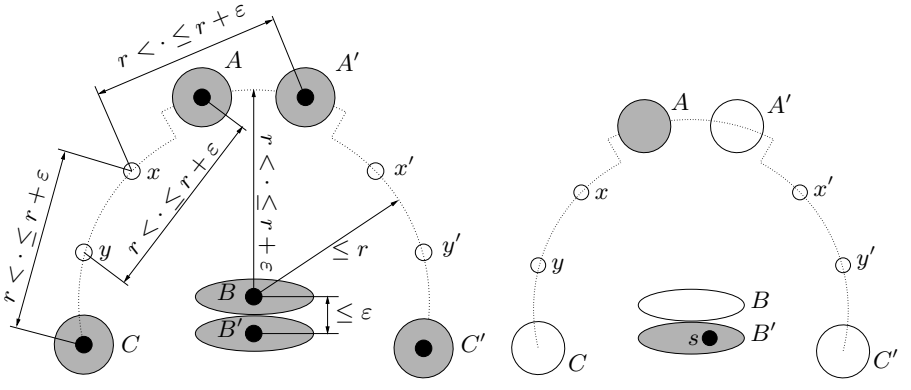
Definition 3. *A protocol of communication for a Radio Network is* equiperiodic *if for each node, the transmissions scheduled are such that the number of consecutive time steps without transmitting, call it $T-1$, is always the same. We say that T is the* period *of transmission of such a node.*

Theorem 4. *For any $V_{max} \geq 0$, $d > r$, $\alpha > 0$, $\beta > 0$, and any deterministic equiperiodic Geocast protocol Π, there exists an (α, β)-connected MANET of n nodes, for which Π takes at least $n(n-1)/2$ time slots to solve the problem, even if nodes do not fail and do not move.*

In Theorems 3 and 4 we showed lower bounds for Geocast for arbitrarily small values of V_{max}. We now show that, by slightly constraining V_{max}, a quadratic lower bound can also be shown for arbitrary deterministic protocols.

Theorem 5. *For any $V_{max} > \pi r/(3(2\alpha + n - 4))$, $d > r$, $\alpha > 0$, $\beta > 0$, and any deterministic Geocast protocol Π, there exists an (α, β)-connected MANET of n nodes, for which Π takes $\Omega((\alpha + n)n)$ time slots to solve the problem, even if all nodes are activated simultaneously and do not fail.*

Proof. The following adversarial configuration and movement of nodes shows the claimed lower bound. Consider six sets of nodes A, A', B, B', C, and C', each deployed in an area of size ε arbitrarily small, such that $0 < \varepsilon < r$ and $d \geq r + \varepsilon$, and four points, x, y, x', and y' placed in the configuration depicted in Figure 1(a).



(a) Distances invariant. (b) Initial configuration.

Fig. 1. Illustration of Theorem 5. A small empty circle depicts a point in the plane. A small black circle depicts a node. A big empty area depicts an empty set. A big shaded area depicts a non-empty set.

The invariant in these sets is that nodes in each set form a clique; each node in A' is placed at some distance $> r$ and $\leq r + \varepsilon$ from the points x, y', and each node in B; each node in A is placed at some distance $> r$ and $\leq r + \varepsilon$ from the points x', y, and each node in B; each node in B is placed within distance r of points x, y, x', and y', and each node in C and C'; each node in C is placed at some distance $> r$ and $\leq r + \varepsilon$ from the point x; each node in C' is placed at some distance $> r$ and $\leq r + \varepsilon$ from the point x'; and each node in B' is placed within distance ε of each node in B and within distance r of each node in C and C'.[1]

At the beginning of the first time slot, the adversary places $n/2$ nodes, including the source node s, in set B', the remaining $n/2$ nodes in the set A, and starts up all nodes. (For clarity, assume that n is even.) All the other sets are initially empty. (See Figure 1(b).)Given that $d \geq r + \varepsilon$, all nodes must be covered to solve the problem. Also, ε is set appropriately so that a node can be moved ε distance in one time slot without exceeding V_{max}, and so that a node can be moved from set A to point x through the curved parts of the dotted line (see Figure 1(a)),of length less than $\pi(r + \varepsilon)/6$, in

[1] During some periods of time a given set could be empty, we mean that x is separated (within) that distance from any point in the area designated to the set X

$\alpha + n/2 - 2$ time slots without exceeding V_{max}. (To see why the length bound is that, it is useful to notice that the distance between each pair of singular points along each of the circular dotted lines is upper bounded by $(r + \varepsilon)/2$.)

Let t be the first time slot when the source is scheduled to transmit. Adversarially, let t be the first time slot when the source is informed. Nodes stay in the positions described until t and, consequently, all the other $n/2-1$ nodes in set B' receive it. Starting at time slot $t+1$, the adversary moves the nodes so that only one new node every $\alpha+n/2$ steps becomes informed. First we give the intuition of the movements and later the details. (See Figure 1(b).)Nodes that are not in B or B' are moved following the dotted lines. Some of the nodes in B' are moved back and forth to B. Nodes initially in A are moved clockwise to A', except for one of them, say u, which is moved simultaneously counter-clockwise to the point x. Upon reaching A' nodes are moved counter-clockwise back to A, except for one of them, say v, which is moved simultaneously clockwise to the point x', while the node u is also moved simultaneously to the point y. Upon reaching A, the remaining nodes repeat the procedure while u keeps moving towards C and v keeps moving towards C' through y' respectively. Nodes initially in A are moved in the above alternating fashion, one to C and the next one to C', until all nodes become informed. Movements are produced so that (α, β)-connectivity is preserved. The details follow.

The movement of each node u moved from A to C is carried out in three phases of at least $\alpha + n/2 - 2$ time slots each as follows. (As explained below, some nodes initially in A will be moved instead to C', but the movement is symmetric. For clarity, we only describe the whole three phases for one node. The movement is illustrated in Figure 4 in [11], which is omitted here for brevity.)

Phase 1 During the first $\alpha - 2$ time slots, u is moved counterclockwise from A towards the point x maintaining a distance $> r$ and $\leq r + \varepsilon$ with respect to every node in B. In the $(\alpha - 1)$-th time slot of this phase, u is moved within distance r of every node in set B preserving (α, β)-connectivity. Nodes in B' stay static during this interval. Given that only nodes in B' are informed and the distance between them and u is bigger than r, u does not become covered during this interval.

During the following $n/2-1$ time slots of the first phase, the counterclockwise movement of node u towards the point x continues, but now maintaining a distance at most r with respect to every node in B. In the last time slot of the second phase, u is moved to point x. During this interval, nodes in B' are moved back and forth to B as described in Theorem 1 to guarantee that u does not become covered before reaching point x. Upon reaching point x, u and all the other nodes in the network not in B or B' remain static. Phase 1 ends the time slot before u becomes covered.

Simultaneously, along the first $\alpha+n/2-2$ time steps of this phase, the remaining nodes initially in A are moved clockwise to A'. Then, even if u becomes informed immediately upon reaching point x, u cannot inform nodes in A' because they are separated by a distance $> r$.

Phase 2 During this phase, u is moved counterclockwise towards point y maintaining a distance at most r with respect to every node in B and B'. Simultaneously,

nodes that were in A' at the end of the second phase are moved counterclockwise to A except for one node v that moves in its own first phase to x'.
Nodes moving from A' to A maintain a distance $> r$ with respect to u. Thus, even if u becomes covered the information cannot be passed to the former. At the end of this phase v is placed in point x'. Thus, even if v becomes covered in the first step of its second phase, v cannot inform nodes in A because they are separated by a distance $> r$.

Phase 3 During this phase, u is moved counterclockwise towards set C maintaining a distance at most r with respect to every node in B and B'. Simultaneously, nodes that were in A at the end of the second phase are moved clockwise to A' except for one node w that moves in its own first phase to x. Also simultaneously, v continues its movement towards set C' in its own second phase.
Nodes moving from A to A' maintain a distance $> r$ with respect to v. Thus, even if v becomes covered the information cannot be passed to the former. Also, nodes u and w are moved maintaining a distance $> r$ between them. Thus, u cannot inform w. At the end of this phase u has reached set C, v is placed in point y', and w is placed in point x. Thus, even if w becomes covered in the first step of its second phase, w cannot inform nodes in A because they are separated by a distance $> r$. Upon completing the third phase, u stays static in C forever so that (α, β)-connectivity is preserved.

The three-phase movement detailed above is produced for each node initially in A, overlapping the phases as described, until all nodes have became covered. Given that when a node u reaches the point x, its phase 1 is stretched until the time step before u becomes covered by a node v in B and all other nodes remain static, the next node w that will be moved from A' to x' does not become covered by v, because w stays in A' until u becomes covered. In each phase of at least $\alpha + n/2 - 2$ time slots every node is moved a distance at most $\pi(r + \varepsilon)/6 + \varepsilon$. Thus, setting ε appropriately, the adversarial movement described does not violate V_{max}. Given that $n/2$ nodes initially in A are covered one by one, each at least within $\alpha + n/2 - 2$ time slots after the previous one, the overall running time is lower bounded as claimed, even if $t = 1$.

The quadratic lower bound shown in Theorem 5 holds for any deterministic protocol, even if it is adaptive. Building upon the argument used in that theorem, but additionally exploiting the adversarial node activation, the following theorem shows a roughly cubic lower bound for oblivious protocols, even relaxing the constraint on V_{max}. The proof is omitted for brevity, the details can be found in [11].

Theorem 6. *For any $n \geq 9$, $d > r$, $\alpha > 0$, $\beta > 0$, $V_{max} > \pi r/6(\alpha + \lfloor (n/3)(n/3 - 1)/\ln(n/3(n/3-1))\rfloor - 2)$, and any oblivious deterministic Geocast protocol Π, there exists an (α, β)-connected MANET of n nodes, for which Π takes $\Omega((\alpha + n^2/\ln n)n)$ time slots to solve the problem.*

4 Upper Bounds

Solving the Dissemination problem under arbitrary node-activation schedule and node-movement is not a trivial task. To the best of our knowledge, deterministic protocols for

such scenarios were not studied before, not even for potentially epidemic networks such as an (α, β)-connected MANET, and not even for specific instances of Dissemination. In this section, a quasi-oblivious protocol and an oblivious one that solve Dissemination, both based on known algorithms particularly suited for our setting, are described and their time efficiency proved. The first bound is asymptotically tight with respect to the more powerful class of adaptive protocols.

A Quasi-Oblivious Protocol. The idea behind the protocol is to augment the well-known Round-Robin protocol with the synchronization of the clock of each node with the time elapsed since the dissemination started, which we call the *global time*. This is done by embedding a counter τ, corresponding to the global time, in the messages exchanged to disseminate the information I. Given that the schedule of transmissions of a node depends only on its ID and the global time, the protocol is quasi-oblivious. More details about the algorithm can be found in [11].

It can be proved that this quasi-oblivious algorithm solves Dissemination for arbitrary values of V_{max} in at most $n(\alpha + n)$ time steps. The details are omitted for brevity and can be found in [11]. Formally,

Theorem 7. *Given an (α, β)-connected MANET where $\beta \geq n$, there exists a quasi-oblivious deterministic protocol that solves Dissemination for arbitrary values of V_{max} in at most $n(\alpha + n)$ time steps.*

Recall that $\beta \geq n-1$ is required for the problem to be solvable as shown in Theorem 1. This upper bound is asymptotically tight with respect to the lower bound for general deterministic Geocast protocols when $V_{max} > \pi r/(3(2\alpha+n-4))$ shown in Theorem 5. Thus, we can conclude that having extra information in this case does not help.

An Oblivious Protocol. We finally describe how to implement an oblivious protocol for Dissemination, based on *Primed Selection*, a protocol presented in [12] for the related problem of Recurrent Communication. Given that in this protocol the schedule of transmissions of a node depends only on its ID, the protocol is oblivious. This upper bound is only a poly-logarithmic factor away from the lower bound shown in Theorem 6.

In order to implement Primed Selection, one of n prime numbers is stored in advance in each node's memory, so that each node holds a different prime number. Let p_ℓ denote the ℓ-th prime number. We set the smallest prime number used to be p_n, which is at least n, because Primed Selection requires the smallest prime number to be at least the maximum number of neighbors of any node, which in our model is unknown. The algorithm is simple to describe, upon receiving the information, each node with assigned prime number p_i transmits with period p_i.

It was shown in [12] that, for any Radio Network formed by a set V of nodes running Primed Selection, for any time slot t, and for any node i such that the number of nodes neighboring i is $k-1$, $1 < k < n$, i receives a transmission without collision from each of its neighbors within at most $k \max_{j \in V} p_j$ steps after t. Given that in our setting the biggest prime number used is p_{2n-1}, that $p_x < x(\ln x + \ln \ln x)$ for any $x \geq 6$ as shown in [25], and that due to mobility all nodes may get close to i in the worst case, we have that $k \max_{j \in V} p_j < n(2n-1)(\ln(2n-1) + \ln \ln(2n-1))$, for $n \geq 4$. Which

is in turn less than $4n(n-1)\ln(2n)$ for $n \geq 3$. Hence, given that in the worst case all nodes must be covered at least one at a time and that the network is (α, β)-connected, the overall running time is less than $n(\alpha + 4n(n-1)\ln(2n))$. We formalize this bound in the following theorem. Recall that $\beta > \lfloor (n-1)(n-3)/4 \ln((n-1)(n-3)/4) \rfloor$ is required for the problem to be solvable when $n \geq 8$ as shown in Theorem 2.

Theorem 8. *Given an (α, β)-connected MANET, where $\beta \geq n(2n-1)(\ln(2n-1) + \ln\ln(2n-1))$ and $n \geq 4$, there exists an oblivious deterministic protocol that solves Dissemination for arbitrary values of V_{max} in at most $n(\alpha + 4n(n-1)\ln(2n))$ time steps.*

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. 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. Journal of Computer and System Sciences 45, 104–126 (1992)
3. Bruschi, D., Del Pinto, M.: Lower bounds for the broadcast problem in mobile radio networks. Distributed Computing 10(3), 129–135 (1997)
4. Chlebus, B., Gąsieniec, L., Lingas, A., Pagourtzis, A.: Oblivious gossiping in ad-hoc radio networks. In: Proc. of 5th Intl. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, pp. 44–51 (2001)
5. Chlebus, B.S., Kowalski, D.R., Radzik, T.: On many-to-many communication in packet radio networks. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 260–274. Springer, Heidelberg (2006)
6. Clementi, A.E.F., Pasquale, F., Monti, A., Silvestri, R.: Communication in dynamic radio networks. In: Proc. 26th Ann. ACM Symp. on Principles of Distributed Computing, pp. 205–214 (2007)
7. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. Journal of Discrete Algorithms 5, 187–201 (2007)
8. Fall, K.: A delay-tolerant network architecture for challenged internets. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), pp. 27–34 (2003)
9. Farach-Colton, M., Mosteiro, M.A.: Sensor network gossiping or how to break the broadcast lower bound. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 232–243. Springer, Heidelberg (2007)
10. Fernández Anta, A., Milani, A.: Bounds for deterministic reliable geocast in mobile ad-hoc networks. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 164–183. Springer, Heidelberg (2008)
11. Fernández Anta, A., Milani, A., Mosteiro, M.A., Zaks, S.: Opportunistic information dissemination in mobile ad-hoc networks: The profit of global synchrony. Technical Report RoSaC-2010-1, GSyC, Universidad Rey Juan Carlos (2010)
12. Fernández Anta, A., Mosteiro, M.A., Thraves, C.: Deterministic communication in the weak sensor model. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 119–131. Springer, Heidelberg (2007)
13. Gasieniec, L., Kranakis, E., Pelc, A., Xin, Q.: Deterministic m2m multicast in radio networks. In: Proc. of 31st Intl. Colloquium on Automata Languages and Programming, pp. 670–682 (2004)

14. Gupta, S.K.S., Srimani, P.K.: An adaptive protocol for reliable multicast in mobile multi-hop radio networks. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (1999)
15. Jinag, X., Camp, T.: A review of geocasting protocols for a mobile ad hoc network. In: Proceedings of Grace Hopper Celebration (2002)
16. Kowalski, D.R.: On selection problem in radio networks. In: Proc. 24th Ann. ACM Symp. on Principles of Distributed Computing, pp. 158–166 (2005)
17. Kowalski, D.R., Pelc, A.: Time complexity of radio broadcasting: Adaptiveness vs. obliviousness and randomization vs. determinism. Theoretical Computer Science 333, 355–371 (2005)
18. Kushilevitz, E., Mansour, Y.: An $\Omega(D \log(N/D))$ lower bound for broadcast in radio networks. SIAM Journal on Computing 27(3), 702–712 (1998)
19. 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, p. 233 (2006)
20. Pagani, E., Rossi, G.P.: Reliable broadcast in mobile multihop packet networks. In: Proc. of the 3rd ACM Ann. Intl. Conference on Mobile Computing and Networking, pp. 34–42 (1997)
21. Peleg, D., Radzik, T.: Time-efficient broadcast in radio networks. In: Koster, A.M.C.A., Muñoz, X. (eds.) Graphs and Algorithms in Communication Networks, ch, 12, pp. 315–339. Springer, Heidelberg (2009)
22. Pelusi, L., Passarella, A., Conti, M.: Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. IEEE Communications Magazine 44(11), 134–141 (2006)
23. Prakash, R., Schiper, A., Mohsin, M., Cavin, D., Sasson, Y.: A lower bound for broadcasting in mobile ad hoc networks. Technical report, Ecole Polytechnique Federale de Lausanne (2004)
24. Roberts, L.G.: Aloha packet system with and without slots and capture. Computer Communication Review 5(2), 28–42 (1975)
25. Rosser, J.B., Schoenfeld, L.: Approximate formulas for some functions of prime numbers. Illinois Journal of Mathematics 6(1), 64–94 (1962)

Brief Announcement: Failure Detectors Encapsulate Fairness*

Scott M. Pike, Srikanth Sastry, and Jennifer L. Welch

Dept. of Computer Science and Engineering
Texas A&M University
College Station, TX 77843, USA
{pike,sastry,welch}@cse.tamu.edu

Abstract. Fairness is a measure of the number of steps a process takes relative to other processes and/or messages in transit. We argue that failure detectors encapsulate fairness. As evidence, we specify models for fairness-based message-passing systems that are the weakest to implement the Chandra-Toueg failure detectors from [1]

Failure Detectors and Partial Synchrony. Failure detectors [1] — system services that provide information about process crashes in an otherwise asynchronous system — are believed to encapsulate partial synchrony [3]. That is, given a partially-synchronous system model M, it is possible to replace M with an asynchronous system augmented with an appropriate failure detector $\mathcal{D}$ (that is implementable in M) such that all the problems solvable in M are also solvable in the asynchronous system augmented with $\mathcal{D}$. This belief has led to the pursuit to find the 'weakest' system models to implement various failure detectors.

Such pursuits have met with limited success, in part, because many such system models are specified with respect to real-time constraints on communication and computation. We argue that such 'weakest' system models should be specified as restrictions on *fairness*: a measure of the number of steps executed by a process relative to other events in the system.

Failure Detector Classes. We consider four failure detector classes from [1]:

1. *Perfect failure detector* (denoted $\mathcal{P}$) never suspects live processes and eventually suspects all crashed processes forever.
2. *Strong failure detector* (denoted $\mathcal{S}$) never suspects *some* correct process and eventually suspects all crashed processes forever.
3. *Eventually Perfect failure detector* (denoted $\Diamond\mathcal{P}$) eventually never suspects correct processes and suspects all crashed processes forever.
4. *Eventually Strong failure detector* (denoted $\Diamond\mathcal{S}$) eventually never suspects some correct process and suspects all crashed processes forever.

* This work was supported in part by NSF grant 0964696 and Texas Higher Education Coordinating Board grant NHARP 000512-0130-2007.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 389–391, 2010.

Fairness. We focus on fairness properties that refer to both the computation and the communication that takes place in executions. *Computational fairness* specifies the number of steps executed by processes relative to each other; *communicational fairness* specifies the number of steps executed by the recipient of a message while that message is in transit.

Computational Fairness. A process i is said to be *k-proc-distinguished* if all processes are *fair* with respect to i; that is, in each execution segment where any process j takes $k+1$ steps, then either (1) i takes at least one step, or (2) i has crashed. If all processes are fair with respect to i only after some unknown global stabilization time (GST), then i is said to be *eventually k-proc-distinguished*. Note that while other processes may be fair with respect to a proc-distinguished process i, process i need not be fair with respect to other processes; *i.e.*, a proc-distinguished process may take an unbounded number of steps in the duration between two consecutive steps by a non-proc-distinguished process. This is an important distinction between computational fairness and bounded relative process speeds defined in [3]. Bounded relative process speeds may be viewed as a special case where every process is (eventually) k-proc-distinguished.

Communicational Fairness. A process i is said to be *d-com-distinguished* if all other processes are *fair* with respect to messages from i; that is, for every message m sent by i to j, the recipient j takes no more than d steps while m is in transit and i is not crashed. Note that if i crashes while m is in transit to j, then the bound d does not apply. If all processes are fair with respect to messages from i only after some unknown GST, then i is said to be *eventually d-com-distinguished*. Like computational fairness, it is not necessary for a d-com-distinguished process i to be fair with respect to messages sent from non-com-distinguished processes to i. That is, i may take an unbounded number of steps while a message from a non-com-distinguished process is in transit to i.

Fairness-Based Partial Synchrony. We present four partially synchronous models — $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, and $\Diamond\mathcal{SF}$ — that specify fairness properties rather than real-time behavior.

1. *All Fair* ($\mathcal{AF}$) is specified as follows: in every run, all processes are both k-proc-distinguished and d-com-distinguished, for known k and d.
2. *Some Fair* ($\mathcal{SF}$) is specified as follows: in every run, some correct process i is both k-proc-distinguished and d-com-distinguished, for known k and d.
3. *Eventually All Fair* ($\Diamond\mathcal{AF}$) is specified as follows: for each run, there exists a (potentially unknown) time after which the system behaves like $\mathcal{AF}$.
4. *Eventually Some Fair* ($\Diamond\mathcal{SF}$) is specified as follows: for each run, there exists a (potentially unknown) time after which the system behaves like $\mathcal{SF}$.

Weakest System Models. Our primary result states that $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, and $\Diamond\mathcal{SF}$ specify the exact amount of fairness encapsulated by $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{P}$, and $\Diamond\mathcal{S}$, respectively. In other words, $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, and $\Diamond\mathcal{SF}$ are the 'weakest' system models to respectively implement $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{P}$, and $\Diamond\mathcal{S}$ with any number of crashes.

We establish this result is as follows. First, we present a construction that uses any failure detector from [1] in an otherwise asynchronous system to schedule distributed applications such that each process executes its application steps fairly with respect to other processes (and messages). The fairness properties guaranteed by the scheduler depend on the failure detector. By employing $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{P}$, or $\Diamond\mathcal{S}$, the scheduler provides fairness guarantees specified by $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, or $\Diamond\mathcal{SF}$, respectively. This shows that the failure detectors encapsulate at least as much fairness as is specified in the corresponding fairness-based system models. For the other direction, we present an algorithm which implements a failure detector on top of these fairness-based systems. When this algorithm is deployed in $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, or $\Diamond\mathcal{SF}$, it implements $\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{P}$, or $\Diamond\mathcal{S}$, respectively. Thus, we show that these failure detectors encapsulate no more guarantees on fairness than what is provided by the corresponding fairness-based systems.

Discussion. The notion of 'capturing the power' of a failure detector was explored in [5] for shared-memory systems. Our work, which focuses on message-passing systems, deviates from [5] in three significant ways. (1) The 'power' of a failure detector is different in shared-memory and message-passing systems. For example, in environments with arbitrary number of process crashes, the weakest failure detectors for consensus and k-set agreement are different in shared-memory and message-passing systems. Hence, 'capturing the power' of a failure detector in message-passing systems merits separate investigation. (2) Asynchronous systems under message passing are weaker than under shared memory because the quorum failure detector is necessary to implement read/write atomicity with message passing. Hence, the results from [5] need not carry over to message-passing systems. (3) We address the synchronism captured by perpetually accurate oracles (in addition to eventually accurate ones) and resolve the issue of *synchronous-systems-vs.-perfect-failure-detector* [2] whereas the results in [5] specify fairness constraints only for classes of eventually accurate oracles.

The full version of the paper can be found at [4].

References

1. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996), http://dx.doi.org/10.1145/226643.226647
2. Charron-Bost, B., Guerraoui, R., Schiper, A.: Synchronous system and perfect failure detector: solvability and efficiency issues. In: International Conference on Dependable Systems and Networks, pp. 523–532 (2000), http://dx.doi.org/10.1109/ICDSN.2000.857585
3. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988), http://dx.doi.org/10.1145/42282.42283
4. Pike, S.M., Sastry, S., Welch, J.L.: Failure detectors encapsulate fairness. Tech. Rep. 2010-7-1, Dept. of Computer Science and Engineering, Texas A&M University (2010), http://www.cse.tamu.edu/academics/tr/2010-7-1
5. Rajsbaum, S., Raynal, M., Travers, C.: The iterated restricted immediate snapshot model. In: Hu, X., Wang, J. (eds.) COCOON 2008. LNCS, vol. 5092, pp. 487–497. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-69733-6_48

Brief Announcement: Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems - A Case Study

Francesco Alberti[1], Silvio Ghilardi[2], Elena Pagani[2], Silvio Ranise[1], and Gian Paolo Rossi[2]

[1] FBK-Irst, Trento, Italia
[2] Università degli Studi di Milano, Milano, Italia

Background and motivations. Algorithms for ensuring fault tolerance are key ingredients in many applications such as avionics and networking. There is an increasing demand to integrate (formal) validation in the design process of these algorithms as they are often part of safety critical systems. When validation fails, the designer would benefit from tracking the sequence of events that led to an incorrect state to recover the error. To productively integrate formal verification in the design phase, tools should be able to return such error traces. Fault tolerant algorithms are often parametric, which makes their automated verification a daunting task. Indeed, checking that an algorithm satisfies a certain property requires to prove it *for any number of processes.*

Contributions. We propose the use of an infinite state model checker for safety properties, called MCMT [3] (`http://www.dsi.unimi.it/~ghilardi/mcmt`), to assist in the design of the considered class of algorithms. MCMT is particularly suitable for this purpose because it is based on a declarative framework in which parametric algorithms can be naturally specified by using first-order formulae I, Tr, and U for the set of initial states, the transitions, and the set of undesired states, respectively. The distinguishing feature of MCMT is that it applies the so-called *backward reachability procedure* [1] in a symbolic setting: a tree whose nodes are labeled by the formulae describing the pre-images of U with respect to the transitions in Tr is constructed and visited on-the-fly. The visit is interleaved with fix-point and safety checks so as to decide when the process can stop. To mechanize this, MCMT puts some *constraints on the format* of I, Tr, and U so that (a) the class of formulae describing the set of backward reachable states are closed under pre-image computation and (b) both fix-point and safety checks can be reduced to decidable logical problems, called Satisfiability Modulo Theories (SMT) [7] problems. Facts (a) and (b) rely on results precisely stated and proved in [2]; satisfiability tests involving *quantified formulae* are preprocessed by manual instantiations - subject to powerful but complete heuristics - and are then discharged via the integration with the SMT-solver YICES (`http://yices.csl.sri.com`).

The **first contribution** of our work is the definition of a sub-set of the formal framework [2] underlying MCMT, which is suitable for the specification of fault tolerant algorithms.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 392–394, 2010.

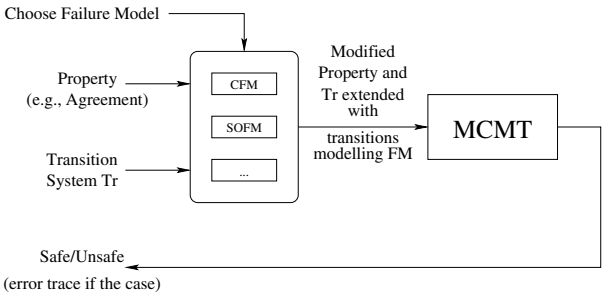


Fig. 1. A methodology for designing fault tolerant algorithms

In order to describe the possible misbehaviors of real distributed systems, a taxonomy of failure models has been presented in the literature [6]: such taxonomy abstracts relevant features concerning faults that may occur in practice. For example, the simplest of such models is the Crash Failure Model (CFM) where processes may halt at any time. A slightly more realistic model consists of considering the possibility that a process may omit to send a message besides crashing at any time, called the Send Omission Failure Model (SOFM).

Since MCMT natively supports only the simplest failure model (i.e. the CFM) [5], the **second contribution** of our work is a technique to re-write the specification of an algorithm for a certain failure model, so as to represent more complex failures and enable the validation of distributed algorithms under critical conditions. The underlying idea is to add a local state variable to each process as a flag signaling whether the process is faulty or not, and to enlarge the specification with faulty behaviors. Then, we propose a design methodology for parametrized and fault tolerant algorithms (see Figure 1) that exploits the previous two contributions consisting of (i) specifying the algorithm and its safety property, (ii) choosing a failure model (e.g., the CFM or the SOFM), (iii) invoking MCMT, and (iv) modifying the specification of the algorithm according to the analysis of the error trace returned by the tool (if any), before repeating the procedure. This schema can be easily blended with a standard incremental and iterative approach to design.

The **third contribution** is to apply this methodology to replay the design of the reliable broadcast algorithms of Chandra and Toueg in [8]. We focus on Agreement as the safety property to check. The authors of [8] consider several parametric algorithms, obtained by stepwise refinement. Table 1 shows the results of our experiments. Algorithm 1 is the simplest one, designed to be correct with the CFM (first column of Table 1). Its unsafety w.r.t. the SOFM is quickly established by MCMT by finding a shorter error trace than the one described in [8] (second column of Table 1). Algorithm 1e is a first refinement of Algorithm 1, which is still found to be unsafe. In this case, the error trace found by MCMT, after a manual analysis, corresponds to that described in [8]. Algorithm 2 is the second refinement and its up-front verification turned out to be quite problematic, even using MCMT invariant synthesis capabilities [4]. Fortunately,

Table 1. MCMT performances

	Algo. 1, CFM	Algo. 1, SOFM	Algo. 1e, SOFM	Algo. 2, SOFM
Safe (agreement)	Yes	No	No	Yes
time (sec)	1.18	17.66	1,709.93	4,719.51
# state vars	8	9	11	15
# transitions	13	13+3	16+6	22+6
# nodes	113	464	9,679	11,158
# SMT calls	2,792	20,009	1,338,058	2,558,986
Length unsafe trace	×	11	33	×
# invariants	×	×	×	19 (+**7**)

Timings obtained on an Intel Core Duo 2.66 GHz with 2 GB, running Debian Linux. (The complete specifications of the algorithms considered here can be downloaded at `http://www.falberti.it/reliableBroadcast`.)

the possibility to interact with MCMT allowed us to add 7 more system properties (e.g., "there is only one coordinator at a time") to the specification. These properties, and 19 more invariants automatically found by the tool, have been validated by MCMT before their use, thus guaranteeing that their inclusion does not affect the main verification result. On the other hand, their adoption significantly pruned the backward search tree and yielded the performances reported in the last column of Table 1. The tool also validated the three lemmata used in [8] to perform the pen-and-paper proof of the correctness of the algorithm. These results show the practical viability of our technique.

There are two main lines for future work. First, we would like to consider more general failure models (e.g., general omission) to conclude the formal validation of the reliable broadcast algorithms in [8]. Second, we intend to refine our models so as to consider temporal constraints that would widen the scope of applicability of our techniques to more realistic algorithms.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proc. of LICS, pp. 313–321 (1996)
2. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT Model-Checking of Array-based Systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008)
3. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. To appear in Proc. of IJCAR (July 2010)
4. Ghilardi, S., Ranise, S.: Goal Directed Invariant Synthesis for Model Checking Modulo Theories. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS (LNAI), vol. 5607, pp. 173–188. Springer, Heidelberg (2009)
5. Ghilardi, S., Ranise, S.: A note on the stopping failure model (Unpublished note), `http://homes.dsi.unimi.it/~ghilardi/mcmt/stop_fail_note.pdf`
6. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
7. The SMT-LIB initiative, `http://www.SMT-LIB.org`
8. Toueg, S., Chandra, T.D.: Time and Message Efficient Reliable Broadcast. In: van Leeuwen, J., Santoro, N. (eds.) WDAG 1990. LNCS, vol. 486, pp. 289–303. Springer, Heidelberg (1991)

Brief Announcement: On Reversible and Irreversible Conversions

Mitre C. Dourado[1], Lucia Draque Penso[2], Dieter Rautenbach[2], and Jayme L. Szwarcfiter[1,⋆]

[1] Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
{mitre,jayme}@nce.ufrj.br
[2] Universität Ulm, 89069 Ulm, Germany
{lucia.penso,dieter.rautenbach}@uni-ulm.de

Abstract. We study two types of iterative 0/1-vertex-labeling processes in arbitrary network graphs where in each synchronous round every vertex

- either never changes its label from 1 to 0, and changes its label from 0 to 1 if sufficiently many neighbours have label 1,
- or changes its label if sufficiently many neighbours have a different label.

In both scenarios the number of neighbours required for a change depends on individual threshold values of the vertices. Our contributions concern computational aspects related to the sets *with minimum cardinality* of vertices with initial label 1 such that during the process all vertices eventually change their label to 1 and remain with 1 as label. We establish hardness results for the general case and describe efficient algorithms for restricted instances.

Keywords: Dynamic monopoly; conversion; consensus; irreversible threshold process; local majority process; iterative polling process; fault propagation; self-stabilization.

1 Introduction

We study iterative 0/1-vertex-labeling processes on finite, simple, and undirected graphs, which model distributed computations in networks whose communication model is synchronous message-passing. Such processes correspond to conversion or consensus problems and occured in a variety of distinct areas such as social influence, neural networks, cellular automata, percolation, marketing strategies, and especially in distributed computing [1–9].

Formally, given a graph G, a threshold function $f : V(G) \to \mathbb{Z}$, and an initial labeling $c_1 : V(G) \to \{0, 1\}$, the IRR$_f$*-process of* c_1 *on* G is the sequence $\mathcal{P} = (c_1, c_2, \ldots)$ of labelings $c_t : V(G) \to \{0, 1\}$ such that for every $v \in V(G)$ and every $t \in \mathbb{N}$, we have $c_{t+1}(v) = 1$ iff either $c_t(v) = 1$ or there are $f(v)$ neighbours

⋆ The authors acknowledge partial support by the CAPES/DAAD Probral project "Cycles, Convexity, and Searching in Graphs".

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 395–397, 2010.

u of v with $c_t(u) = 1$. Similarly, given G, f, and c_1 as above, the R_f*-process of* c_1 *on* G is the sequence $\mathcal{P} = (c_1, c_2, \ldots)$ of labelings $c_t : V(G) \to \{0, 1\}$ such that for every $v \in V(G)$ and every $t \in \mathbb{N}$, we have $c_{t+1}(v) \neq c_t(v)$ iff there are $f(v)$ neighbours u of v with $c_t(u) \neq c_t(v)$. A process $\mathcal{P} = (c_1, c_2, \ldots)$ is said to *converge to* 1 if eventually all vertices have label 1, i.e. there is some $t_0 \in \mathbb{N}$ such that $c_t(v) = 1$ for every $v \in V(G)$ and every $t \in \mathbb{N}$ with $t \geq t_0$. The parameter $\mathrm{irr}_f(G)$ ($\mathrm{r}_f(G)$, resp.) is the minimum number of vertices with label 1 in an initial labeling c_1 such that the IRR_f-process (R_f-process, resp.) of c_1 on G converges to 1.

IRR_f-processes are *irreversible* in the sense that vertices whose label is 1 will never change their label to 0. An IRR_f-process on a graph G models the spread of something like a virus, a fashion, or a permanent fault. The *irreversible* k*-threshold processes* considered by Dreyer and Roberts [3] coincide exactly with IRR_k-processes where the index k means that the threshold function f is constant k. R_f-processes are *reversible* in the sense that vertices may change their label several times. An R_f-process on a graph G models iterative voting or updating mechanisms related to consensus problems. The *local majority processes* considered by Mustafa and Pekeč [6] coincide exactly with R_f-processes where $f(v) = \left\lfloor \frac{d_G(v)}{2} \right\rfloor + 1$ for every vertex $v \in V(G)$. Certain IRR_f-processes, R_f-processes, and many of their natural variants were proposed under names such as *local majority processes* or *iterative polling processes* in distributed computing. The two parameters $\mathrm{irr}_f(G)$ and $\mathrm{r}_f(G)$ are closely related to *dynamic monopolies* or *dynamos* for short, and to *catastrophic fault patterns*.

Our contributions concern computational aspects of these two parameters.

2 IRR_f-Processes

First, we consider the decision problem IRR_k-Conversion Set, which, for a given graph G and a given integer $c \geq 0$, consists of deciding "$\mathrm{irr}_k(G) \leq c$".

Dreyer and Roberts [3] prove the NP-completeness of IRR_k-Conversion Set for every $k \geq 3$ and they explicitely mention the complexity of IRR_2-Conversion Set as an open problem, which is solved by our following result.

Theorem 1. IRR_2-Conversion Set *is NP-complete.*

Next, we consider the optimization problem Minimum IRR-Conversion Set, which, for a given graph G and a given function $f : V(G) \to \mathbb{Z}$, consists of determining an initial labeling $c_1 : V(G) \to \{0, 1\}$ with $c_1(V(G)) = \mathrm{irr}_f(G)$ such that the IRR_f-process of c_1 on G converges to 1.

In [3] Dreyer and Roberts present an algorithm and claim that it determines $\mathrm{irr}_2(G)$ for (rooted) trees. Unfortunately, if the tree T_k arises by joining a vertex r to a vertex of degree 2 in each of k disjoint paths of order 4, then choosing r as the root of T_k, the algorithm of Dreyer and Roberts returns 4 if $k = 1$ and $3k$ if $k \geq 2$ while $\mathrm{irr}_2(T_k) = 2k + 1$, i.e. it does not work correctly.

We develop a reduction principle, which leads to the following results.

Theorem 2. *For $n_0 \in \mathbb{N}$, there is a linear time algorithm that solves* Minimum IRR-Conversion Set *restricted to instances (G, f) such that all blocks of G have order at most n_0.*

Theorem 3. *There is a quadratic time algorithm that solves* Minimum IRR-Conversion Set *restricted to instances (G, f) such that G is chordal and $f(v) \leq 2$ for every $v \in V(G)$.*

3 R_f-Processes

R_f-processes are much harder to understand than IRR_f-processes. It is an open problem to efficiently determine $\mathrm{r}_2(T)$ even for trees T. We present a best possible bound.

Proposition 1. *If T is a tree of order n with l leaves, then $\mathrm{r}_2(T) \leq \frac{n+l}{2}$.*

Next, we consider the decision problem R_k-Conversion Set, which, for a given graph G and a given integer $c \geq 0$, consists of deciding "$\mathrm{r}_k(G) \leq c$".

It is unknown whether R_k-Conversion Set lies in NP and we establish the following hardness result.

Theorem 4. R_2-Conversion Set *is NP-hard.*

Finally, we consider the problem of computing $\mathrm{r}_f(G)$ for certain paths and cycles.

Theorem 5. *There is a polynomial time algorithm that determines $\mathrm{r}_f(G)$ restricted to instances (G, f) such that G is a path or a cycle and $f(v) = 1$ for some vertex v of G implies that v has a neighbour u with $f(u) = 1$.*

References

1. Balister, P., Bollobás, B., Johnson, J.R., Walters, M.: Random majority percolation. Random Struct. Algorithms 36, 315–340 (2010)
2. Bermond, J.-C., Bond, J., Peleg, D., Perennes, S.: The power of small coalitions in graphs. Discrete Appl. Math. 127, 399–414 (2003)
3. Dreyer Jr., P.A., Roberts, F.S.: Irreversible k-threshold processes: Graph-theoretical threshold models of the spread of disease and of opinion. Discrete Appl. Math. 157, 1615–1627 (2009)
4. Hassin, Y., Peleg, D.: Distributed probabilistic polling and applications to proportionate agreement. Inf. Comput. 171, 248–268 (2001)
5. Kempe, D., Kleinberg, J., Tardos, É.: Maximizing the spread of influence through a social network. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 137–146 (2003)
6. Mustafa, N.H., Pekeč, A.: Listen to your neighbors: How (not) to reach a consensus. SIAM J. Discrete. Math. 17, 634–660 (2004)
7. Peleg, D.: Local majorities, coalitions and monopolies in graphs: A review. Theor. Comput. Sci. 282, 231–257 (2002)
8. Peleg, D.: Size bounds for dynamic monopolies. Discrete Appl. Math. 86, 263–273 (1998)
9. Poljak, S., Turzík, D.: On pre-periods of discrete influence systems. Discrete Appl. Math. 13, 33–39 (1986)

Brief Announcement: A Decentralized Algorithm for Distributed Trigger Counting

Venkatesan T. Chakaravarthy[1], Anamitra R. Choudhury[1], Vijay K. Garg[2], and Yogish Sabharwal[1]

[1] IBM Research - India, New Delhi
[2] University of Texas at Austin
{vechakra,anamchou,ysabharwal}@in.ibm.com, garg@ece.utexas.edu

Abstract. Consider a distributed system with n processors, in which each processor receives some triggers from an external source. The distributed trigger counting problem is to raise an alert and report to a user when the number of triggers received by the system reaches w, where w is a user-specified input. The problem has applications in monitoring, global snapshots, synchronizers and other distributed settings. The main result of the paper is a decentralized and randomized algorithm with expected message complexity $O(n \log n \log w)$. Moreover, every processor in this algorithm receives no more than $O(\log n \log w)$ messages with high probability.

1 Introduction

Consider a distributed system with n processors, in which each processor receives some triggers from an external source. The distributed trigger counting (DTC) problem is to raise an alert and report to a user when the number of triggers received by the system reaches w, where w is a user specified input. The sequence of processors receiving the w triggers is not known apriori to the system. Moreover, the number of triggers received by each processor is also not known. We are interested in designing distributed algorithms for the DTC problem that are communication efficient and are also decentralized. The DTC problem arises in applications such as distributed monitoring and global snapshots. We refer to [3] and [2] for a discussion of these two aspects, respectively. The DTC problem is also related to the distributed resource controller problem (see e.g. [1]).

Our goal is to design a distributed algorithm for the DTC problem that is communication efficient and decentralized. We use the following two natural parameters that measure these two important aspects:

- The *message complexity*, which is defined to be the number of messages exchanged between the processors.
- The MaxLoad, which is defined to be the maximum number of messages received by any processor in the system.

Garg et al. [2] studied the DTC problem and presented two algorithms: a centralized algorithm and a tree-based algorithm. The centralized algorithm has

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 398–400, 2010.

message complexity $O(n \log w)$. However, the MaxLoad of this algorithm can be as high as $\Omega(n \log w)$. The tree-based algorithm has message complexity $O(n \log n \log w)$. This algorithm is more decentralized in a heuristic sense, but its MaxLoad can be as high as $O(n \log n \log w)$, in the worst case. They also proved a lowerbound on the message complexity. They showed that any deterministic algorithm for the DTC problem must have message complexity $\Omega(n \log(w/n))$. So, the message complexity of the centralized algorithm is optimal asymptotically. However, this algorithm has MaxLoad as high as the message complexity.

We consider a general distributed system where any processor can communicate with any other processor. We assume an asynchronous model of computation and messages. We assume that the messages are guaranteed to be delivered but there is no fixed upper bound on the message arrival time.

Our main result is a decentralized randomized algorithm called LayeredRand that is efficient in terms of both the message complexity and MaxLoad. Its message complexity is $O(n \log n \log w)$. Moreover, with high probability, its MaxLoad is $O(\log n \log w)$. The message complexity of our algorithm is the same as that of the tree based algorithm of Garg et al. [2]. However, the MaxLoad of our algorithm is significantly better than both their tree based and centralized algorithms. It is important to minimize MaxLoad for many applications. For example, in sensor networks where the message processing may consume limited power available at the node, a high MaxLoad may reduce the lifetime of a node.

Our main result is formally stated next. For $1 \leq i \leq w$, the external source delivers the ith trigger to some processor x_i. We call the sequence $x_1, x_2, \ldots, x_w$ as a *trigger pattern.*

Theorem 1. *Fix any trigger pattern. The message complexity of the* LayeredRand *algorithm is* $O(n \log n \log w)$. *Furthermore, there exist constants* c *and* $d \geq 1$ *such that* $\Pr[\text{MaxLoad} \geq c \log n \log w] \leq 1/n^d$. *The above bounds hold for* any *trigger pattern, even if fixed by an adversary.*

2 LayeredRand Algorithm

For the sake of simplicity, we assume that $n = 2^L - 1$, for some integer L. The n processors are arranged in L layers numbered 0 through $L-1$. For $0 \leq \ell < L$, layer ℓ consists of 2^ℓ processors. Layer 0 consists of a single processor, which we refer to as the *root.* Layer $L-1$ is called the *leaf layer.* Only processors occupying adjacent layers will communicate with each other.

The algorithm proceeds in multiple rounds. In the beginning of each round, the system needs to know how many triggers are yet to be received. This can be computed by keeping track of the total number of triggers received in all the previous rounds and subtracting this quantity from w. Let the term *initial value of a round* mean the number of triggers yet to be received at the beginning of the round. We use a variable $\hat{w}$ to store the initial value of any round. In the first round, we set $\hat{w} = w$, since all the w triggers are yet to be received.

We next describe the procedure followed in a single round. Let $\hat{w}$ denote the initial value of this round. For each $1 \leq \ell < L$, we compute $\tau(\ell)$ for the layer ℓ:

$$\tau(\ell) = \left\lceil \frac{\hat{w}}{4 \cdot 2^{\ell} \cdot \log(n+1)} \right\rceil .$$

Each processor x maintains a counter $C(x)$, which is used to keep track of some of the triggers received by x and other processors occupying the layers below of that of x. The exact semantics $C(x)$ will become clear shortly. The counter is reset to zero in the beginning of the round.

Consider any non-root processor x occupying a level ℓ. Whenever x receives a trigger, it will increment $C(x)$ by one. If $C(x)$ reaches the threshold $\tau(\ell)$, x chooses a processor y occupying level $\ell - 1$ *uniformly at random* and sends a message to y. We refer to such a message as a *coin*. Upon receiving the coin, y updates $C(y)$ by adding $\tau(\ell)$ to $C(y)$. Intuitively, receipt of a coin by y means that y has evidence that some processors below the layer $\ell - 1$ have received $\tau(\ell - 1)$ triggers. After the update, if $C(y) \geq \tau(\ell - 1)$, y will pick a processor z occupying level $\ell - 2$ uniformly at random and send a coin to z. Then, y updates $C(y) = C(y) - \tau(\ell - 1)$. Processor z handles the coin similarly.

The behavior of the root is similar to that of the other processors, except that it does not send coins to anybody above. The root processor r also maintains a counter $C(r)$. Whenever it receives a trigger from the external source, it increments $C(r)$ by one. If it receives a coin from level 1, it updates $C(r) = C(r) + \tau(1)$.

The crucial activity of the root is to initiate an *end-of-round* procedure. When $C(r)$ reaches $\lceil \hat{w}/2 \rceil$ (i.e., when $C(r) \geq \lceil \hat{w}/2 \rceil$), the root declares *end-of-round*. At this stage the root needs to get a count of the total number of triggers received by all the processors in this round. Let this count be w'. Notice that the sum of $C(x)$ over all the processors yields the value w'. Using binary-tree formation over the processors, this task can be accomplished in such a way that each processor receives at most a constant number of messages. At the end of any round, if the newly computed $\hat{w}$ is zero, we know that all the w triggers have been received. So, the root can raise an alert to the user and the algorithm is terminated.

It can be shown that the algorithm correctly raises an alert if and only if it receives w triggers. We now present a brief sketch of the analysis. As $\hat{w}$ falls by a factor of two in each round, the number of rounds is $O(\log w)$. The number of messages exachanged in each round can be shown to be $O(n \log n)$. Hence, the message complexity is $O(n \log n \log w)$. It can be shown that for any node, the expected number of messages received is $O(\log n \log w)$. Furthermore, using Chernoff bounds, we can show that with high probability the MaxLoad is $O(\log n \log w)$. This establishes Theorem 1.

References

1. Emek, Y., Korman, A.: Brief announcement: New bounds for the controller problem. In: PODC '09 (2009)
2. Garg, R., Garg, V.K., Sabharwal, Y.: Scalable algorithms for global snapshots in distributed systems. In: 20th Int. Conf. on Supercomputing, ICS (2006)
3. Keralapura, R., Cormode, G., Ramamirtham, J.: Communication-efficient distributed monitoring of thresholded counts. In: SIGMOD Conference (2006)

Brief Announcement: Flash-Log – A High Throughput Log

Mahesh Balakrishnan, Philip A. Bernstein,
Dahlia Malkhi, Vijayan Prabhakaran, and Colin Reid

Microsoft Corporation

1 Introduction

Modern storage solutions, such as non-volatile solid-state devices, offer unprecedented speed of access over high-bandwidth interconnects. An array of flash memory chips attached directly to a 1-10 GB fiber switch can support up to 100K page writes per second. While no single host can drive such throughput, the combined power of a large group of clients, accessing the shared storage over a common interconnect, can utilize the system at full capacity.

One useful structure for shared storage is a totally-ordered log. Such a log can be used to store updates from transactions that originate at different clients. This could be preferable to server-local logs that are used in shared-storage database systems, as in Oracle RAC and Rdb/VMS, to avoid the need to merge logs at recovery time. It could also be used as a log-structured database, as in the Hyder system [1]. A log-structured database is particularly attractive for non-volatile solid-state storage, where log-structuring improves write performance and automates wear-leveling.

These settings motivate the design of a shared totally-ordered log named Flash-Log. Flash-Log supports an `appendlog`(entry) API, which provides clients with a totally ordered, durable shared log. The log is stored on network-attached storage units, which we call segments. For fault tolerance and bandwidth, the log is stored on more than one segment. That is, each log entry is spread across the segments using replication or erasure coding.

In addition to the utilization benefit of giving multiple clients access to shared storage, there is also a scale-out benefit. To grow the system, one simply adds more clients, up to the capacity and throughput limit of the shared storage. This is easier than in a system with direct attached storage, where data needs to be partitioned and/or replicated across clients and the workload needs to be partitioned so that each request is directed to a client that has the required data.

Since clients can independently write to segments, coordination is needed to ensure they agree on a total order of the entries that are appended to the log, despite failures and message delays. This constitutes a classical scenario handled by state machine replication. However, Flash-Log deviates from classical solutions in that it uses shared storage as participants in the process of reaching a decision about the total order. The core difficulty is to implement certain functionalities using segments that support only simple operations that are appropriate for implementation in a device controller. This is particularly challenging when dealing

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 401–403, 2010.

with autonomous configuration management: Our design automatically migrates to new storage devices if a failure occurs or a segment fills up.

One might hope for an existing replication recipe for these settings. Unfortunately, previous work in these settings laid an insufficient algorithmic foundation. The Disk Paxos (DP) algorithm of Gafni and Lamport [3] and the Active Disk Paxos work of Chockler and Malkhi [2] both build on the foundations of Paxos. Hence, they build reliable storage using $2N-1$ ($2F+1$) replicas in order to tolerate failure of any $F = N-1$. In today's large storage settings, these solutions are too costly. Instead, with Flash-Log, we deploy only N disks, which is suitable for the storage setting because storage servers may be huge and replicating the actual service state is costly. When we lose a disk, we immediately switch to a new set of segments, and start a background recovery process to prevent a double failure from causing data loss.

Additionally, in DP, each client is designated an exclusive block on disk to announce when it wants to become a leader; a wannabe leader must read the blocks allocated for the entire universe of clients to discover if another leader has been activated. This method not only uses $2N-1$ disks for availability, but pre-reserves one disk block for exclusive write per client per consensus decision on each of the disks, which can hardly be practical. Additionally, in DP everyone must know the entire set of clients from the outset. Some of these limitations are alleviated in Active DP, but to pipeline updates effectively, Active DP requires clients to funnel updates through a leader. In contrast, with Flash-Log the set of clients is completely dynamic. Multiple clients contending for the next append to the log reserve a unique slot and then write it directly to network attached storage. This makes Flash-Log suitable for sustaining high-throughput load.

2 Append-Log Steady State

The log provides applications running on client nodes with an `appendlog`(entry) interface. An entry has a fixed size, which fits precisely in one disk page. The append-log operations are made atomic in the face of failures by committing provisional append-ordering information onto a log. Two protocols were crafted for use in Flash-Log, both of which are novel variants of Vertical Paxos [4].

In the first variant, Flash-Log1, the configuration consists of a set of segments, which clients (potentially numerous) access directly over an inter-connect. Note that in this architecture, no single process is a bottleneck in the critical path of normal operation. The Flash-Log1 segment-set is comprised of several components: an **edge segment**, a set of L **log segments**, and a set of N **end-write segments**, all required to provide an append operation. The first phase of append-log writes the log entry to the edge and set of L log-segments. This stores the entry and simultaneously reserves an offset for it, the one returned from the append to the edge segment. The second phase of append-log commits the reserved offset by writing it to the N end-write segments. Figure 2 (left) depicts the steady-state interaction pattern of Flash-Log1.

In the second variant, Flash-Log2, the configuration of each epoch consists of a set of N segments and a designated leader process running on some compute

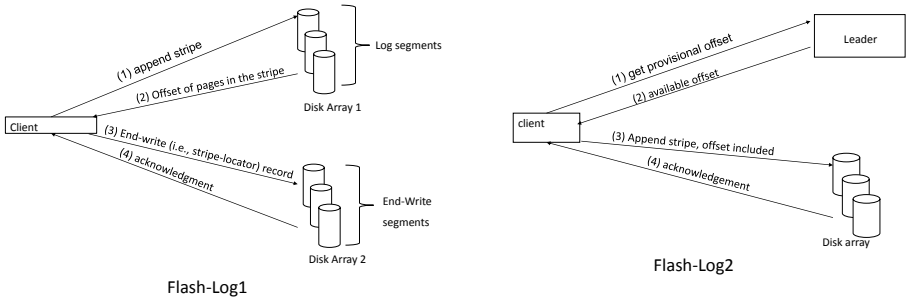


Fig. 1. Steady-State Communication Flow in Flash-Log

node. This variant works with standard read/write operations on commodity disks, but requires a leader process. Although this variant is not purely data-centric, the leader role is so light that it can likely handle requests at interconnect speed. The first phase gets a reserved offset from the leader; the second phase commits the entry with the offset embedded in it onto the N segments. Figure 2 (right) depicts the steady-state interaction pattern of Flash-Log2.

3 Auto Reconfiguration

If a fault occurs or a segment fills up, Flash-Log automatically migrates to new storage devices. Whereas the steady-state protocol is fairly straightforward, reconfiguration requires care. With correct algorithmic foundation, we can correctly handle failures, even complex cascading scenarios. The Flash-Log reconfiguration protocol borrows from the Vertical Paxos (VP) protocol [4], but modifies it, using passive storage devices in the role of various protocol participants.

References

1. Bernstein, P.A., Reid, C.W.: Scaling out without partitioning. In: 13th Workshop on High Performance Transaction Systems, HPTS (2009)
2. Chockler, G., Malkhi, D.: Active disk paxos with infinitely many processes. In: PODC '02: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 78–87. ACM, New York (2002)
3. Gafni, E., Lamport, L.: Disk paxos. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 330–344. Springer, Heidelberg (2000)
4. Lamport, L., Malkhi, D., Zhou, L.: Vertical paxos and primary-backup replication. In: PODC '09: Proceedings of the 28th ACM Symposium on Principles of distributed computing, pp. 312–313. ACM, New York (2009)

Brief Announcement: New Bounds for Partially Synchronous Set Agreement

Dan Alistarh[1], Seth Gilbert[1], Rachid Guerraoui[1], and Corentin Travers[2]

[1] Swiss Federal Institute of Technology, Lausanne, Switzerland
[2] Technion, Haifa, Israel

Set agreement [4] is a fundamental problem in distributed computing, in which processes collectively choose a small subset of values from a larger set of proposals. Set agreement has been extensively studied in both synchronous and asynchronous systems [10,11,3,5,8,9]. Real world distributed systems, however, are neither purely synchronous nor purely asynchronous. To describe such a system, Dwork et al. [6] introduced the idea of *partial synchrony*. They assume for every execution some (unknown) time GST (*global stabilization time*), after which the system is synchronous. In a recent paper [1,2], we study the complexity of set agreement in the context of partially synchronous systems, determining the minimum-sized window of synchrony in which set agreement can be solved. We show that at least $\lfloor \frac{t}{k} \rfloor + 2$ synchronous rounds are required for k-set agreement, where $t < n$ is the number of crashes, and k is the agreement parameter of the set agreement task. We then introduce an algorithm that terminates in any window of synchrony of size at least $\lfloor \frac{t}{k} \rfloor + 4$ rounds. Together, these results tightly bound the inherent price of tolerating some asynchrony.

Lower Bound By Reduction. The technique for deriving the lower bound is an important contribution in end of itself, as it provides new insights into the complexity of set agreement. Instead of relying on topology, as is typically required for set agreement lower bounds, we derive our result by reducing the feasibility of asynchronous set agreement to the problem of solving set agreement in a window of size $\lfloor \frac{t}{k} \rfloor + 1$. Since asynchronous set agreement is known to be impossible, this reduction implies that at least $\lfloor \frac{t}{k} \rfloor + 2$ synchronous rounds are required for k-set agreement.

This technique can be viewed as a generalization of the simulation from [7], moving from synchronous systems to cover the spectrum of partially synchronous ones. There are two new key observations. First, when the simulation is run for an epoch of length $\lfloor \frac{t}{k} \rfloor + 1$ rounds, we show that either some simulator sees a window of synchrony of size $\lfloor \frac{t}{k} \rfloor + 1$ rounds, or some simulator fails. Second, we observe that these epochs of length $\lfloor \frac{t}{k} \rfloor + 1$ can be repeated until some simulator sees a synchronous window of $\lfloor \frac{t}{k} \rfloor + 1$ rounds. From this we conclude that we have successfully simulated a set agreement protocol, resulting in the desired reduction.

Early Deciding Synchronous Set Agreement. Our technique turns out to be of more general interest as we can re-derive and extend existing lower bounds for synchronous *early deciding* set agreement. It has been previously shown [8,9], using sophisticated techniques, that even in an execution with $f < t$ failures, some process cannot decide prior to round $\lfloor \frac{f}{k} \rfloor + 2$. We re-derive both lower

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 404–405, 2010.

bounds in a simpler and more general manner. Of note, both lower bounds are corollaries of a single theorem that relates the number of processes which decide early with the worst-case round complexity of an algorithm. Basically, we show that if d processes decide by round $\lfloor \frac{f}{k} \rfloor + 1$ in executions with at most f failures, then in the worst-case, some process takes at least time $\lfloor \frac{t}{k} \rfloor + E(\cdot) + 1$ to decide (where E is a function of t, k and d).

Upper Bound for Partially Synchronous Agreement. We then present the first known algorithm for k-set agreement that tolerates periods of asynchrony. Our algorithm guarantees correctness, regardless of asynchrony, and terminates as soon as there is a window of synchrony of size $\lfloor \frac{t}{k} \rfloor + O(1)$. In our paper [1,2], we show synchronous round complexity of $\lfloor \frac{t}{k} \rfloor + 4$.

Implications. Our simulation technique [1,2] provides additional evidence that the impossibility of fault-tolerant asynchronous k-set agreement is a central result in distributed computing, as it implies non-trivial results in both partially synchronous and synchronous models. Second, it highlights close connections between models that have differing levels of synchrony, since it takes advantage of structural similarities between *partially synchronous* set agreement and *early deciding* set agreement to establish lower bounds in two different models of synchrony. The uncertainty regarding asynchrony (found in a partially synchronous execution) turns out to be fundamentally similar to the uncertainty regarding failures (found in an early deciding execution). Characterizing this similarity remains an intriguing open question.

References

1. Alistarh, D., Gilbert, S., Guerraoui, R., Travers, C.: Of choices, failures and asynchrony: The many faces of set agreement. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 943–953. Springer, Heidelberg (2009)
2. Alistarh, D., Gilbert, S., Guerraoui, R., Travers, C.: Of choices, failures and asynchrony: The many faces of set agreement. Algorithmica (to appear)
3. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC, pp. 91–100 (1993)
4. Chaudhuri, S.: More choices allow more faults: Set consensus problems in totally asynchronous systems. Inf. Comput. 105(1), 132–158 (1993)
5. Chaudhuri, S., Herlihy, M., Lynch, N.A., Tuttle, M.R.: A tight lower bound for k-set agreement. In: FOCS, pp. 206–215. IEEE, Los Alamitos (1993)
6. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (1988)
7. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In: PODC, pp. 143–152 (1998)
8. Gafni, E., Guerraoui, R., Pochon, B.: From a static impossibility to an adaptive lower bound: the complexity of early deciding set agreement. In: STOC, pp. 714–722 (2005)
9. Guerraoui, R., Herlihy, M., Pochon, B.: A topological treatment of early-deciding set-agreement. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 20–35. Springer, Heidelberg (2006)
10. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM 46(6), 858–923 (1999)
11. Saks, M.E., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000)

It's on Me! The Benefit of Altruism in BAR Environments

Edmund L. Wong, Joshua B. Leners, and Lorenzo Alvisi

Department of Computer Science, The University of Texas at Austin
1616 Guadalupe, Suite 2.408
Austin, TX 78701 USA
{elwong,leners,lorenzo}@cs.utexas.edu

Abstract. Cooperation, a necessity for any peer-to-peer (P2P) cooperative service, is often achieved by rewarding good behavior now with the promise of future benefits. However, in most cases, interactions with a particular peer or the service itself eventually end, resulting in some last exchange in which departing participants have no incentive to contribute. Without cooperation in the last round, cooperation in any prior round may be unachievable. In this paper, we propose leveraging altruistic participants that simply follow the protocol as given. We show that altruism is a simple, necessary, and sufficient way to incentivize cooperation in a realistic model of a cooperative service's last exchange, in which participants may be Byzantine, altruistic, or rational and network loss is explicitly considered. By focusing on network-level incentives in the last exchange, we believe our approach can be used as the cornerstone for incentivizing cooperation in any cooperative service.

Keywords: P2P, cooperative services, game theory, BAR.

1 Introduction

Establishing and maintaining cooperation between peers in decentralized services spanning multiple administrative domain (MAD) is hard [17,21]. Because participants may be selfish and withhold resources unless contributing is in their best interest, these services must provide sufficient incentives for participants to contribute. These incentive structures must of course be resilient against buggy or malicious peers; however, they must also be robust against a more subtle threat: an overabundance of good will from the unselfish peers who simply follow the protocol run by the service. It is, after all, the unselfishness of correct peers—as codified in the protocol they obediently follow—that allows selfish peers to continue receiving service without contributing their fair share. Yet, the efforts of well-meaning peers alone may be insufficient to sustain the service. Further, asking these peers to increase their contribution to make up for free-riders may backfire: even well-meaning peers, if blatantly taken advantage of, may give in to the temptation of joining the ranks of the selfish, leading in turn to more defections and to the service's collapse.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 406–420, 2010.

Although real MAD systems include a sizable fraction of correct and unselfish peers [2], their impact on the incentive structure of MAD services is not well understood. The BAR model [3] does explicitly account for these peers—they are the *altruistic* peers, who, together with the selfish *rational* peers and the potentially disruptive *Byzantine* peers, give the model its acronym—but existing BAR-tolerant systems have side-stepped the challenge of altruism by designing protocols that neither depend on nor leverage the presence of altruistic peers.[1] This paper asks the following question: *Can we leverage the good will of altruistic nodes and still motivate rational participants to cooperate?* We find that not only is altruism not antithetical to rational cooperation, but that, in a fundamental way, rational cooperation can only be achieved in the presence of altruism. To do so, we distill the issue to a rational peer's *last* opportunity to cooperate.

The last exchange. Rational peers are induced to cooperate with another peer (or, more generally, with a service) by the expectation that, if they cooperate, they will receive future benefit. However, in most cases, interaction with a particular peer or with the service itself eventually comes to an end. In this last exchange, rational peers do not have incentive to contribute, as doing so incurs cost without any future benefit. Unfortunately, rational cooperation throughout the protocol often hinges on this critical last exchange: the lack of incentive to cooperate at the end may, in a sort of reverse domino effect, demotivate rational peers from cooperating in *any* prior exchange.

Most current systems address this problem in one of three ways (or some combination of them). Some systems [3,16] assume that rational peers interact with the service forever, and thus future incentives always exist; others [13,14,15] assume rational peers deviate only if their increase in utility is above a certain threshold; others, finally, try to threaten rational peers with the possibility of losing utility if they deviate. For instance, in BAR Gossip [16], peers that do not receive the data they expect *pester* the guilty peer by repeatedly requesting the missing contribution.

Unfortunately, each of these approaches relies on somewhat unrealistic assumptions. Few relationships in life are infinite in length; worse, as we will show later in this paper, with a lossy network and the possibility of Byzantine peers it may be impossible to incentivize cooperation even in an infinite-length protocol. The real possibility of penny-pinching peers can undermine any system that assumes no deviation unless their expected gain is "large enough." Finally, threats such as pestering are effective only when they are credible: to feel threatened, a peer must believe that it will be rational for the other peer to pester. Since pestering incurs cost for the initiator as well as for the receiver, it is surprisingly hard to motivate rational peers to pester in the first place. For example, pestering in BAR Gossip is credible only under the rather implausible assumption that a peer, even when faced with enduring silence, will never give up on an

[1] Gossip-based BAR-tolerant streaming protocols [15,16] do rely on an altruistic source for seeding the stream but otherwise model the gossiping peers as either rational or Byzantine.

unresponsive peer and forever continue to attribute a peer's lack of contribution to the unreliability of the network [16].

Our contributions. We model the last-exchange problem as a finite-round game between two peers, $\mathcal{P}_1$ and $\mathcal{P}_2$; neither peer expects to interact with the other beyond this exchange. We assume $\mathcal{P}_1$ holds a contribution (*e.g.*, some information) that is of value to $\mathcal{P}_2$; however, contributing yields no expectation of further benefit for $\mathcal{P}_1$. We are interested in studying whether $\mathcal{P}_2$ can nonetheless induce a selfish $\mathcal{P}_1$ to contribute by threatening to pester it if $\mathcal{P}_1$ fails to do so. Pestering is an attractive threat because it is simple and does not require the involvement of a third party. We want to determine whether it can be made a credible threat under realistic system assumptions, unlike in BAR Gossip [16]. In each round, $\mathcal{P}_1$ is given a choice whether to contribute or not; in response, $\mathcal{P}_2$ may pester $\mathcal{P}_1$. Peers communicate through a lossy channel and therefore do not necessarily share the same view of the ongoing game. For instance, $\mathcal{P}_1$ may have contributed, but $\mathcal{P}_2$ may not have received the contribution.

We show that, without requiring implausible network assumptions or the specter of never-ending pestering, the presence of altruistic peers is both necessary and sufficient to make pestering a credible threat and motivate rational peers to contribute. In particular:

- We prove that there exists no equilibrium strategy where rational peers contribute if all peers are either rational or Byzantine—even if we allow for an infinite number of pestering rounds.
- We show that the presence of altruistic peers is sufficient to transform pestering into a credible threat. Intuitively, if rational peers have sufficiently high beliefs that they may be interacting with an altruistic peer, they are motivated to pester, making it in turn preferable for rational peers to contribute.

The fraction of altruistic peers sufficient to sustain rational contribution depends on several system parameters, including the probability of network loss, the fraction of Byzantine peers in the system, and the behavior that rational peers expect from altruistic and Byzantine peers. Exploring this space through a simulator we find that:

- Altruistic peers make rational cooperation easy to achieve under realistic conditions. In particular, we find that even if less than 10% of the population is altruistic, rational peers are incentivized to cooperate in a system where the network drops 5% of all packets and Byzantine peers make up over 50% of the remainder of the population.
- Prodigal altruistic peers do harm rational cooperation: if altruistic peers contribute every time they are pestered, then we cannot always achieve rational cooperation; when we do, it requires an implausibly high fraction of altruistic peers. This is good news: the less foolishly generous is the altruistic behavior sufficient to incentivize rational contribution, the more feasible it is to design systems with a sustainable population of altruistic peers.

– The uncertainty introduced by network loss is both a bane and a boon. On the one hand, it significantly complicates the analysis of a peer's optimal strategy because each peer does not know what the other has observed. On the other, it lowers the threshold for rational cooperation by leaving open some possibility that the other peer may be altruistic, even when the observed behavior suggests otherwise.

Organization of paper. After presenting in Section 2 the game theoretic framework used to analyze the last exchange problem, we show in Section 3 that rational cooperation is impossible in the absence of altruistic peers. We proceed to derive, in Section 4, conditions under which altruism is sufficient to elicit rational cooperation in the last exchange and, in Section 5, use simulations to study the implications of these conditions on the design of cooperative services. We discuss related work in Section 6 before concluding in Section 7. Because of space limitations, we omit proofs or provide proof sketches of most of our results; detailed proofs can be found in a companion technical report [23].

2 Formalizing the Last Exchange Problem

We consider cooperative services that can be modeled as a collection of peer-to-peer pairwise exchanges, in which two players $\mathcal{P}_1$ and $\mathcal{P}_2$ communicate over unreliable channels. In particular, we focus on the last exchange between $\mathcal{P}_1$ and $\mathcal{P}_2$; we are interested in studying under which conditions $\mathcal{P}_2$ can induce a selfish $\mathcal{P}_1$ to contribute with the threat of pestering.

We model this last exchange as a $(T+1)$-round stochastic sequential game, which is similar to a repeated game except that it allows players' payoffs to change. This flexibility is critical to model the intuition that $\mathcal{P}_2$ benefits from $\mathcal{P}_1$'s contribution only the first time $\mathcal{P}_2$ receives it. In each round, $\mathcal{P}_1$ moves first by choosing between two actions: contribute (denoted by c) or do nothing (n). $\mathcal{P}_2$ follows by choosing between two actions: pester (p) or do nothing (n). Since our analysis of the game often relies on the number of rounds remaining rather than on the round number, we think of the game as starting with round T and ending with round 0.

While doing nothing has neither cost nor benefit, $\mathcal{P}_1$ incurs a cost s_c in every round in which it contributes and a cost r_p in every round in which it is pestered; $\mathcal{P}_2$ incurs a cost r_c in every round it receives a contribution and a cost s_p in every round it pesters $\mathcal{P}_1$. A non-Byzantine $\mathcal{P}_2$ starts off being *destitute*, *i.e.*, $\mathcal{P}_2$ does not have $\mathcal{P}_1$'s contribution. A destitute $\mathcal{P}_2$ receives a one-time benefit $b_c \gg r_c + s_p$ the first time it receives $\mathcal{P}_1$'s contribution; now no longer destitute, $\mathcal{P}_2$ gains no further benefit from receiving further copies of the contribution.

Network loss, signals, and utilities. To model the unreliable channel through which $\mathcal{P}_1$ and $\mathcal{P}_2$ communicate, we adopt from game theory the concept of *private signals*: for every action a played by some player, both players privately observe some (possibly different) resulting *signal*. Specifically, let ρ, $0 < \rho < 1$, be the rate of network loss, which we assume to be common knowledge. When $\mathcal{P}_i$ plays

a, $\mathcal{P}_i$ observes a, and its peer $\mathcal{P}_{-i}$ observes a with probability $1-\rho$ and n otherwise. Thus, players do not always observe their peer's actions accurately and cannot rely on their peer accurately observing their own actions.

The sequence of signals observed by $\mathcal{P}_i$ until round t defines $\mathcal{P}_i$'s *history* h_i^t. At the beginning of the game, h_1^T is the empty sequence; h_2^T consists instead of the signal $\omega_2^{T,1}$ observed by $\mathcal{P}_2$ in round T, corresponding to $\mathcal{P}_1$'s action. In round $k < T$, h_1^k is obtained by appending to $\mathcal{P}_1$'s prior history h_1^{k+1} the next pair of signals $(\omega_1^{k+1,1}, \omega_1^{k+1,2})$ observed by $\mathcal{P}_1$: $h_1^k = (h_1^{k+1}, \omega_1^{k+1,1}, \omega_1^{k+1,2})$; similarly for $\mathcal{P}_2$, $h_2^k = (h_2^{k+1}, \omega_2^{k+1,2}, \omega_2^{k,1})$.

$\mathcal{P}_1$ and $\mathcal{P}_2$'s utilities are defined with respect to the signals they observe:

$$u_1(C, \hat{P}) = -\left(|\hat{P}|r_p + |C|s_c\right) \qquad u_2(P, \hat{C}) = H[|\hat{C}| - 1]b_c - \left(|P|s_p + |\hat{C}|r_c\right)$$

where C and $\hat{P}$ are the sets of rounds in which $\mathcal{P}_1$ respectively contributed and observed $\mathcal{P}_2$ pester; P and $\hat{C}$ are the sets of rounds in which $\mathcal{P}_2$ respectively pestered and observed $\mathcal{P}_1$ contribute; and $H[n]$ is the unit step function.[2]

Strategies, types, beliefs, and equilibrium. All players are assigned an initial *strategy*, *i.e.*, the protocol. The strategy that a player ultimately follows, however, depends on the player's *type*:

- Byzantine (**B**): These players play an arbitrary strategy.
- Altruistic (**A**): These players follow the strategy assigned to them initially.
- Rational (**R**): These players follow a strategy only if deviating unilaterally from it does not increase their utility.

As noted earlier, our game is stochastic: the payoffs of a non-Byzantine $\mathcal{P}_2$ change depending on whether $\mathcal{P}_2$ is destitute (and wants $\mathcal{P}_1$ to contribute) or not (and wants $\mathcal{P}_1$ to do nothing). For convenience, we abuse notation and introduce two additional types, **D** and $\neg$**D**, to characterize the state of a non-Byzantine player $\mathcal{P}_2$, depending on whether or not it is destitute. Note that if $\mathcal{P}_1$ contributes, a $\mathcal{P}_2$ of type **D** will observe c, and hence change to type $\neg$**D**, with probability $1-\rho$.

We are interested in finding strategies in which a rational players is not able to deviate and increase its utility *ex-ante*, *i.e.*, in expectation over its peers' types and strategies. Determining whether a given strategy is a rational player's best response, however, is complicated in our game because, while each player knows its own type, it does not know for certain the type of its peer. Instead, $\mathcal{P}_i$ starts with initial *beliefs* $\mu_i(\theta)$ representing the probabilities that $\mathcal{P}_i$ assigns to the statement that its peer $\mathcal{P}_{-i}$ is of type θ. We assume that, for all $i \in \{1, 2\}$, $\mu_i(\theta)$ equals an initial value $\mu(\theta)$, which is common knowledge, and that the beliefs of a rational $\mathcal{P}_i$'s evolve based on the history h_i^t it has observed; we use $\mu_i(\theta|h_i^t)$ to denote $\mathcal{P}_i$'s conditional beliefs.

For a given set of beliefs, a rational player's strategy $\sigma_{\mathbf{R},i}$ depends on the specific strategy that it expects its peer to adopt—which, in turn, depends on the peer's type. A rational player expects an altruistic $\mathcal{P}_i$'s strategy $\sigma_{\mathbf{A},i}$ to be

[2] $H[n] = 0$ if $n < 0$; else $H[n] = 1$.

identical to the initially assigned protocol and a rational player to follow $\sigma_{\mathbf{R},i}$ (assuming that it is a best response). If $\mathcal{P}_i$ is Byzantine, however, its strategy $\sigma_{\mathbf{B},i}$ can in principle be arbitrary, significantly complicating the task of identifying a rational player's best response ex-ante. We address this difficulty by restricting the beliefs that a rational player can hold vis-à-vis Byzantine behaviors. In particular, we assume that a rational player does not expect to be able to influence $\sigma_{\mathbf{B},i}$ through its actions, *i.e.*, a rational player expects to observe a Byzantine peer $\mathcal{P}_i$ do nothing in round t with some probability $\beta_i^t \geq \rho$ that does not depend on $\mathcal{P}_i$'s current history h_i^t. Thus, rational players, instead of considering an arbitrary $\sigma_{\mathbf{B},i}$, best-respond to this expected Byzantine strategy $\bar{\sigma}_{\mathbf{B},i}$. While this restriction sacrifices the generality of Byzantine behavior, it models the reasonable distrust that a rational player is likely to harbor towards a Byzantine peer's threats and promises. If $\mathcal{P}_i$ is not Byzantine, however, the expectation is that its strategy will depend on its observed history h_i^t; we use $\sigma_{\theta,i}(a|h_i^t)$ to denote the conditional probability that a is played by $\mathcal{P}_i$ of type θ given h_i^t.

Formally, let $\sigma_i = (\bar{\sigma}_{\mathbf{B},i}, \sigma_{\mathbf{A},i}, \sigma_{\mathbf{R},i})$ denote the strategies that a rational player expects $\mathcal{P}_i$ to adopt, depending on $\mathcal{P}_i$'s type; $\sigma = (\sigma_1, \sigma_2)$ denote the strategy profile that describes the (expected) strategies for $\mathcal{P}_1$ and $\mathcal{P}_2$; and $\mu = (\mu_1, \mu_2)$ denote the belief profile that describes the beliefs μ_1 and μ_2 held by a rational $\mathcal{P}_1$ and $\mathcal{P}_2$. We are interested in perfect Bayes equilibrium: a strategy profile and set of beliefs (σ^*, μ^*) such that for all $i \in \{1, 2\}$, $\mu_i^*(\theta|h_i^t)$ is computed using Bayes rule whenever h_i^t is reached via a signal that may be observed with positive probability; and for all histories h_i^t and strategies $\sigma'_{\mathbf{R},i}$:

$$E^{(\sigma^*_{\mathbf{R},i}, \sigma^*_{-i})}[u_i|h_i^t] \geq E^{(\sigma'_{\mathbf{R},i}, \sigma^*_{-i})}[u_i|h_i^t]$$

where $E^{(\sigma_{\mathbf{R},i}, \sigma^*_{-i})}[u_i|h_i^t]$ is a rational $\mathcal{P}_i$'s expected utility from playing $\sigma_{\mathbf{R},i}$ with beliefs μ_i^*, with both strategy and beliefs conditional on h_i^t, while its peer $\mathcal{P}_{-i}$ plays σ^*_{-i}. To lighten the already substantial notation, we will refer to $\sigma_{\mathbf{R},i}$ as σ_i when it is obvious we are referring to the rational strategy.

We assume that all players are limited to actions in the strategy space. This can be accomplished in practice if actions outside of the strategy space generate a proof of misbehavior [3,9] and if the associated punishments (*e.g.*, financial penalties) are sufficient to deter rational players. Finally, we assume that a rational $\mathcal{P}_1$ does not try to avoid pestering by severing its network connection: if losing a fraction of bandwidth from pestering is undesirable, disconnecting and losing all of it is even less desirable.

3 The Need for Altruism

Altruism is not only sufficient to incentivize cooperation, it is necessary. In this section, we assume that there is no altruism, and we show that, as a result, rational players never pester or contribute.

Theorem 1. *There exists no equilibrium in which rational $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively contribute and pester.*

Proof. (Sketch) Suppose such an equilibrium exists. Then there exists some rounds t_c and t_p such that $\mathcal{P}_1$ and $\mathcal{P}_2$ contribute and pester, respectively, with some positive probability for the last time. However, a rational $\mathcal{P}_1$ never contributes after round t_p since $\mathcal{P}_1$ incurs cost by contributing, yet there is no further threat of pestering; thus, $t_c \geq t_p$.[3] On the other hand, a rational $\mathcal{P}_2$ only pesters until round $t_c + 1$ since $\mathcal{P}_2$ incurs cost by pestering with no chance of further contribution from $\mathcal{P}_1$; thus $t_p > t_c$. Contradiction.

Theorem 1 only holds when the game lasts for a finite number of rounds. When there exists no bound on the number of rounds, a weaker, yet in practice still crippling, result holds. We summarize the main result here; the model of the infinitely-repeated game, which generalizes the finitely-repeated model, and details of the results can be found in the companion technical report [23].

Theorem 2. *In the infinitely-repeated game, suppose a non-destitute $\mathcal{P}_2$ always prefers to do nothing and rational players expect that there exists some positive fraction of Byzantine peers that either (a) when playing as $\mathcal{P}_1$, never contributes; or (a) when playing as $\mathcal{P}_2$, plays the same strategy played by a destitute rational $\mathcal{P}_2$. Then there exists no pure equilibrium in which rational $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively contribute and pester.*[4]

Proof. (Sketch) If some Byzantine $\mathcal{P}_2$ pesters as if playing the destitute rational strategy, despite $\mathcal{P}_1$'s contributions, then $\mathcal{P}_1$'s belief that $\mathcal{P}_2$ is Byzantine eventually grows arbitrarily close to 1. Similarly, if a Byzantine $\mathcal{P}_1$ never contributes despite $\mathcal{P}_2$'s incessant pestering, then $\mathcal{P}_2$ becomes increasingly certain $\mathcal{P}_1$ is Byzantine. It can be shown that a player's belief in its peer being Byzantine eventually grows sufficiently high such that the expected utility of contributing (in the first case) or pestering (in the second) is lower than that of doing nothing. By showing a bound of the number of rounds in which a rational $\mathcal{P}_1$ contributes or $\mathcal{P}_2$ pesters, it follows, using an argument similar to the finitely-repeated game, that this bound must be 0.

4 Altruism to the Rescue

We now show that altruism is sufficient to incentivize rational peers to, respectively, pester and contribute by constructing a cooperative strategy profile and proving that it is an equilibrium. We start by specifying the *altruistic strategy*:

- $\sigma_{\mathbf{A},1}$: $\mathcal{P}_1$ contributes during round T. During round $t < T$, $\mathcal{P}_1$ contributes, only if pestered, with probability $(1 - \alpha)/(1 - \rho)^2$, where α is a known parameter such that $0 < (1 - \alpha)/(1 - \rho)^2 \leq 1$.[5]
- $\sigma_{\mathbf{A},2}$: For any round $t > 0$, $\mathcal{P}_2$ pesters if and only if $\mathcal{P}_2$ is destitute.

[3] Recall that we count rounds in reverse.

[4] We can also show there exist no mixed equilibria that put positive probability on a finite number of histories. We leave other mixed strategies to future work.

[5] Hence, if $\mathcal{P}_2$ pesters an altruistic $\mathcal{P}_1$ during round t, $\mathcal{P}_2$ expects to observe a contribution in round $t - 1$ with probability $1 - \alpha$.

In practice, all players are initially given the altruistic strategy. Although we cannot guarantee that a rational $\mathcal{P}_1$ will follow $\sigma_{\mathbf{A},1}$, we can (and will) prove that, under the expectation that its peer $\mathcal{P}_{-i}$ of type θ plays $\sigma_{\theta,-i}$,[6] a rational $\mathcal{P}_i$ will play the following *rational strategy*:

- $\sigma_{\mathbf{R},1}$: During round t, $\mathcal{P}_1$ contributes if and only if $t > \lceil s_c/((1-\rho)^2 r_p) \rceil - 1$ (*i.e.*, if being pestered is sufficiently expensive to overcome the cost of contributing), $\mathcal{P}_1$ observes pestering (for $t < T$), and $\mathcal{P}_1$'s belief that $\mathcal{P}_2$ is destitute exceeds some threshold $\bar{\mu}_1^t$.
- $\sigma_{\mathbf{R},2}$: Same as $\sigma_{\mathbf{A},2}$.

In a perfect Bayes equilibrium, whether a rational player deviates or not depends on its beliefs for all histories, both those on and off the equilibrium path. In our desired equilibrium, almost every history has some positive probability of being observed: a rational $\mathcal{P}_2$ expects that an altruistic $\mathcal{P}_1$ contributes with positive probability (if $\mathcal{P}_2$ pestered); destitute $\mathcal{P}_2$ always pester; and, as a result of network loss, doing nothing is always observable with positive probability from either player. Thus, for most histories, Bayes rule can be applied to calculate a rational player's beliefs. For those histories that are not observed with positive probability, beliefs can be assigned which support the rational strategy as an equilibrium strategy. The details, omitted here for lack of space, can be found in the companion technical report [23].

Generally, there may exist multiple strategies that result in a cooperative equilibrium. We believe that our rational strategy represents a sensible design point: incentivizing a rational $\mathcal{P}_1$ to contribute in every round would require $\mathcal{P}_1$ to start with an unrealistically low belief in $\mathcal{P}_2$ being Byzantine. Fortunately, this is unnecessary: we show in Section 5 that the rational strategy results in $\mathcal{P}_1$ often contributing multiple times.

4.1 When Does a Rational $\mathcal{P}_2$ Pester?

Intuitively, $\mathcal{P}_2$ pesters only if it is destitute *and* it believes that $\mathcal{P}_1$ is sufficiently altruistic (and thus willing to contribute, even in the final rounds).

Lemma 3. *If a rational $\mathcal{P}_2$ has received a contribution, $\mathcal{P}_2$ does nothing.*

Proof. (Sketch) If $\mathcal{P}_2$ already has the contribution, $\mathcal{P}_2$ receives no further benefit from receiving another contribution. In fact, pestering and receiving another contribution only incurs cost.

Theorem 4. *A rational, destitute $\mathcal{P}_2$ pesters in round $t > 0$ following some history h_2^t if its belief $\mu_2(\boldsymbol{A}|h_2^t)$ that $\mathcal{P}_1$ is altruistic satisfies the following condition:*

$$\mu_2(\boldsymbol{A}|h_2^t) > \frac{s_p}{\alpha^{t-1}(1-\alpha)(b_c - r_c) + (1-\alpha^{t-1})s_p} \tag{1}$$

[6] Thus, all our lemmas and theorems should be prefaced by "In our cooperative equilibrium...".

Proof. (Extended sketch) By contradiction. Assume $\mathcal{P}_2$ prefers to do nothing despite condition (1): there exists some strategy σ_2^n in which $\mathcal{P}_2$ does nothing in round t despite having beliefs which satisfy condition (1). Construct an alternate strategy σ_2^p in which: $\mathcal{P}_2$ pesters following h_2^t; if $\mathcal{P}_2$ receives a contribution in round $t-1$, $\mathcal{P}_2$ does nothing for the remainder of the game; otherwise, σ_2^p and σ_2^n are identical for the remaining rounds.

Consider $\mathcal{P}_2$'s difference in expected utility between playing σ_2^n and σ_2^p. There are three cases. If $\mathcal{P}_1$ is Byzantine, the expected difference in utility between σ_2^n and σ_2^p is s_p. If $\mathcal{P}_1$ is rational, it can be shown that $\mathcal{P}_2$ has a better chance of receiving a contribution in the future if $\mathcal{P}_2$ pesters now (versus doing nothing); hence, if $\mathcal{P}_2$ is destitute starting from h_2^t, then the expected difference in utility between σ_2^n and σ_2^p is at most s_p. Finally, if $\mathcal{P}_1$ is altruistic, then the expected utility, starting from $\mathcal{P}_2$'s turn in round $t-1$, of playing σ_2^n or σ_2^p is the same; let $V(\mathbf{A}, t-1)$ represent this utility. Thus, the expected difference in utility between σ_2^n and σ_2^p when facing an altruistic $\mathcal{P}_1$ is then

$$s_p - (1-\alpha)(b_c - r_c - V(\mathbf{A}, t-1))$$

It can be shown that pestering an altruistic $\mathcal{P}_1$ until $\mathcal{P}_2$ gets the contribution or $t=0$ is in $\mathcal{P}_2$'s best interest; thus, for $i < t$, $V(\mathbf{A}, i) \leq -s_p + (1-\alpha)(b_c - r_c) + \alpha V(\mathbf{A}, i-1)$, where $V(\mathbf{A}, 0) = 0$. Solving the recursion and using condition (1) we find that the expected difference in utility between σ_2^n and σ_2^p is at most

$$s_p - \mu_2(\mathbf{A}|h_2^t)(\alpha^{t-1}(1-\alpha)(b_c - r_c) + (1-\alpha^{t-1})s_p) < 0$$

and thus $\mathcal{P}_2$ prefers to pester. Contradiction.

4.2 When Does a Rational $\mathcal{P}_1$ Contribute?

It is obvious that $\mathcal{P}_1$ never contributes when the threat of pestering does not offset the cost of contributing.

Lemma 5. *$\mathcal{P}_1$ does nothing for rounds $t \leq \tau$, where*

$$\tau = \left\lceil \frac{1}{(1-\rho)^2} \frac{s_c}{r_p} \right\rceil - 1 \tag{2}$$

The next two results limit when $\mathcal{P}_1$ contributes. Even after it contributes, if $\mathcal{P}_1$ is unsure whether or not $\mathcal{P}_2$ is still destitute, a combination of several factors (a highly lossy network that may have dropped the contribution, a high cost for being pestered, enough rounds to make pestering a sufficiently costly threat) may induce $\mathcal{P}_1$ to contribute again, without being pestered.

Lemma 6 gives a condition under which $\mathcal{P}_1$ contributes only if a non-Byzantine $\mathcal{P}_2$ is known to be destitute. It follows in Theorem 7 that, after the first round, $\mathcal{P}_1$ contributes only if pestered in the prior round.

Lemma 6. *Let $t < T$ be the current round, where*

$$T < \frac{1-\rho+\rho^2}{\rho^2(1-\rho)^2}\frac{s_c}{r_p} \tag{3}$$

If $\mathcal{P}_1$ contributed in the past and has not been pestered since, then $\mathcal{P}_1$ does nothing in round t.

Theorem 7. *Let $t < T$ be the current round, and suppose that $\mathcal{P}_1$ observed nothing from $\mathcal{P}_2$ in round $t+1$. Then $\mathcal{P}_1$ does nothing in round t if (3) holds.*

Proof. (Sketch) By contradiction. Consider some round t in which $\mathcal{P}_1$ contributes despite having done nothing and observed $\mathcal{P}_2$ do nothing in round $t+1$ (if $\mathcal{P}_1$ contributed, Lemma 6 holds). It can be shown that since (1) $\mathcal{P}_1$'s belief in $\mathcal{P}_2$ being destitute is strictly non-decreasing when observing $\mathcal{P}_2$ do nothing and (2) the number of expected pesters decreases with the number of remaining rounds, $\mathcal{P}_1$ increases its utility by contributing in round $t+1$ instead of round t. Contradiction.

We now consider the conditions under which $\mathcal{P}_1$ actually contributes. In every round, $\mathcal{P}_1$ must make a choice:

- Pay the cost of contributing now (s_c), hoping to stop a non-Byzantine $\mathcal{P}_2$ from pestering in the future. The savings are a function of the remaining rounds and the beliefs about $\mathcal{P}_2$.
- Delay contributing, at the risk of being pestered (with cost at most $(1-\rho)r_p$), hoping to glean more about $\mathcal{P}_2$'s type.

Procrastination has its lure. Since we are considering strategies where a non-Byzantine $\mathcal{P}_2$ always pesters (minus the last round) whereas a Byzantine $\mathcal{P}_2$ may not, every additional signal can drastically affect $\mathcal{P}_1$'s expected utility and possibly save $\mathcal{P}_1$ the cost of contributing. Moreover, doing nothing now does not preclude $\mathcal{P}_1$ from contributing in the future. Yet, we find that if $\mathcal{P}_1$ has sufficiently strong belief that $\mathcal{P}_2$ is destitute, procrastination is something best put off until tomorrow: for every round sufficiently removed from the end of the game, there exists a belief threshold above which contributing yields a higher expected utility for $\mathcal{P}_1$.

Theorem 8. *Let h_1^t be the history of $\mathcal{P}_1$ in round t, where $\tau < t \leq T$ and τ is defined by condition (2). A rational $\mathcal{P}_1$ contributes if its belief that $\mathcal{P}_2$ is destitute exceeds some threshold $\bar{\mu}_1^t \leq 1$. In particular, if $\mu_1(\boldsymbol{D}|h_1^t) \geq \bar{\mu}_1^t$, $\mathcal{P}_1$ contributes; otherwise, $\mathcal{P}_1$ does nothing.*

Proof. (Extended sketch) By induction on t. The base case, $t = \tau + 1$, is simple: since $\mathcal{P}_1$ never contributes after round $\tau + 1$, we can calculate the threshold at which $\mathcal{P}_1$ prefers to contribute. For the inductive step, we assume the theorem holds for all t, $\tau < t \leq t_0$; we prove $t = t_0 + 1$ by contradiction. If a threshold does

not exist, there must be some belief $\mu^*(\mathbf{D}|h_1^t)$ in which $\mathcal{P}_1$ prefers to contribute and higher beliefs in which $\mathcal{P}_1$ prefers to do nothing. By the inductive hypothesis, it can be shown that we can always find a belief $\mu_1'(\mathbf{D}|h_{1'}^t)$, arbitrarily close to $\mu^*(\mathbf{D}|h_1^t)$ from above, such that:

- $\mathcal{P}_1$ does nothing in round t; and
- $\mathcal{P}_1$ plays the same actions in subsequent rounds after observing the same non-empty sequence of signals as if $\mathcal{P}_1$ had started with belief $\mu^*(\mathbf{D}|h_1^t)$.

It follows that $\mathcal{P}_1$'s expected utility of playing action a_1^t followed by the threshold strategy with either belief $\mu_1^*(\theta|h_1^t)$ or $\mu_1'(\theta|h_{1'}^t)$ must be equal; let $V(a_1^t, \theta)$ be this expected continuation utility. By Lemmas 3 and 6, we know that $V(a_1^t, \neg\mathbf{D}) = 0$. Given belief $\mu_1^*(\theta|h_1^t)$, $\mathcal{P}_1$ prefers to contribute; given belief $\mu_1'(\theta|h_{1'}^t)$, $\mathcal{P}_1$ prefers to do nothing. This implies that

$$-s_c \geq \mu_1^*(\mathbf{D}|h_1^t)(V(n,\mathbf{D}) - \rho V(c,\mathbf{D})) + \mu_1^*(\mathbf{B}|h_1^t)(V(n,\mathbf{B}) - V(c,\mathbf{B}))$$
$$-s_c < \mu_1'(\mathbf{D}|h_{1'}^t)(V(n,\mathbf{D}) - \rho V(c,\mathbf{D})) + \mu_1'(\mathbf{B}|h_{1'}^t)(V(n,\mathbf{B}) - V(c,\mathbf{B}))$$

Using these conditions, the fact that $\mathcal{P}_1$ is never better off contributing to a Byzantine $\mathcal{P}_2$ (*i.e.*, $V(n,\mathbf{B}) - V(c,\mathbf{B}) + s_c \geq 0$), and Lemma 6, we can derive a contradiction.

4.3 The Rational Strategy Is an Equilibrium Strategy

Theorem 9. *Assume an altruistic $\mathcal{P}_i$ plays the altruistic strategy $\sigma_{\boldsymbol{A},i}$. Let T be constrained by condition (3) and condition (1) hold in all histories h_2^t, for $t > 0$, in which $\mathcal{P}_2$ is destitute. Then the rational strategy is an equilibrium strategy.*

Proof. By Lemmas 3 and 5 and Theorems 4, 7, and 8.

5 Characterizing the Equilibrium

To understand the implications of Section 4 on the design of cooperative services, we explore, through simulation, the parameter space for which our cooperative equilibrium holds. We ask the following questions:

1. What fraction of altruistic peers suffices to motivate $\mathcal{P}_2$ to pester? The shaded areas in Figure 1 show (for different rates of network loss, different initial beliefs about the likelihood of $\mathcal{P}_1$ being Byzantine, and different worth of receiving a contribution) the fraction of altruistic peers that suffices to trigger $\mathcal{P}_2$'s pestering, as a function of the probability $((1-\alpha)/(1-\rho)^2)$ that an altruistic $\mathcal{P}_1$ will contribute if pestered ($\mathcal{P}_1$'s *generosity*). We assume that $\mathcal{P}_2$ believes an altruistic $\mathcal{P}_1$ follows the altruistic strategy; a Byzantine $\mathcal{P}_1$ never contributes; and a rational $\mathcal{P}_1$ only contributes in round T. This is a conservative estimate on the fraction of altruistic peers sufficient to motivate $\mathcal{P}_2$; in practice, the actual amount is likely to be lower than we report. As expected, for a given level of generosity, it is easier to incentivize $\mathcal{P}_2$ if the value of the contribution increases

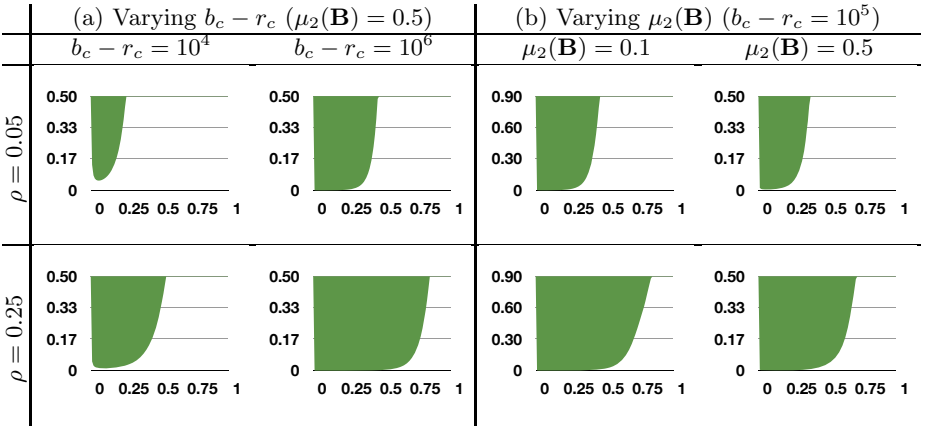
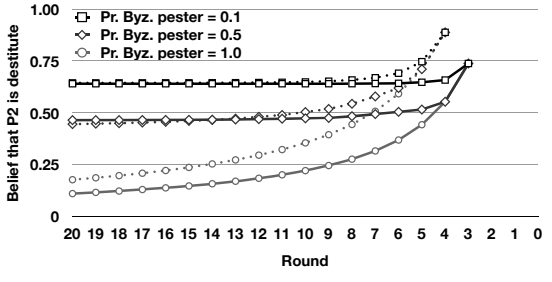


Fig. 1. Sufficient initial beliefs for a rational $\mathcal{P}_2$ in its peer $\mathcal{P}_1$ being altruistic to incentivize $\mathcal{P}_2$ to pester (y-axis, shaded area) for varying amounts of altruistic generosity (x-axis). Simulation run with $s_p = 1$, $T = 20$.



Pr. Byz. pester	$\rho = 0.05$	$\rho = 0.25$
0.1	14	17
0.5	5	9
1.0	2	3

Fig. 2. Left: the belief thresholds of $\mathcal{P}_1$'s beliefs; solid lines represent $\rho = 0.05$; dotted lines represent $\rho = 0.25$. Right: the maximum number of times $\mathcal{P}_1$ contributes for each of these thresholds. Simulation run with $s_c/r_p = 2$, $T = 20$.

and the likelihood of $\mathcal{P}_1$ being Byzantine decreases. Given a highly lossy network, $\mathcal{P}_2$ is also more willing to continue pestering, as it is more willing to attribute to network loss its failure to receive a contribution.

2. How are $\mathcal{P}_1$'s beliefs and the rate of network loss affecting $\mathcal{P}_1$'s willingness to contribute? $\mathcal{P}_1$ may contribute only if its belief that $\mathcal{P}_2$ is destitute is above a certain threshold. We show in Figure 2 how that threshold changes over the course of a game in which $T = 20$. For six configurations, obtained by taking the cross product of two rates of network loss and three different probabilities that a Byzantine $\mathcal{P}_2$ will pester, we plot the belief threshold and report the number of times that $\mathcal{P}_1$ contributes. For a given round, we assign $\mathcal{P}_1$ some initial belief that $\mathcal{P}_2$ is destitute and construct the game tree to determine whether that initial belief is sufficient to motivate $\mathcal{P}_1$ to contribute in that round; we use binary search to approximate the threshold value. As

expected, when the game has only few rounds left and the cost from being pestered is not enough to overcome the cost of contributing, there is no threshold above which $\mathcal{P}_1$ contributes. Note also that increasing ρ increases the belief threshold required to convince $\mathcal{P}_1$ to contribute (as it reduces the expected threat from pestering) but also makes $\mathcal{P}_1$ more likely to contribute when pestered, since past contributions are more likely to have been dropped. Also, the belief threshold increases as the likelihood of a Byzantine $\mathcal{P}_2$ pestering decreases, since it becomes more in $\mathcal{P}_1$'s interest to delay contribution, waiting to see whether $\mathcal{P}_2$ will pester. However, when $\mathcal{P}_1$ observes pestering, its belief that $\mathcal{P}_2$ is destitute increases, and $\mathcal{P}_1$ becomes more more willing to contribute. Finally, decreasing the relative cost of contributing (s_c/r_p) has an obvious effect on $\mathcal{P}_1$'s likelihood to contribute (not shown).

3. Too much generosity? An intriguing conclusion from Figure 1 is that altruistic generosity can make it much harder to motivate $\mathcal{P}_2$ to pester. The reason is that the more generous altruistic peers are, the easier it is for a rational $\mathcal{P}_2$ to determine, from observed signals, whether $\mathcal{P}_1$ is altruistic or not, which in turn affects whether $\mathcal{P}_2$ continues to pester. Figure 1 shows the effects that an altruistic peer's generosity has on cooperation. For higher levels of altruistic generosity, we can only guarantee cooperation if such generosity is offset by a high ρ or $b_c - r_c$. Altruistic generosity becomes a more obvious discriminant if a Byzantine $\mathcal{P}_1$ never contributes, but it becomes less conspicuous with higher rates of network loss, which affects the observed generosity from $\mathcal{P}_2$'s perspective. As expected, $\mathcal{P}_2$ is more willing to pester given a more valuable contribution.

6 Related Work

Incentive-compatible systems and protocols. There has been much work in incentive-compatible systems (*e.g.*, [3,4,13,14,16,20]). None of these systems assume the existence of altruistic players, and only a few [3,16,20] consider Byzantine peers. Our techniques can be applied to many of these systems. For example, BAR Gossip [16], FOX [14], and PropShare [13] can use altruism to incentivize key exchange. Our technique can be used towards implementing BAR Gossip's "fair-enough" exchange and may provide insight into the larger fair exchange problem [11,19]. Finally, rational secret sharing [10] faces a similar problem to the last exchange. However, without a pestering mechanism, our work is not directly applicable.

Irrationality in incentive-compatible protocols. Eliaz [7] proposed the generalization of Nash equilibrium to scenarios where some number of peers may be Byzantine. Aiyer et al. [3] generalized this to the BAR model, which introduced the possibility of altruistic peers and on which our model is based. Abraham et al. [1] describe (k, t)-robust equilibrium, a solution concept in which a rational player does not deviate despite the possibility of collusion by groups up to size k and up to t "irrational" agents that may play any strategy. Similarly, Martin [18] introduces an equilibrium concept in which rational players do not deviate regardless of Byzantine or altruistic players' actions. Our work differs from

previous work by showing the need for altruism to address a key problem in cooperative services and considering real-world issues such as network costs and lossy links. Vassilakis et al. [22] study how altruism affects content sharing in P2P services at the application level. Their approach complements our own; we focus on network-level incentives and issues (such as lossy links) that motivate participants to actually send the content they share at the application level. Their work does not address Byzantine participants.

Game-theory. There has been extensive work that has covered imperfect knowledge, private signaling, and the use of altruism in game theory. The use of altruism to achieve cooperation in the finitely-repeated prisoner's dilemma game was first proposed by Kreps et al. [12]. It was shown that reputations could be maintained even when there was imperfect observation of actions [8]. Cripps et al. later showed that, under certain conditions, reputations cannot be maintained forever unless the action played by the irrational player was part of a rational player's equilibrium strategy [5,6]. None of the previous work consider both the possibility of Byzantine and altruistic players. Many of them also assume that actions or their corresponding signals can either be observed at least publicly [5,8], if not perfectly [12]. More importantly, the focus of this work is the existence (or nonexistence) of equilibrium under general conditions. We focus on the application of theory to a specific problem and a realistic model that we believe to be applicable to many distributed protocols.

7 Conclusion

Despite the presence of altruistic peers in real-world MAD systems, little attention has been given to their role in establishing rational cooperation. In this paper, we take the first step in understanding their function by showing that altruism is necessary and sufficient to motivate rational cooperation in the crucial last exchange between MAD peers. Our results suggest that, while a small fraction of altruistic peers is sufficient to spur rational peers into action even in systems with a large fraction of Byzantine peers, overly generous altruistic peers can irreparably harm rational cooperation.

Acknowledgments. We are grateful to Tom Wiseman and Joe Halpern for many illuminating discussions, and to the anonymous referees. This work is supported by NSF Grant No. 0905625.

References

1. Abraham, I., Dolev, D., Gonen, R., Halpern, J.: Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In: PODC '06, pp. 53–62 (July 2006)
2. Adar, E., Huberman, B.A.: Free riding on Gnutella. First Monday 5(10), 2–13 (2000), http://www.firstmonday.org/issues/issue5_10/adar/index.html
3. Aiyer, A.S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.P., Porth, C.: BAR fault tolerance for cooperative services. In: SOSP '05, pp. 45–58 (October 2005)

4. Cohen, B.: Incentives build robustness in BitTorrent. In: First Workshop on the Economics of Peer-to-Peer Systems (June 2003)
5. Cripps, M.W., Mailath, G.J., Samuelson, L.: Imperfect monitoring and impermanent reputations. Econometrica 72(2), 407–432 (2004)
6. Cripps, M.W., Mailath, G.J., Samuelson, L.: Disappearing private reputations in long-run relationships. J. of Economic Theory 127(1), 287–316 (2007)
7. Eliaz, K.: Fault tolerant implementation. Rev. of Econ. Studies 69, 589–610 (2002)
8. Fudenberg, D., Levine, D.K.: Maintaining a reputation when strategies are imperfectly observed. Review of Economic Studies 59(3), 561–579 (1992)
9. Haeberlen, A., Kouznetsov, P., Druschel, P.: PeerReview: Practical accountability for distributed systems. In: SOSP '07, pp. 175–188 (October 2007)
10. Halpern, J., Teague, V.: Rational secret sharing and multiparty computation. In: Proc. 36th STOC, pp. 623–632 (2004)
11. Kremer, S., Markowitch, O., Zhou, J.: An intensive survey of non-repudiation protocols. Computer Communications 25(17), 1606–1621 (2002)
12. Kreps, D., Milgrom, P., Roberts, J., Wilson, R.: Rational cooperation in the finitely repeated prisoners' dilemma. J. of Economic Theory 27(2), 245–252 (1982)
13. Levin, D., LaCurts, K., Spring, N., Bhattacharjee, B.: BitTorrent is an auction: analyzing and improving BitTorrent's incentives. SIGCOMM Comput. Commun. Rev. 38(4), 243–254 (2008)
14. Levin, D., Sherwood, R., Bhattacharjee, B.: Fair file swarming with FOX. In: IPTPS '06 (February 2006)
15. Li, H., Clement, A., Marchetti, M., Kapritsos, M., Robinson, L., Alvisi, L., Dahlin, M.: FlightPath: Obedience vs choice in cooperative services. In: OSDI '08. pp. 355–368 (December 2008)
16. Li, H.C., Clement, A., Wong, E., Napper, J., Roy, I., Alvisi, L., Dahlin, M.: BAR Gossip. In: OSDI '06, pp. 191–204 (November 2006)
17. Locher, T., Moor, P., Schmid, S., Wattenhofer, R.: Free riding in bittorrent is cheap. In: HotNets '06 (November 2006)
18. Martin, J.P.: Leveraging altruism in cooperative services. Tech. Rep. MSR-TR-2007-76, Microsoft Research (June 2007)
19. Pagnia, H., Gärtner, F.C.: On the impossibility of fair exchange without a trusted third party. Tech. Rep. TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany (March 1999)
20. Peterson, R.S., Sirer, E.G.: Antfarm: efficient content distribution with managed swarms. In: NSDI '09, pp. 107–122 (2009)
21. Piatek, M., Isdal, T., Anderson, T., Krishnamurthy, A., Venkataramani, A.: Do incentives build robustness in BitTorrent? In: NSDI '07, pp. 1–14 (April 2007)
22. Vassilakis, D.K., Vassalos, V.: An analysis of peer-to-peer networks with altruistic peers. Peer-to-Peer Networking and Applications 2(2), 109–127 (2009)
23. Wong, E.L., Leners, J.B., Alvisi, L.: It's on me! The benefit of altruism in BAR environments. Technical Report TR-10-08, UT Austin (2010)

Beyond Lamport's *Happened-Before*: On the Role of Time Bounds in Synchronous Systems

Ido Ben-Zvi[1] and Yoram Moses[2]

[1] Department of Computer Science, Technion
idobz@cs.technion.ac.il
[2] Department of Electrical Engineering, Technion
moses@ee.technion.ac.il

Abstract. Lamport's *Happened-before* relation is fundamental to coordinating actions in asynchronous systems. Its role is less dominant in synchronous systems, in which bounds are available on transmission times over channels. This paper initiates a study of the role that time bounds play in synchronous systems by focusing on two classes of problems: *Ordered Response*, in which a triggering event must be followed by a sequence of events (*"responses"*) performed in a prescribed temporal order, and *Simultaneous Response*, in which the responses must be performed simultaneously. In both cases, information about the triggering event must flow from its site of origin to the responding sites, and the responses must be timed as specified. A generalization of *happened- before* called *Syncausality*, is defined. A pattern of communication consisting of a syncausal chain coupled with an appropriate set of time bound guarantees gives rise to a communication structure called a *centipede*. Centipedes are a nontrivial generalization of message chains, and their existence is shown to be necessary in every execution of every protocol that solves ordered response. A variation on centipedes called *centibrooms* are shown to play an analogous role for Simultaneous Response: Every execution of a protocol for Simultaneous Response must contain a centibroom.

Keywords: synchronous message passing, bounded communication, ordered response, simultaneous response, causality, syncausality, knowledge, common knowledge, levels of knowledge, knowledge and time.

Dedicated to the memory of Amir Pnueli, a magnificent person and great inspiration.

1 Introduction

Many distributed systems applications need to react to spontaneous events, or ones initiated by their environment. Examples for such events are the activation of a fire alarm or smoke detector, a deposit or withdrawal from a bank account, or the identification of an interesting subject in a multi-camera surveillance system. In response to an external event, correct behavior of the system may require

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 421–436, 2010.

that an action, or more generally a set of actions, be performed. When multiple actions are performed, the temporal order in which the actions are performed is often important. Indeed, financial transactions will typically require a variety of actions involving testing various conditions and making related updates to be completed. To capture such situations, we define the following general problem:

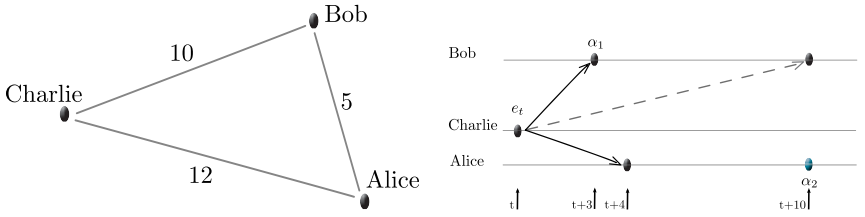
Definition 1 (Ordered Response [OrR]). *An instance of the* Ordered Response *problem is defined by a tuple* $\mathsf{OR} = \mathsf{OR}(e_t, \alpha_1, \ldots, \alpha_k)$*, where* e_t *(the* triggering event*) is a spontaneous external input and* $\alpha_h = \langle a_h, i_h \rangle$ *where* a_h *is an action for process* i_h*.*[1] *A protocol solves* OR *if it guarantees that*
(1) if e_t *occurs, then process* i_h *will perform action* a_h*, for* $h = 1, \ldots, k$*;*
(2) α_h *will happen before* α_{h+1} *does, for all* $h < k$*; and finally*
(3) none of the actions a_h *will be performed in runs in which* e_t *does not occur.*

In the Ordered Response problem, α_h stands for the event of process i_h performing the action a_h. We shall denote the site of the triggering event e_t by i_0. Since the triggering event in $\mathsf{OR}(e_t, \alpha_1, \ldots, \alpha_k)$ is a spontaneous event, information about the occurrence of e_t must flow from i_0 to each of the responding sites. Moreover, these sites must coordinate to perform the actions in the specified order. OR thus combines notification about e_t with a coordination problem. In asynchronous systems, both aspects of **OrR** are handled in a similar fashion, by generating message chains that ensure that Lamport's *happened-before* relation holds between e_t and α_1, and then between α_h to α_{h+1}, for all $h < k$. This paper studies **OrR** and related issues in *synchronous* systems, in which there are known bounds on message transmission, and processes share a global clock. Message chains play a somewhat different role in this setting. The following example illustrates some of the issues at play.

Example 1. Charlie's bank account is temporarily suspended due to credit problems. Should Charlie make a sufficient deposit at his local branch, Banker Bob at headquarters will re-activate the account. Alice holds a cheque from Charlie, but trying to cash it before the account is re-activated will result in her being fined by the bank rather than receiving payment. Alice, Bob and Charlie are connected by a communication network as depicted in Figure 1(a). In particular, messages from Charlie to Bob and Alice take up to 10 and 12 days to be delivered, respectively. We can view this as an instance of **OrR** in which the triggering event is a deposit by Charlie, and the responses are the account re-activation by Bob and, no sooner than Bob's action, Alice's cashing the cheque.

In a particular instance, depicted in Figure 1(b), Charlie makes a deposit at time t, and immediately broadcasts a message stating this to both Alice and Bob. The message reaches Bob in 4 days and Alice in 6. Bob immediately re-activates Charlie's account at $t + 4$. When can Alice deposit the cheque? The

[1] For simplicity, we assume that e_t happens at most once in any given execution, as do each one of the actions performed in response to it. The processes i_h need not be distinct, although it is natural and instructive to assume that adjacent processes in the sequence are distinct, so that $i_h \neq i_{h+1}$.



(a) The network of Example 1 (b) Cheque cleared at $t + 10$

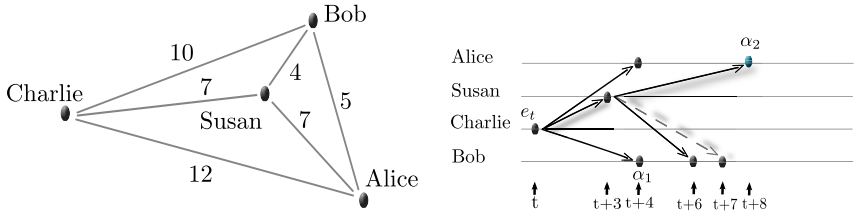
Fig. 1

cheque would be cashed successfully at any time after $t + 4$. However, Alice only knows about Charlie's deposit at $t + 6$. But even at that point, she must wait further. In the absence of additional information indicating when Bob actually received Charlie's message, she is only guaranteed that this will happen by time $t + 10$. Knowing Bob's protocol, she can safely submit the cheque at or after time $t + 10$, but not sooner. □

In this example, Alice acts after Bob does. While in the asynchronous setting she must obtain explicit notification that Bob acted, in the synchronous setting she can base her action on the information that Charlie sent Bob the message at time t, combined with the bound determining when this message will arrive, and her knowledge of Bob's protocol, which ensures action when Bob receives Charlie's message. Her action, which clearly depends on Bob's action having taken place, can be performed without an explicit message chain from Bob. This is no surprise. It is generally accepted that in the presence of bounds, Lamport's happened-before relation is not the sole factor in coordinating actions. Nevertheless, we know of no systematic treatment of how events can be coordinated in such synchronous systems. The purpose of this paper is to study the role that time bounds can play in coordination problems such as **OrR**. As we shall see, the set of communication structures that underly coordination in the synchronous case is considerably richer than it is in the asynchronous case.

Example 2. In a setting similar to Example 1, Susan is Bob's supervisor at the bank. The network is now as depicted in Figure 2(a). Suppose that Charlie broadcasts his deposit to all three, and that communication is delivered as in Figure 2(b). In this case Alice can, as before, submit her cheque at $t + 10$. But she can do even better. Since she receives a message from Susan at $t + 8$ that was sent at $t + 3$, the bound on the (S, B) channel ensures her that Bob is be notified of Charlie's deposit by time $t + 7$. The account will be solvent as of time $t + 7$, and Alice can safely cash her cheque upon receiving Susan's message. □

In both examples, the timing of Alice's action depends on the time bounds. In Example 2, however, information that Alice receives in a message from Susan serves to update her knowledge about when Bob's action is performed, and enables her to perform her action earlier than she could before receiving it. This example illustrates the fact that the proper ordering of events can depend on a subtle interplay between the actual delivery times of messages, and the



(a) The network of Examples 2 and 3 (b) Susan speeds up Alice's response

Fig. 2

time bound guarantees. Notice that, for the purpose of properly ordering Alice's action, Susan plays a similar role in Example 2 to that played by Charlie in Example 1. Information about the triggering event is, in both cases, obtained from Charlie.

Intuitively, any solution to the **OrR** problem must ensure that particular knowledge is obtained following the occurrence of the triggering event. Since the response actions are performed if and only if e_t occurs, each of the responding processes must know that e_t occurred before performing any response action(s). If the triggering event e_t was unconditionally guaranteed to take place at some time t_0, then it would be trivial to coordinate an **OrR** response to e_t without need for any communication. However, being a spontaneous external input, e_t is not guaranteed to occur. Hence, information about its occurrence must flow from i_0 to the responders. The second aspect of **OrR** is the proper ordering of the responses. Before the response α_{h+1} can be performed, i_{h+1} must know that α_h has taken place. As the above examples suggest, this does not require explicit notification, and can be obtained by combining information about actual timing obtained through messages, with *a priori* bound information. But before α_h can take place, process i_h must know that all previous responses have occurred. As we shall show, ordered response is governed by a communication structure that we call a *centipede*, that generalizes the dynamics observed in Example 2.

Examples 1 and 2 illustrate how a process can come to know that communication directed at a remote site has arrived, based on transmission bound information. But bounds can be used in an additional fashion. Namely, if by time $t + b_{ij}$ process j receives no message sent by i at time t, then j can know that no such message was sent. Depending on i's protocol, this can provide j information about i's state at time t. Consider the following refinement of Example 2.

Example 3. In the network of Example 2 depicted in Figure 2(a), suppose that Susan sends Alice a message in every round as long as Susan has not heard from Charlie about an appropriate deposit. In this particular instance, Susan receives a message from Charlie at time $t+2$, at which point she stops sending her update messages. At time $t + 9$ Alice will be able to "time-out" on Susan's time $t + 2$ message. She then knows that Susan heard from Charlie at $t + 2$. Moreover, knowing that Susan relays information to Bob as before, Alice knows that Bob heard about the deposit no later than time $t + 5$. Hence, Alice can safely cash her cheque at time $t + 9$ rather than $t + 10$. □

In Example 3 Alice learns of Charlie's deposit without receiving *any message whatsoever*. She clearly receives no message chain originating from Charlie. Nevertheless, it seems instructive to think of Susan as sending Alice a "silent message" at time $t + 2$, carrying relevant information, by *not* sending an actual message. This view will motivate an extension of Lamport's happened-before relation that we shall call *syncausality*, standing for "synchronous potential causality." A syncausal chain will then be a chain consisting of a sequence of messages and timeouts. As we will see, syncausality is a central element in informing processes about nondeterministic events such as spontaneous external inputs.

The main contributions of this paper are:

- The notion of syncausality is defined, generalizing Lamport's happened-before relation by adding timeout precedence.
- The Ordered Response (**OrR**) and Simultaneous Response (**SiR**) problems are defined, capturing a natural form of coordination in distributed systems. **OrR** is shown to require attaining nested knowledge, while Simultaneous Response requires obtaining common knowledge about the triggering event.
- Syncausality is shown to be a necessary condition for obtaining knowledge about nondeterministic events at a remote site.
- A notion of *Bound-based guarantees* is defined, corresponding to the causal guarantees that depend solely on transmission bounds. These guarantees account for changes in knowledge about remote sites that are not based solely on syncausality.
- *Centipedes*, a particular form of temporal communication structure, are defined. A centipede consists of a syncausal chain with "legs" that consist of bound-based guarantees. Centipedes are shown to be necessary in any run in which nested knowledge of an external input is attained. Consequently, every instance of **OrR** requires the construction of a centipede whose form depends on the instance of **OrR** being implemented. In a precise sense, centipedes are shown to be the analogue in synchronous systems to message chains through a given set of processes in asynchronous systems.
- *Centibrooms*, a slight variant of centipedes, are shown to be necessary for obtaining *common knowledge* (see [12]) of nondeterministic events in synchronous systems. This formally captures the fact that a single *pivotal event* is needed in order to obtain common knowledge in this setting.
- Finally, it is shown that every instance of **SiR** requires the construction of a centibroom whose structure depends on the instance of **SiR** being implemented.
- The technical results are obtained by way of a knowledge-based analysis. This is another illustration of the power of knowledge theory in the analysis of distributed system.

Related work: Explicit and implicit use of time bounds for coordination and improved efficiency is ubiquitous in distributed computing. An elegant example of its use is made by Hadzilacos and Halpern in [11]. That knowledge can be gained by way of timeouts when timing guarantees are available has been part of the folklore from decades. A tutorial by the second author suggests as a viable

topic for future work performing an explicit analysis of the effect of timeouts on knowledge gain [20]. He also presents an example in which communication can be saved by using timeouts. However, [20] does not or suggest modifying Lamport causality to suit synchronous, and none of the new notions or technical results in the current paper were suggested in [20]. Krasucki and Ramanujam in [13] study of the interaction between knowledge and the ordering of events in a distributed system. They consider concurrency in a rather abstract setting, where they show that causality is related to the existence of particular partially ordered sets. They do not explicitly study the synchronous model, however, and do not explicitly consider synchronous time bounds on channels. Moses and Bloom [18] perform a knowledge-based analysis of clock synchronization in the presence of bounds on transmission times. They generalize Lamport's relation by defining a notion of timed causality $e \xrightarrow{\alpha} e'$ that corresponds to e taking place *at least* α time units before e'. It appears that '$\xrightarrow{\alpha}$' is a quantitative generalization of Lamport causality for the purpose of determining relative *timing* of events, while syncausality is a qualitative causality relation more suitable for studying knowledge gain and information flow. A similar notion appears in the work of Patt-Shamir and Rajsbaum [23]. Knowledge about knowledge touches on many fields, ranging from philosophy [15] and psychology [4], to linguistics [10,21], economics [1], AI [16], cryptography [5,24,9] and distributed systems [12,3,22]. In computer security, for example, it often becomes important to ensure that particular agents have access to particular information, and not know particular facts. Our analysis suggests tacit ways in which information can be transmitted in synchronous systems. One implication is that in order to guard against a particular form of knowledge gain, it is essential to deny the possibility of the appropriate centipede forming.

Organization: This paper is organized as follows. Section 2 presents a brief sketch of the model and the definitions used in the theorems and the proofs. Section 3 relates the Ordered Response problem to nested knowledge, and proves a knowledge gain theorem for two processes. Section 4 proceeds to define centipedes and state the Centipede Theorem for multi-process knowledge gain. In Section 5 we introduce the Simultaneous Response problem, review the definition of common knowledge, and relate the two. Centibrooms are defined, and are shown to be necessary in every instance of simultaneous response. Finally, Section 6 presents conclusions.

2 Background and Preliminary Definitions

As mentioned in Section 1, we focus on a simple synchronous setting with a global clock, in which processes that take steps at integer times, and bounds on transmission times over channels are given. We analyze knowledge in protocols that execute in such a setting by following the approach described in [8]. Namely, we separate the definition of the environment for which protocols are designed, formally called the *context*, from the actual protocol being executed in that context. Given a context γ and a protocol P designed to run in γ, there is a

unique set $\mathcal{R} = \mathcal{R}(P, \gamma)$, of all runs of P in γ. This set is called a *system*, and we study how knowledge evolves in systems. The reason why it does not suffice to consider just one system—say the system consisting of all possible runs in γ—is because the protocol being executed plays an important role in determining what is known. Typically, the information inherent in receiving a particular message (or in not receiving one) depends on the protocol being used.

A fully detailed model is beyond the scope of this abstract. The crucial elements are the following.

- We assume that processes can receive external inputs from the outside world. These are determined in a genuinely nondeterministic fashion, and are not correlated with anything that comes before in the execution or with external inputs of other processes. Triggering events are always external events.
- The set of processes is denoted by $\mathbb{P}$. The network consists of the weighted channels graph over $\mathbb{P}$, in which the weights are the bounds b_{ij} for every channel (i, j). A copy of the (weighted) network, as well as the current global time, are part of every process' local state at all times.
- The scheduler, which we typically call the *environment*, is in charge of choosing the external inputs, and of determining message transmission times. The latter are also determined in a nondeterministic fashion, subject to the constraint that delivery satisfies the transmission bounds.
- Time is identified with the natural numbers, and each process is assumed to take a step at each time t. For simplicity, the processes follow deterministic protocols. Hence, a given protocol P for the processes and a given behavior of the environment completely determine the run.
- Events are sends, receives, external inputs and internal actions. All events in a run are distinct, and we denote a generic event by the letter e. For simplicity, events do not take time to be performed. At a given time point a process can perform an arbitrary finite set of actions.

We denote such a context by γ^{s}, and use $\mathcal{R}^{\mathsf{s}}$ to denote a system $\mathcal{R}(P, \gamma^{\mathsf{s}})$ consisting of the set of all runs of some protocol P in synchronous context γ^{s}. An *ND* (or *nondeterministic*) event is either (a) the arrival an external input, or (b) an early receive, i.e., a message delivery that occurs strictly before the transmission bound for its channel is met.

A *process-time node* (or simply *node*) is a pair (i, t), where i is a process and t is a time. Such a node represents an instant on i's timeline. While Lamport's happened-before relation is typically defined among events, we define *syncausality*, its generalization to synchronous systems, as a relation among process-time nodes. This choice allows us to avoid defining *non-receipt* events, for capturing timeouts and the expiration of time bounds. Since every event takes place at a particular node, working with nodes suffices. Formally, we proceed as follows.

Definition 2 (Syncausality). *Fix a run r. The* syncausality *relation $\rightsquigarrow$ over nodes of r is the smallest relation satisfying the following four conditions:*

1. *If $t \leq t'$, then $(i, t) \rightsquigarrow (i, t')$;*
2. *If some message is sent at (i, t) and received at (j, t') then $(i, t) \rightsquigarrow (j, t')$;*

3. *If i and j are connected by a channel with bound b_{ij} then $(i,t) \rightsquigarrow (j, t+b_{ij})$ for all t; and*
4. *If $(i,t) \rightsquigarrow (h,\hat{t})$ and $(h,\hat{t}) \rightsquigarrow (j,t')$, then $(i,t) \rightsquigarrow (j,t')$.*

Essentially, the first two clauses correspond to the local precedence and message precedence steps of Lamport's happened-before. Syncausality thus directly generalizes of happened-before. The third clause corresponds to *timeout precedence*.

2.1 Definition of Knowledge

We focus on a very simple logical language in which the set Φ of primitive propositions consists of propositions $\mathsf{occurred}(e)$ and $\mathtt{ND}(e)$, where e is an event. To obtain the logical language $\mathcal{L}$, we close Φ under propositional connectives and knowledge formulas. Thus, $\Phi \subset \mathcal{L}$, and if $\varphi \in \mathcal{L}$ and $i \in \mathbb{P}$, then $K_i\varphi \in \mathcal{L}$.[2] The formula $K_i\varphi$ is read *process i knows φ*. For ease of exposition, we assume that processes have *perfect recall* so that, intuitively, their local state at any time contains the full history of events that they have experienced. This assumption is needed only for the analysis of Response problems, and can be obtained by adding an auxiliary variable—only at the modeling stage and not the implementation—keeping track of the local history.

For defining the meaning of knowledge formulas, we follow the framework of [8]. The truth of formulas is evaluated with respect to a triple (R,r,t) consisting of a set of runs R, a run $r \in R$, and a time $t \in \mathbb{N}$, and we use $(R,r,t) \vDash \varphi$ to state that φ holds at time t in run r, with respect to R. Denoting by $r_i(t)$ process i's local state at time t in r, we inductively define

$(R,r,t) \vDash \mathsf{occurred}(e)$ if e occurs in r at a time $t' \leq t$;
$(R,r,t) \vDash \mathtt{ND}(e)$ if e is an ND event in r and $(R,r,t) \vDash \mathsf{occurred}(e)$; while
$(R,r,t) \vDash K_i\varphi$ if $(R,r',t') \vDash \varphi$ for every run r' satisfying $r_i(t) = r'_i(t')$.

By definition, $K_i\varphi$ is satisfied at a point (r,t) if φ holds at all points of R at which i has the same local state as at (r,t). Thus, given R, the local state determines what facts are true.

3 Knowledge and Ordered Response

We can now prove that particular nested knowledge is a precondition to action in **OrR**, formalizing and justifying the informal discussion in the introduction of how performing response actions in **OrR** requires processes to obtain nested knowledge.

Theorem 1. *Let $\mathsf{OR} = \mathsf{OR}(e_{\mathtt{t}}, \alpha_1, \ldots, \alpha_k)$ be an instance of* **OrR***, and assume that protocol P solves OR in $\gamma^{\mathtt{s}}$. Let $r \in \mathcal{R}^{\mathtt{s}}$ be a run in which $e_{\mathtt{t}}$ occurs, let $1 \leq h \leq k$, and let t_h be the time at which i_h performs action a_h in r. Then*

$$(\mathcal{R}^{\mathtt{s}}, r, t_h) \vDash K_{i_h} K_{i_{h-1}} \cdots K_{i_1} \mathsf{occurred}(e_{\mathtt{t}}).$$

[2] This is a simplified language for ease of exposition. In Section 5 we extend it to allow for common knowledge.

Theorem 1 implies that for the last action in an ordered response to be performed, a nested knowledge formula stating that the last responder knows that the previous one knows... that the first responder knows e_{t}. An analysis of when this property can hold will uncover the communication structure required in every run of an **OrR** protocol.

3.1 Causal Cones and Knowledge Gain

Having related the Ordered Response problem to knowledge, we now proceed to develop a theory that will provide the link that is still missing, relating knowledge and causality in synchronous systems. Lamport relates the happened-before relation to light cones in Minkowski space-time [14]. In the same vein, it is natural to consider past and future causal "cones" induced by syncausality. We define the *future causal cone* of a node α to be $\mathsf{fut}(r,\alpha) = \{\theta : \alpha \overset{r}{\rightsquigarrow} \theta\}$. Similarly, the *past causal cone* of α is $\mathsf{past}(r,\alpha) = \{\theta : \theta \overset{r}{\rightsquigarrow} \alpha\}$. Observe that the cones induced by syncausality in synchronous systems are significantly larger than the ones that follow just from Lamport's happened-before relation. Moreover, just as the future and past cones meet at the current point in space-time for light cones, we can show:

Lemma 1. *For all runs $r \in \mathcal{R}^{\mathsf{s}}$ and nodes α:* $\mathsf{fut}(r,\alpha) \cap \mathsf{past}(r,\alpha) = \{\alpha\}$.

The first step in relating syncausality to knowledge in synchronous systems comes from the observation that the events that occur in the past (syncausal) cone of a node completely determine the local state at the node. A proof by induction on all nodes (j,t') with $0 \le t' \le t$ shows:

Lemma 2. *Let $r, r' \in \mathcal{R}^{\mathsf{s}}$. If $\mathsf{past}(r,(i,t)) = \mathsf{past}(r',(i,t))$ and both runs agree on the initial states, external inputs, and early receives at all nodes of $\mathsf{past}(r,(i,t))$, then $r_i(t) = r'_i(t)$.*

Since the knowledge of a process in $\mathcal{R}^{\mathsf{s}}$ is determined by its local state, Lemma 2 implies that this knowledge depends only on the past causal cone. In the asynchronous setting, Chandy and Misra have shown that a process can know only about events in its past causal cone [3]. This is not the case in synchronous systems. It is true, however, for events known to be nondeterministic: Indeed, we can now state and prove using Lemma 2 the following knowledge gain theorem for two processes:

Theorem 2 (2-process Knowledge Gain). *Assume that e takes place at (i_0,t) in $r \in \mathcal{R}^{\mathsf{s}} = \mathcal{R}(P,\gamma^{\mathsf{s}})$. If $(\mathcal{R}^{\mathsf{s}},r,t') \vDash K_{i_1}\mathtt{ND}(e)$ then $(i_0,t) \rightsquigarrow (i_1,t')$.*

The proof of Theorem 2 is obtained by constructing a run r' indistinguishable to i_1 at t' from r in which no ND events occur outside $\mathsf{past}(r',(i_1,t')) = \mathsf{past}(r,(i_1,t'))$. Theorem 2 captures a natural sense in which syncausality is a notion of potential causality for the synchronous model. Recall that every external input is, in particular, an ND event. Thus, Theorem 2 implies that knowledge about the occurrence of an external input requires a syncausal connection. Thus,

by Theorem 1 it follows that in every run of an instance $\mathsf{OR} = \mathsf{OR}(e_{\mathtt{t}}, \alpha_1, \ldots, \alpha_k)$ there must be syncausal chains connecting the node (i_0, t) at which $e_{\mathtt{t}}$ to each of the nodes θ_h at which the responses α_h are performed.

4 Bound Guarantees and Centipedes

Our purpose in this section is to formalize and generalize the examples in the introduction, which dealt with a two-response instance of **OrR**, to obtain a structural requirement for arbitrary instances of **OrR**. Before we do so, let us revisit the issue of bounds on message transmission times. Recall that the bound b_{ij} provides a guarantee on the pace at which messages from i will reach j. We are sometimes interested in using these bounds to obtain upper bounds on message chains (or syncausal chains) that may involve a path of channels in the network, and not just one. It is natural, and will be useful, to define the *transmission distance* between processes h and k, which we denote by $D(h, k)$, to be the minimal distance between h and k in the weighted graph consisting of the communication network, in which edges (i, j) are assigned as weights the bounds b_{ij} on transmission times. In particular, $D(i, i) = 0$ for all $i \in \mathbb{P}$.

Intuitively, the examples in the Introduction demonstrated that without explicit information about actual communication deliveries, knowledge that a syncausal chain from an ND event has reached a destination node cannot be obtained before time equal to or exceeding the transmission distance from the source to destination has transpired. It is convenient to relate nodes by **bound guarantees** based on transmission distances as follows:

Definition 3. *We write* $(i, t) \dashrightarrow (j, t')$ *if* $t' \geq t + \mathsf{D}(i, j)$.

Observe that bound guarantees are independent of the speed at which messages *actually* arrive; they depend only on the weighted network topology. This is why it may be possible to know that a delivery has taken place based on the send event and the bound guarantees, without requiring further direct proof. As we shall see, if anyone knows at time t' that $(i, t) \rightsquigarrow (j, t')$ without obtaining syncausal information about early receives in the syncausal chain from (i, t) to (j, t'), then necessarily $(i, t) \dashrightarrow (j, t')$. Bound guarantees tie in with syncausal chains to determine the information structure that must underly solutions to ordered response. In fact, we now define a structure consisting of a careful combination of $\rightsquigarrow$ and $\dashrightarrow$ relations, that, in a precise sense, captures knowledge gain and ordering of events in synchronous systems.

Definition 4 (Centipede). *Let* $r \in \mathcal{R}^{\mathtt{s}}$, *let* $i_h \in \mathbb{P}$ *for* $0 \leq h \leq k$ *and let* $t \leq t'$. *A* centipede *for* $\langle i_0, \ldots, i_k \rangle$ *in the interval* $(r, t..t')$ *is a sequence of nodes* $\theta_0 \rightsquigarrow \theta_1 \rightsquigarrow \cdots \rightsquigarrow \theta_k$ *such that* $\theta_0 = (i_0, t)$, $\theta_k = (i_k, t')$, *and* $\theta_h \dashrightarrow (i_h, t')$ *holds for* $h = 1, \ldots, k-1$.

A centipede for $\langle i_0, \ldots, i_k \rangle$ in the interval $(r, t..t')$ is depicted in Figure 3. Intuitively, every node θ_{h+1} serves to ensure that its corresponding "leg" node (i_{h+1}, t') is causally affected, both by the origin node (i_0, t), and by the node θ_h.

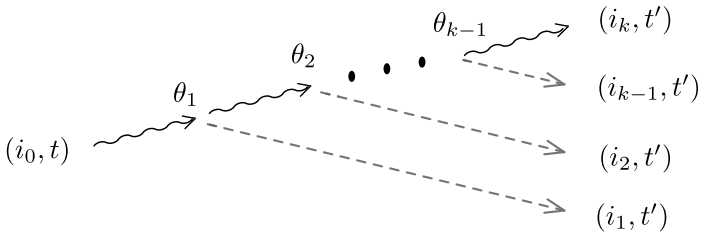


Fig. 3

The node θ_h, in turn, ensures the existence of a shorter centipede. We emphasize that the bound guarantee in $\theta_h \dashrightarrow (i_h, t')$ does not only (and will typically *not*) stand for causality based on unsent messages. Rather, it also stands for the fact that the information available at θ_h guarantees that i_h has enough time to learn by time t' about the part of the centipede to its left. This inductive structure underlies the close relationship the we will show between centipedes and knowledge gain. Observe that the shaded lines in Figure 2(b) outline an underlying 1-legged centipede.

We remark that, since both $\rightsquigarrow$ and $\dashrightarrow$ are reflexive, it is possible for adjacent θ_j's to coincide. Moreover, it is possible (in fact, probably quite common) for θ_h to occur at the same site i_h, with its "leg" (i_h, t_h) in many cases. Indeed, every simple (Lamport-style) message chain gives rise to a centipede of a simple form in which all internal nodes θ_h are co-located in this sense with their respective legs (i_h, t_h). It follows that a centipede is a natural, albeit nontrivial, generalization of a Lamport-causal chain. We can now show that a centipede is a necessary condition for knowledge gain in synchronous systems:

Theorem 3 (Centipede Theorem). *Let P be an arbitrary protocol, and let $r \in \mathcal{R}^{\mathsf{s}} = \mathcal{R}(P, \gamma^{\mathsf{s}})$. Moreover, assume that e is an ND event at (i_0, t) in r. If $(\mathcal{R}^{\mathsf{s}}, r, t') \vDash K_{i_k} K_{i_{k-1}} \cdots K_{i_1} \mathtt{ND}(e)$, then there is a centipede for $\langle i_0, \ldots, i_k \rangle$ in $(r, t..t')$.*

The proof of Theorem 3 is based on a nontrivial and subtle knowledge-based analysis of the interaction between bound guarantees and syncausality. Combining the Centipede Theorem with Theorem 1 we can obtain:

Corollary 1. *Let P be a protocol solving $\mathsf{OR} = \mathsf{OR}(e_{\mathtt{t}}, \alpha_1, \ldots, \alpha_k)$ in γ^{s}, and assume that $e_{\mathtt{t}}$ occurs at (i_0, t) in $r \in \mathcal{R}^{\mathsf{s}}$. If i_k performs a_k at time t' in r then there is a centipede for $\langle i_0, \ldots, i_k \rangle$ in $(r, t..t')$.*

5 Simultaneous Response and Centibrooms

In synchronous systems it is often desirable to perform actions simultaneously at different sites. A natural variant of **OrR** is the *Simultaneous Response* problem, defined as follows.

Definition 5 (Simultaneous Response [SiR]). *Let e_t be an external input. Then $\mathsf{SR} = \mathsf{SR}(e_t, \alpha_1, \ldots, \alpha_k)$ defines an instance of the* Simultaneous Response *problem. A protocol solves SR if it guarantees that if the triggering event e_t occurs, then at some later point all actions $\alpha_1, \ldots, \alpha_k$ in the response set of SR will be performed simultaneously.*

As in the case of ordered response, we can obtain insight into the structure of simultaneous response via knowledge theory. The state of common knowledge has been shown to play an important role in agreements and in coordinating simultaneous actions [12,7,8]. Common knowledge, however, involves knowledge about knowledge for unbounded depths. To formally treat common knowledge, we extend our logical language $\mathcal{L}$ by adding the operators E_G (*everyone in G knows*) and C_G (*the processes in G have common knowledge that*) for every $G \subseteq \mathbb{P}$. Thus, if $\varphi \in \mathcal{L}$ then so are $E_G\varphi$ and $C_G\varphi$. We use $(E_G)^k$ as shorthand for nesting k levels of E_G. The definition of satisfaction for formulas is now extended by the following clauses:

$$(R, r, t) \vDash E_G\varphi \text{ if } (R, r, t) \vDash K_i\varphi \text{ for every } i \in G; \quad \text{and}$$
$$(R, r, t) \vDash C_G\varphi \text{ if } (R, r, t) \vDash (E_G)^k\varphi \text{ for every } k \geq 1.$$

In the terminology of [8], the responses $\alpha_1, \ldots, \alpha_k$ in an instance of **SiR** induce a *perfectly coordinated ensemble* of events in $\mathcal{R}^s$. Using Proposition 11.2.2 of [8] we can conclude that common knowledge of the occurrence of triggering event e_t must hold before the responses can be performed in **SiR**:

Theorem 4. *Let $\mathsf{SR} = \mathsf{SR}(e_t, \alpha_1, \ldots, \alpha_k)$ be an instance of* **SiR***, and assume that protocol P solves SR in γ^s. Moreover, $G = \{i_1, \ldots, i_k\}$ be the set of processes appearing the response set of SR. Finally, let $r \in \mathcal{R}^s$ be a run in which e_t occurs. If the responses are performed at time t in r, then $(\mathcal{R}^s, r, t) \vDash C_G\mathtt{ND}(e_t)$.*

As we now show, the Centipede Theorem can be extended to show an analogous result for common knowledge, with the centipede replaced by a simpler structure, defined as follows:

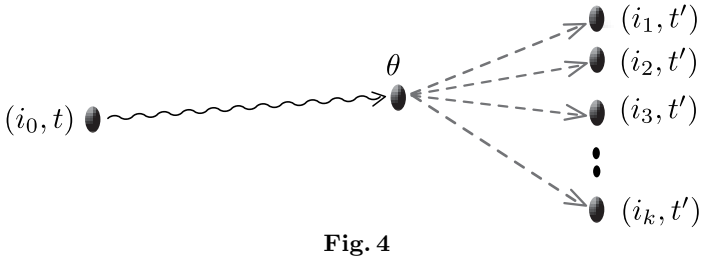


Fig. 4

Definition 6 (Centibroom). *Let $t \leq t'$ and $G \subseteq \mathbb{P}$. There is a* centibroom *$Br\langle i_0, G\rangle$ in $(r, t..t')$ if there is a node θ satisfying $(i_0, t) \rightsquigarrow \theta \dashrightarrow (i_h, t')$ for all $i_h \in G$.*

A pictorial depiction of a centibroom is given in Figure 4. The node θ is called the *pivot* of the centibroom. Observe that a pivot node embodies a "pivotal event" for the group G of processes: This pivot makes it possible, in principle, to guarantee that all members of G will know by time t' of the existence of this pivot for time t'.

Clearly, centibrooms are simpler structures than general centipedes. Notice, however, that a centibroom for $G = \{j_1, \ldots, j_\ell\}$ can be considered as a condensed representation of infinitely many centipedes, each of which can support knowledge gain of a particular formula. More concretely, we have the following.

Lemma 3. *Let $G \subseteq \mathbb{P}$, and let θ be a pivot node for $Br\langle i_0, G\rangle$ in $(r, t..t')$. Then for every sequence $\langle i_1, \ldots, i_k\rangle \in G^k$ of processes in G, the sequence $(i_0, t) \cdot \theta^k$ (where θ repeats k times) is a centipede for $\langle i_0, \ldots, i_k\rangle$ in $(r, t..t')$.*

Notice that Lemma 3 does not bound the value of k, nor does it restrict the possibility of repetitions in the sequence $\langle i_1, \ldots, i_k\rangle$ in question. Given the centipede Theorem, it seems natural to conjecture that a centibroom is a candidate to serve as the structure underlying common knowledge. We now show that this is indeed the case.

Theorem 5 (Common Knowledge Gain). *Let P be an arbitrary protocol, let $G \subseteq \mathbb{P}$, let $\mathcal{R}^{\mathsf{s}} = \mathcal{R}(P, \gamma^{\mathsf{s}})$, and let $r \in \mathcal{R}^{\mathsf{s}}$. Moreover, assume that e is an ND event at (i_0, t) in r. If $(\mathcal{R}^{\mathsf{s}}, r, t') \vDash C_G\mathtt{ND}(e)$, then there is a centibroom $Br\langle i_0, G\rangle$ in $(r, t..t')$.*

The proof of Theorem 5 is based on the Centipede Theorem. Recall that $C_G\varphi$ implies arbitrarily deeply nested knowledge of φ. Every such nested knowledge formula implies the existence of a centipede. A nested knowledge formula is constructed whose centipede has sufficiently many nodes that at least one of them must be a pivot for G at t'. What results is the desired centibroom.

The pivot node in a centibroom embodies a "pivotal event" for the group G of processes. Theorem 5 shows that such a pivotal event is the *only* way common knowledge can arise in synchronous systems. This demonstrates that the nature of common knowledge is finitistic, despite its familiar definition being based on an infinite conjunction of facts. This phenomenon is consistent with the analysis of common knowledge in the work on fault-tolerance [6,19,17]. There, too, common knowledge arises at some time t' exactly if there is some property S of the correct nodes that ensures that all agents will know by time t' that the property S held in the run.

We remark that Theorem 5 relates to a familiar situation involving the evolution of knowledge in broadcasts. In a flooding protocol or a radio broadcast, for example, the contents being broadcast become common knowledge to a growing set of participants with time. Typically, after a time interval equivalent to the diameter of the system, the contents can become common knowledge to *all* processes in the system.

As in the case of Ordered Response, we can use Theorem 5 relate the *Simultaneous Response* problem **SiR** and centibrooms, as follows:

Corollary 2. *Let P be a protocol solving $\mathsf{SR} = \mathsf{SR}(e_{\mathtt{t}}, \alpha_1, \ldots, \alpha_k)$ in $\gamma^{\mathtt{s}}$, and assume that $e_{\mathtt{t}}$ occurs at (i_0, t) in $r \in \mathcal{R}^{\mathtt{s}}$. If the response actions are performed at time t' in r, then there is a centibroom $Br\langle i_0, G\rangle$ in $(r, t..t')$.*

6 Conclusions

We have performed an analysis of causality in synchronous systems, where processes share a global clock and channels have bounded transmission times. We introduced the Ordered Response and Simultaneous Response problems, as natural coordination tasks in a distributed system. They involve natural forms of temporal and simultaneous coordination in any given context. While in asynchronous systems we have, via Lamport's relation, that causality requires message chains, causality has a richer structure in the synchronous setting. First of all, timeouts allow information flow via non-messages. This gives rise to syncausal chains, which consist of sequences of messages and timeouts. But syncausal chains do not tell the full story of causality in synchronous systems. We showed that guaranteeing that one event happens before another does not depend on a linear structure of information flow. Rather, the centipede structure, consisting of a restricted tree form combining syncausality and bound guarantees, is at the base of temporally ordering events under synchrony. Similarly, centibrooms are at the essence of simultaneous coordination in synchronous systems.

Our results all hold in particular in the case in which $b_{ij} = \infty$ for all channels, so that communication is asynchronous (although processes share the global clock and can move at each step). Because communication is asynchronous, bound guarantees are useless in this setting. Syncausality reduces to Lamport's happened-before, all possible centipedes collapse to message chains, and centibrooms do not exist. Thus, our results also apply to such contexts, reproving Chandy and Misra's Knowledge-gain theorem in a slightly more general setting. Asynchrony of communication alone suffices for this type of implosion.

Our theorems provide necessary conditions for information flow based on syncausality. How knowledge actually evolves in a system will depend on the particular protocol used. For example, a timeout at $t + b_{ij}$ is ineffective if the protocol would never have the "sender" i send a message at time t. Alice can learn by timing out on Susan's message in Example 3 only because, had Susan not obtained confirmation of Charlie's deposit, she would have sent a message at $t + 2$. The protocol used plays a crucial role in this knowledge transfer. As a first study of the role that protocols play in determining information flow in the synchronous contexts $\gamma^{\mathtt{s}}$, we analyze the full-information protocol in a follow-on paper [2]. The necessary conditions in this paper's theorems are necessary and sufficient in that case. It follows that our characterization of coordination in terms syncausality, centipedes, and centibrooms is, in a precise sense, tight.

In future work we plan to study causality and coordination in a similar fashion for other models with synchronous features. Perhaps the most urgent would be a study of semi-synchronous systems in which clocks are private, and may drift over time. Some of our ideas regarding timeouts, syncausality and bound

guarantees should have suitably modified analogues in such contexts. But additional issues may also be also be involved there, knowledge about the current level of synchronization, and about other processes' knowledge regarding clocks. Our work proves that outside the realm of asynchronous systems causality and the ordering of events involve more than Lamport's happened before relation. The theory of causality beyond asynchronous systems and Lamport's happened before promises to be rich and interesting.

Acknowledgments. We thank Danny Dolev, Idit Keidar and Ayelet Shiri for useful discussions that improved the presentation. Special thanks to Moshe Vardi for suggesting the term *centibroom*.

This research was supported in part by ISF grant 1339/05.

References

1. Aumann, R.J.: Agreeing to disagree. Annals of Statistics 4(6), 1236–1239 (1976)
2. Ben-Zvi, I., Moses, Y.: Sufficient conditions for knowledge gain and information flow in synchronous systems (2010) (in preparation)
3. Chandy, K.M., Misra, J.: How processes learn. Distributed Computing 1(1), 40–52 (1986)
4. Clark, H.H., Marshall, C.R.: Definite reference and mutual knowledge. In: Joshi, A.K., Webber, B.L., Sag, I.A. (eds.) Elements of Discourse Understanding. Cambridge University Press, Cambridge (1981)
5. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(5), 644–654 (1976)
6. Dwork, C., Moses, Y.: Knowledge and common knowledge in a Byzantine environment: crash failures. Information and Computation 88(2), 156–186 (1990)
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Common knowledge revisited. In: Proc. 6th TARK, pp. 283–298. Morgan Kaufmann, San Francisco (1996)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (2003)
9. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(1), 186–208 (1989)
10. Grice, H.P.: Logic and conversation, pp. 41–58 (1975)
11. Hadzilacos, V., Halpern, J.Y.: Message-optimal protocols for byzantine agreement. Mathematical Systems Theory 26(1), 41–102 (1993)
12. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM 37(3), 549–587 (1990)
13. Krasucki, P.J., Ramanujam, R.: Knowledge and the ordering of events in distributed systems (extended abstract). In: Proc. 5th TARK, pp. 267–283. Morgan Kaufmann, San Francisco (1994)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
15. Lewis, D.: Convention, A Philosophical Study. Harvard University Press, Cambridge (1969)
16. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Michie, D. (ed.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)

17. Mizrahi, T., Moses, Y.: Continuous consensus via common knowledge. Distributed Computing 20(5), 305–321 (2008)
18. Moses, Y., Bloom, B.: Knowledge, timed precedence and clocks. In: Proc. 13th ACM Symp. on Principles of Distributed Computing, pp. 294–303 (1994)
19. Moses, Y., Tuttle, M.R.: Programming simultaneous actions using common knowledge. Algorithmica 3, 121–169 (1988)
20. Moses, Y.: Knowledge and communication: a tutorial. In: Proc. 4th TARK, pp. 1–14. Morgan Kaufmann, San Francisco (1992)
21. Parikh, R.: Finite and infinite dialogues. In: Moschovakis, Y.N. (ed.) Logic from Computer Science, MSRI Publication No. 21, pp. 481–497. Springer, Heidelberg (1991)
22. Parikh, R., Krasucki, P.: Levels of knowledge in distributed computing. Sādhanā 17(1), 167–191 (1992)
23. Patt-Shamir, B., Rajsbaum, S.: A theory of clock synchronization (extended abstract). In: Proc. 26th ACM STOC, pp. 810–819 (1994)
24. Rabin, M.O.: How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187 (2005), Originally written in (1981)

On the Power of Non-spoofing Adversaries

H.B. Acharya[1] and Mohamed Gouda[1,2]

[1] Department of Computer Science
University of Texas at Austin
[2] National Science Foundation
{acharya,gouda}@cs.utexas.edu

Abstract. One of the fundamental concepts in network security is the *active adversary*. Such an adversary is defined, in the classic paper by Dolev and Yao, as an adversary that (in addition to eavesdropping passively), can "impersonate another user and ... alter or replay the message". Thus, the original definition of an active adversary includes the ability to spoof (lie about its identity). In this paper, we study the special case of active adversaries who are restricted from spoofing. As in the original study by Dolev and Yao, the motivation of our adversary is to break the confidentiality of the message being transmitted using a cascade protocol (a protocol in which neither sender nor receiver name stamps the messages they send). We prove a very surprising result: our weaker adversary, who is restricted from spoofing, is in fact exactly as powerful as the unrestricted Dolev-Yao adversary with respect to the goal of breaking confidentiality of cascade protocols.

1 Introduction

Public-key encryption is widely used to secure network communication. As Needham and Schroeder [12] as well as Dolev and Yao [6] point out, these systems can be attacked by "active" adversaries that can, in addition to listening passively, "*impersonate another user* and ... alter or replay the message".

In recent years, there has been substantial research on guaranteeing the authenticity of packet information. IPsec [10] and hop integrity [9] are two important examples of signing packets. Cryptographic signatures are computationally expensive, so other groups have developed other anti-spoofing safeguards. The most popular approach is packet filtering (notably ingress filtering [7], Martian address filtering [1], forwarding-table based filtering, route-based distributed filtering [13], and Source Address Validity Enforcement [11]). A similar approach, packet tracing, involves observing traffic at routers and reconstructing a packet's actual path [2]. Network intrusion detectors such as DECIDUOUS [4] can also be used to locate an adversary.

Given this wide range of tools against spoofing, it becomes reasonable to assume that in many cases the active adversary is no longer able to "impersonate another user". How much of the power of Dolev and Yao's active adversary is lost when it cannot spoof?

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 437–449, 2010.

In this paper, we study the power of a non-spoofing adversary to break the security of cascade protocols, as studied in the original paper that defines active adversaries [6]. We obtain an extremely surprising result: if the aim of the adversary is to compromise confidentiality, then there is no difference whatsoever between the power of an adversary that can spoof and an adversary that cannot. If a protocol can protect the confidentiality of a message from a non-spoofing adversary, it will also resist an adversary that can spoof.

In this paper, we first provide a proof of the above statement for the simple case of two-step cascade protocols. Next, we prove the result that the security of a general k-step cascade protocol is equivalent to the security of a set of two-step cascade protocols. Formally, we demonstrate how to convert any given k-step cascade protocol P into a set of two-step cascade protocols $P_{\{\}}$, such that P can be broken by adversary X iff $P_{\{\}}$ can be broken by X. (We define a set of protocols to be broken by an adversary iff at least one of the protocols in the set is broken by the adversary.) Note that, iff P can be broken by a spoofing adversary A, $P_{\{\}}$ can be broken by A also. By definition, there exist one or more two-step cascade protocols in $P_{\{\}}$ that can be broken by A. But by our first result, as these are two-step cascade protocols, they are also broken by a non-spoofing adversary Z. In other words, iff P is broken by a spoofing adversary A, $P_{\{\}}$ is also broken by the non-spoofing adversary Z. Applying the equivalence of P and $P_{\{\}}$ again, we find that this means P is broken by Z. This concludes our proof.

The next section introduces our notational and conceptual conventions.

2 Users and Adversaries

In this paper, we have used the lambda-calculus convention of representing the application of a function to an argument, so $F(X)$ is written FX. However, we assume right-associativity: FGX represents $F(G(X))$ (unlike lambda calculus, where FGX means $H(X)$ where $H = F(G)$).

As FX means $F(X)$, in order to represent concatenation, we use the comma operator. "F concatenated with X" is written F, X.

A *user* is a process with a unique identifier such as I, J or K. The users are connected by a communication network, and can send messages to each other. The protocol followed by such messages, in order to ensure their confidentiality, is the focus of this paper.

The users of a protocol form a *public key system*.

1. Every user X has two functions:
   (a) The public key function B_X
   (b) The private key function R_X
   Both B_X and R_X map finite binary sequences (i.e. numbers) to finite binary sequences.
2. The pairs (X, B_X) are available to all users.
3. R_X is known only to X.
4. B_X and R_X satisfy the conditions

- $\forall M, B_X R_X M = R_X B_X M = M$.
- It is cryptographically hard to obtain M from $B_X M$ without access to R_X.

Note that the second condition forces every user X to have distinct B_X as well as R_X. (Otherwise, let $B_I = B_J$. As J has access to all public keys, it knows this. Now J can deduce that $R_J B_I M = M$, so it can obtain M from $B_I M$.)

The only operations users can employ are their public and private key functions. Note that it is not possible to distinguish whether it is encryption or decryption which is being applied to an argument, unless the argument is known. For instance, suppose user I applies B_I to the argument M; this is an encryption. But if I applies B_I to $R_I M$, it is a decryption.

However, several layers of these operations can be applied. For instance, user I has access to the key R_I and all keys B_X (where X is any user), so from M it can generate $R_I B_J B_I M$. Such a combination of operations is called a *key sequence*.

Adjacent matching public and private key functions in a key sequence cancel each other out. So for instance, $R_I B_J R_J B_I M = R_I B_I M = M$. A key sequence with no such adjacent matching pairs is called a *simple* key sequence. Reducing a key sequence to a simple key sequence by recursively removing all such matching pairs is called *simplification*.

The *adversary* is a valid user in the protocol system, whose motive is to obtain the message M being communicated between two other users (say I and J). In this paper, we consider two models of adversary:

1. The "classic" Dolev-Yao adversary A.
2. The "non-spoofing" adversary Z.

The adversary A can perform the following actions:

1. A can obtain any message passing through the network.
2. As a legitimate user, A can send any message to any other user in the network. In particular, A can send messages to I with a fake source address stating the message is from J, and similarly, can send messages to J pretending to be I. Of course, it can also send messages that claim, honestly, that they are from A.
3. A has a private key function R_A, and access to the public key functions of all the other users in the system. It can apply any key sequence composed of these keys, to any message it obtains.

A has the following restrictions to its power:

- A cannot break the cryptography used. For instance, it cannot extract M from $B_I M$ without obtaining R_I.
- A cannot tamper with the public information. For instance, it cannot cause, say, I to think B_A is the value of B_J.

A can obtain messages, apply its own keys, and send messages to any user *with its own or a different source address*. Hence, as A can masquerade as another user (*spoof*), we refer to it as a *spoofing* adversary.

Adversary Z is identical to A, but has one more restriction on its power : it is a valid user of the system, and can obtain messages, apply its own keys, and send messages to any user, but such messages *bear the true source address*. Such an adversary cannot spoof, i.e. lie about its identity in a message; hence it is called a *non-spoofing* adversary.

In the next section, we describe the simple family of two-step cascade protocols; section 4 shows that A and Z are equivalent with respect to their power to break the security of these protocols.

3 Two-Step Cascade Protocols

In this section, we discuss a simple class of cascade protocols, consisting of only two steps : a *request* and a *reply*. The objective of such a protocol is to transmit a secret message (the plaintext) M between two users. This is a simplification of the general theory for k-step cascade protocols developed in [6].

A *two-step cascade protocol* is defined by two sets of key sequences a_{XY} and b_{XY}.

$$a_{XY} \in \{R_X, B_X, B_Y\}^*$$
$$b_{XY} \in \{R_Y, B_X, B_Y\}^*$$

where X and Y are any two users. In practice, X and Y are distinct (there is no reason why a user should send messages to itself).

The protocol has the property of being *uniform*: the key sequences have the same structure, irrespective of which users are trying to communicate. For instance, suppose $a_{IJ} = R_I B_J B_J$; then in this protocol, $a_{KL} = R_K B_L B_L$. Note that it is not necessary that $a_{XY} = a_{YX}$ or $b_{XY} = b_{YX}$.

The first class of messages in the protocol, the *request*, has the form

$$X \rightarrow Y : X, a_{XY} M, Y$$

The only part of the message that is encrypted is the plaintext. Source and destination are sent in the clear.

The second class of messages, the *reply*, has the form

$$X \leftarrow Y : Y, b_{XY} a_{XY} M, X$$

Note that b_{XY} is applied to the entire sequence $a_{XY} M$ and not to the original message M.

A well-known example of such a protocol is due to Diffie and Hellman [5].

$$\begin{aligned} I \rightarrow J &: I, B_J R_I M, J \\ I \leftarrow J &: J, B_I R_J B_I R_J B_J R_I M, I \\ &= J, B_I R_J M, I \end{aligned}$$

We observe that $a_{XY} = B_Y R_X$ and $b_{XY} = B_X R_Y B_X R_Y$ in this protocol.

3.1 Conditions for Security

For a protocol to be secure, the adversary should not be able to extract the message plaintext M. (Note that, in this paper, we only consider attacks on confidentiality, and not on other measures of security such as freshness and integrity. For example, we do not care if A obtains messages by intercepting them instead of eavesdropping. We are also not concerned by attacks in which A generates and injects new messages into the system impersonating another user.)

We call the entire key sequence applied to the plaintext in a message the *guard* of the message. The adversary should not be able to remove the entire guard if the message is secure.

We now provide an example of a successful attack. Suppose I and J are using the Diffie-Hellman protocol shown in the previous section. A breaks the security in the following way :

1. A captures the first message, $B_J R_I M$.
2. A uses the protocol to send this message to J as a new "request" and receives the corresponding "reply" in return.

$$\begin{aligned} A \rightarrow J &: A, B_J R_I M, J \\ A \leftarrow J &: J, B_A R_J B_A R_J B_J R_I M, A \\ &= J, B_A R_J B_A R_I M, A \end{aligned}$$

3. A applies the key sequence $B_I R_A B_J R_A$ to the payload, obtaining M.

This attack proves that the Diffie-Hellman protocol which we presented in the previous section is clearly not secure.

However, an interesting feature of cascade protocols is that there exist clear necessary and sufficient conditions for the security of a cascade protocol. These conditions are derived in [6]; we will briefly discuss them below.

$\Sigma_A = R_A \cup \{\forall X : B_X\}$ is the library of keys user A has access to.

$\Pi_A = \{\forall X : b_{AX}\}$ is the library of key sequences user A can indirectly cause to be applied to a message. This is because, if A sends a message M_1 as a request to I, it receives $b_{AI} M_1$ as the reply.

Communication between users I and J using protocol P:

$$\begin{aligned} I \rightarrow J &: I, a_{IJ} M, J \\ I \leftarrow J &: J, b_{IJ} a_{IJ} M, I \end{aligned}$$

is secure from attack by adversary A if (and only if) there exists no sequence a'_{IJ} with both the following properties:

- a'_{IJ} is composed only of keys and key sequences which the adversary A can cause to be applied to a message, i.e.

$$a'_{IJ} \in (\Sigma_A \cup \Pi_A)^*$$

- The simplified key sequence $a'_{IJ} a_{IJ}$ is the empty key sequence, i.e. $a'_{IJ} a_{IJ} M = M$.

(Note that there is no need to separately state that the adversary should also not be able to remove $b_{IJ}a_{IJ}$, the guard of the reply message. This requirement is covered by the non-existence of a'_{IJ} defined above. The reason for this asymmetry is that $b_{IJ} \in \Pi$, i.e. the adversary can apply b_{IJ} to a message. To see how, note that the adversary can send $a_{IJ}M$ to J, pretending to be I, and obtain $b_{IJ}a_{IJ}M$.

Now, consider the case that there exists a key sequence c_{IJ} such that the adversary can apply c_{IJ}, and $c_{IJ}b_{IJ}a_{IJ}M = M$. But in this case, there clearly exists a sequence a'_{IJ}: $a'_{IJ} = c_{IJ}b_{IJ}$. Thus, the condition that the adversary A should not be able to remove the guard of the reply is subsumed by the condition that A should not be able to remove the guard of the request.)

Protocol P is secure from adversary A iff, for all possible choices of I, J, and A (given A is not I or J), communication between users I and J using P is secure from attack by A.

In the next section, we study the security of two-step cascade protocols from both adversaries A and Z defined in the previous section.

4 Security of Two-Step Cascade Protocols

The actions available to the spoofing adversary A are a proper superset of the actions available to a non-spoofing adversary Z, so A is clearly at least as powerful as Z. In this section, we prove the very interesting result that the converse also holds: Z is as powerful as A, with respect to the goal of attacking a two-step cascade protocol.

Theorem 1. *A non-spoofing adversary and a spoofing adversary are equivalent in power with respect to the goal of breaking the confidentiality of a two-step cascade protocol.*

Proof. We begin our proof by noting that the spoofing adversary A is at least as powerful as the non-spoofing adversary Z. If protocol P cannot be broken by A, then both A and Z are (equally) ineffective attacking it.

To prove that Z is also as powerful as A, we demonstrate that any protocol P that can be broken by A can also be broken by Z.

The necessary and sufficient conditions to ensure that P cannot be broken by A are given by Theorem 1 of Dolev and Yao [6]:

1. a_{IJ} contains either B_I or B_J.
2. If b_{IJ} has R_J then it also has B_J.

As A can break P, one of these conditions must be false. We now demonstrate that, if either of these two conditions does not hold, then Z can extract message M from the message $a_{IJ}M$, breaking protocol P.

1. Consider the case that a_{IJ} has neither B_I nor B_J.
   In this case, a_{IJ} is composed of R_I.
   As B_I is a public key, it is available to Z. Thus Z can remove any guard composed of R_I.
   Hence Z can obtain M from the message $a_{IJ}M$.

2. The second possible case where P is insecure occurs when b_{IJ} is composed of R_J and B_I only.
   We begin by observing two facts.
   First, for any G_x, the keys in every sequence G_x are constrained to be R_I, B_I, or B_J. In particular, there are three possible values for the left-most key in G_x.
   Second, b_{ZJ} is simply b_{IJ}, with B_Z substituted for B_I. In other words, b_{ZJ} is composed of R_J and B_Z only. But Z has R_Z and B_J. Thus, if any subsequence of b_{ZJ} (including b_{ZJ} itself) occurs as the outermost sequence of keys in a guard, Z can remove this sequence.
   We now present the attack Z can use to compromise protocol P.
   Note that a_{IJ} is a key sequence of finite length (say, of length n).
   We introduce the symbols $G_n, \ldots G_1, G_0$. G_x means the suffix of a_{IJ}, which has length x. Thus $G_n = a_{IJ}$, ... G_0 is the empty sequence.
   Furthermore, G_0 is a proper suffix of G_1, G_1 is a proper suffix of G_2, etc.
   We show that, given any G_xM, $0 \leq x \leq n$, adversary Z can always remove at least one of the left-most keys to obtain G_yM, $0 \leq y \leq x$.
   Let the $k+1$ right-most elements in b_{IJ} be $\ldots R_J B_I^k$. ($k \geq 0$. Note that, for the protocol to be insecure, it is guaranteed that there must be at least one R_J in the key sequence b_{IJ}.)
   (a) If the left-most element in G_xM is R_I, then Z can remove this R_I since Z has B_I.
   (b) If the left-most element in G_xM is B_J, then Z first applies the key sequence R_Z^k to G_xM. Next, Z (stating its true identity as Z) initiates protocol P with J:

$$Z \rightarrow J : Z, R_Z^k G_x M, J$$
$$Z \leftarrow J : J, b_{ZJ} R_Z^k G_x M, Z$$

   b_{ZJ} is the same as b_{IJ}, except that each occurrence of B_I is replaced by B_Z. In particular, its right-most $k+1$ elements are $R_J B_Z^k$. This sequence cancels out the $k+1$ leftmost elements in $R_Z^k G_x M$ (recall these elements were $R_Z^k B_J$).
   Adversary Z then removes all remaining elements of b_{ZJ} forming the outermost key sequence in the reduced $b_{ZJ} R_Z^k G_x M$. The resulting G_yM is shorter (at least one element shorter) than G_xM.
   (c) If the left-most element in G_xM is B_I, then by symmetry Z simply follows the same attack detailed in the item above. Z applies R_Z^k, then initiates protocol P with I, and so on. (Note that this algorithm removes B_K for any K that Z communicates with. In the previous item, we used $K = J$; here $K = I$. The working is identical.)
   This concludes our proof of the fact that Z can always remove at least the left-most element from the guard G_x of G_xM.
   But the length of the original guard G_n is finite. By well ordering, it is not possible to have an infinite chain of the form $G_nM, G_{n-1}M, G_{n-2}M \ldots$ without eventually reaching G_0M, i.e. M. Hence, at some point, Z will obtain M.

Hence, we conclude that Z is as powerful as A in compromising the confidentiality of two-step cascade protocols.

5 Security of k-Step Cascade Protocols

In this section, we generalize our study of two-step cascade protocols to k-step cascade protocols where $k \geq 2$. Such protocols consist of repeatedly sending messages back and forth between two users. (Note that we assume the traditional "ping-pong" model of cascade protocols, where only two users pass the message back and forth and apply layers of encryption and decryption.)

We begin by defining the following protocol used by I and J to securely communicate the confidential message plaintext M.

$$
\begin{aligned}
&I \rightarrow J : I, g^1_{IJ} M, J \\
&I \leftarrow J : J, g^2_{IJ} g^1_{IJ} M, I \\
&I \rightarrow J : I, g^3_{IJ} g^2_{IJ} g^1_{IJ} M, J \\
&\qquad \ldots \\
&I \rightarrow J : I, g^k_{IJ} g^{k-1}_{IJ} \ldots g^1_{IJ} M, J
\end{aligned}
$$

The initial step consists of encrypting the plaintext with a key sequence (in this case g_1) and sending the result to another user. In each subsequent step, a key sequence is applied to the entire message received in the step before. The result of the operation is sent to the other user.

This subsequent step is repeated until the total number of messages reaches k. Note that, although in the example k is odd, in general k may also be even, in which case the final message is of the form

$$I \leftarrow J : J, g^k_{IJ} g^{k-1}_{IJ} \ldots g^1_{IJ} M, I$$

For convenience, we assume for the remainder of this section that the two legitimate users communicating are always I and J. This allows us to use the clean notation $g_1, g_2 \ldots$ as shorthand for the hard-to-read $g^1_{IJ}, g^2_{IJ} \ldots$.

If we consider any two consecutive steps (say steps l and $l+1$) of the protocol, they must be of one of the two forms

$$
\begin{aligned}
&I \rightarrow J : I, g_l \ldots g_1 M, J \\
&I \leftarrow J : J, g_{l+1} g_l \ldots g_1 M, I
\end{aligned}
$$

or

$$
\begin{aligned}
&I \leftarrow J : J, g_l \ldots g_1 M, I \\
&I \rightarrow J : I, g_{l+1} g_l \ldots g_1 M, J
\end{aligned}
$$

The first form results when l is odd, and the second when l is even.

The first form is obviously a two-step cascade protocol in its own right, with $a_{IJ} = g_l...g_1$ and $b_{IJ} = g_{l+1}$. For the second form, rewriting it as

$$J \rightarrow I : J, g_l...g_1 M, I$$
$$J \leftarrow I : I, g_{l+1} g_l...g_1 M, J$$

shows clearly that it is also a two-step cascade protocol with $a_{JI} = g_l...g_1$ and $b_{JI} = g_{l+1}$. (Note that in this case, J issues the request and I the reply; thus, we use a_{JI} instead of a_{IJ} and b_{JI} instead of b_{IJ}.)

From these observations, we can conclude the following lemma :

Lemma 1. *Any two consecutive steps of a k-step cascade protocol constitute a valid two-step cascade protocol.*

The set of all two-step cascade protocols "contained" in a k-step cascade protocol P is called the *decomposition* of P. There are $k-1$ such two-step cascade protocols, consisting of steps 1 and 2, 2 and 3 ... $k-1$ and k.

The question naturally arises as to how the security of a k-step cascade protocol is related to the security of the elements of its decomposition. We now show the very interesting result that, if we define a set of protocols to be secure iff all the member protocols in the set are secure, then the security of a k-step cascade protocol P and the security of its decomposition $P_{\{\}}$ are equivalent.

Theorem 2. *A k-step cascade protocol P is secure iff every two-step cascade protocol in its decomposition is secure.*

Proof. We need to prove both directions - if and only if. To prove security, we will again employ Theorem 1 of Dolev and Yao [6], which states that a cascade protocol is secure iff it satisfies the following two conditions:

1. The first key sequence applied to the message plaintext (i.e. g_1) must, in its simplified form, contain either B_I or B_J.
2. All key sequences that are subsequently applied must, in their simplified forms, contain B_I if they contain R_I and B_J if they contain R_J.

The decomposition of protocol P is the set

$$P_{\{\}} = \{P_1, P_2, \ldots P_{k-1}\}$$

where P_l is the two-step cascade protocol formed by steps l and $l+1$ of P.

To prove the "if" direction, suppose P is insecure even though $P_{\{\}}$ is secure. As P is insecure, it must violate at least one of the two conditions given above.

1. Consider the case that P violates the condition 1 above. In this case, the security is broken in the first step of P. As $P_{\{\}}$ is known to be secure, P_1 is secure.
   g_1 is the guard of the message in the first step of the secure protocol P_1. Thus by condition 1, g_1 (in its simplified form) contains either B_I or B_J.
   g_1 is also the first key sequence applied to the message plaintext in P. Hence the first guard applied to M in P contains B_I or B_J, i.e. P obeys condition 1. Thus we obtain a contradiction.

2. Suppose the security of P is broken in step no. l, where $l > 1$. (The first step has been proved secure in the above item.) If it is broken in more than one step, we choose the smallest l. If P violates condition 1, then the first step is broken; as in this case we consider $l > 1$, it follows that P is insecure because it does not satisfy the second condition of the theorem.
   (a) If l is odd.
   In this case, the simplified form of g_l contains R_I and not B_I.
   The second step of P_{l-1} involves applying the key sequence g_l. We know that the simplified form of g_l contains R_I and not B_I. Hence we see that P_{l-1} violates condition 2 - in other words, P_{l-1} is insecure. But we started with the assumption that $P_{\{\}}$ is secure, which of course requires that the protocol P_{l-1} is secure. Hence we have a contradiction.
   (b) If l is even.
   This case is exactly analogous to the one above.
   The simplified form of g_l contains R_J and not B_J. In other words, P_{l-1} is insecure. But P_{l-1} is known to be secure, as $P_{\{\}}$ is secure. Thus, we obtain a contradiction.

As the assumption that P is insecure but its decomposition $P_{\{\}}$ is secure always leads to a contradiction, it must be impossible. Hence we conclude that, if $\{P_1, P_2, \ldots P_{k-1}\}$ is secure, P must also be secure.

To prove the "only if" direction, suppose P is secure even though $P_{\{\}} = \{P_1, P_2, \ldots P_{k-1}\}$ is insecure.

Let l be the smallest value such that P_l is insecure. There are two ways in which this can happen: violation of the first and of the second conditions of the theorem above.

1. If P_l violates the first condition.
   (a) Consider the case $l = 1$.
   P is known to be secure.
   g_1 is the guard of the first message in secure protocol P.
   Hence, by condition 1, g_1 (in its simplified form) contains either B_I or B_J.
   g_1 is also the guard of the first message in P_1. As g_1 in its simplified form contains either B_I or B_J, P_1 does not violate condition 1. This contradicts our initial assumption that P_1 violates the condition.
   (b) Consider $l > 1$.
   P_l is insecure because $g_l ... g_1$ contains neither B_I nor B_J.
   In this case, the guard $g_l ... g_1$ is composed of only R_I and R_J. In other words, an adversary can capture message $g_l ... g_1 M$ and completely remove the guard to obtain plaintext M, using only the keys B_I and B_J (which are public keys).
   This means that protocol P is also insecure as it has the message $g_l ... g_1 M$. But protocol P is known to be secure - a contradiction.
2. If P_l violates the second condition.
   This means that g_{l+1} contains R_I but not B_I, or R_J but not B_J.

(a) If l is even (so $l+1$ is odd).
g_{l+1}, which is applied by I, contains R_I but not B_I.
As $l+1 > 1$, g_{l+1} is not the first key sequence applied to M in protocol P. (g_1 is this first key sequence.)
Consequently, if g_{l+1} contains R_I but not B_I, protocol P violates condition 2 of Dolev and Yao, so it is insecure. But P is known to be secure - a contradiction.

(b) If l is odd.
This case is exactly analogous to the one presented above.
g_{l+1} contains R_J but not B_J.
g_{l+1} is not the first key sequence applied to M in protocol P.
P is thus proven insecure by the Dolev-Yao theorem. This contradicts our initial assumption that P is secure.

Therefore, the assumption that P is secure but its decomposition is insecure always leads to a contradiction and must be impossible. Hence, given that P is secure, $\{P_1, P_2, \ldots P_{k-1}\}$ is also secure.

As we have proved both the "if" and the "only if" directions, we conclude our proof.

It is now simple to prove our main result.

Theorem 3. *A non-spoofing adversary and a spoofing adversary are equivalent in power with respect to the goal of breaking the confidentiality of a two-step cascade protocol.*

Proof. A set of protocols is secure iff every protocol in the set is secure. By duality, we define a set of protocols to be broken by an adversary iff at least one of the protocols in the set is broken by the adversary.

By Theorem 2, if (and only if) P can be broken by an adversary A, so can its decomposition $P_{\{\}}$. Let us consider that this successful adversary is a spoofing adversary A.

As $P_{\{\}}$ is broken, we know that there must exist at least one member protocol $P_l \in P_{\{\}}$ such that P_l is vulnerable to A. (If there are multiple vulnerable members, we can choose any one at random).

But P_l is a two-step cascade protocol. By Theorem 1, P_l (and hence $P_{\{\}}$) is also broken by a non-spoofing adversary Z.

From Theorem 2, we conclude that P is also broken by Z.

As Z can break the confidentiality of protocol P in every case where A can, Z is at least as powerful as A. It is known that A is at least as powerful as Z. From these two statements, we conclude that A and Z have equivalent power. This concludes our proof.

6 Conclusion

This paper presents a novel result concerning the security of cascade protocols, as defined in the landmark paper of Dolev and Yao [6]. In the original definition

of an "active" adversary, the adversary is defined as having two novel powers: impersonation, and altering or replaying messages. (This is in addition to the powers of a "passive" adversary, i.e., eavesdropping and applying its own keys to captured messages). As there now exist effective means to prevent spoofing, we studied the effect on an active adversary if it is made unable to spoof. The result was completely unexpected: if the aim of the adversary is to compromise confidentiality, then a guarantee of no spoofing does not reduce the power of the active adversary to compromise cascade protocols. As a component of our proof of this theorem, we also obtain the independently interesting theorem that any k-step cascade protocol can be decomposed into a set of two-step protocols, and this set is equivalent in security to the original protocol.

We believe this result to be interesting because, while it is directly applicable only to one particular class of protocols (namely cascade protocols), it allows us to carry on a discussion of the powers of different kinds of adversaries. Clearly, the scope for further research extends in two directions. In the first place, it would be interesting to explore the relative power of adversaries that are restricted from using one or more of the powers mentioned above. The other question raised by our research is how the protocol model could be strengthened. Starting with the simple model of cascade protocols, if we use more general protocol models, at what point does the ability to spoof, for example, become non-redundant? Thus, our adoption of a simple protocol model exposes several interesting lines for further inquiry. In contrast, previous work, which began by assuming stronger models such as namestamp protocols, yielded mostly negative results. For example, it has been proven that namestamp protocols have no simple test for security [3].

We propose, as an open problem, the generalization of this discussion (regarding the powers of different adversaries) to include more practical protocols. Stated as a question, "How can the protocol model be strengthened so that it remains possible to derive interesting results about the power of the adversary, but the family of protocols covered by the model becomes broad enough to include protocols which are in practical use?" In our immediate future work, we intend to explore one such stronger model: we are studying whether our results continue to hold when cascade protocols are generalized to allow more than two parties [8].

References

1. Baker, F.: Requirements for ip version 4 routers. RFC 1812 (1995)
2. Bellovin, S.: Icmp traceback messages. Internet Draft: draft-bellovin-itrace-00.txt (2000)
3. Book, R., Otto, F.: On the security of name-stamp protocols. In: Third Conference on Foundations of Software Technology and Theoretical Computer Science, vol. 39, pp. 319–325 (1985)
4. Chang, H., Narayan, R., Wu, S., Vetter, B., Wang, X., Brown, M., Yuill, J., Sargor, C., Jou, F., Gong, F.: Deciduous: decentralized source identification for network based intrusions. In: Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management (1999)

5. Diffie, W., Hellman, M.: Multiuser cryptographic techniques. In: Proceedings of the AFIPS (1976)
6. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (1983)
7. Ferguson, P., Senie, D.: Network ingress filtering: Defeating denial of service attacks which employ source ip address spoofing. RFC 2827 (2000)
8. Goldreich, O.: On the security of cryptographic protocols and cryptosystems. D.Sc. Thesis, Technion. (1983)
9. Gouda, M.G., Elnozahy, E., Huang, C., McGuire, T.: Hop integrity in computer networks. In: Proceedings of the 8th IEEE International Conference on Network Protocols (2000)
10. Kent, S., Atkinson, R.: Security architecture for the internet protocol. RFC 2401 (1998)
11. Li, J., Mirkovic, J., Wang, M., Reiher, P., Zhang, L.: Save: Source address validity enforcement protocol. In: Proceedings of IEEE INFOCOM (2002)
12. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. Communications of ACM 2, 993–999 (1978)
13. Park, K., Lee, H.: On the effectiveness of probabilistic packet marking for ip traceback under denial of service attack. In: Proceedings of IEEE INFOCOM (2001)

Implementing Fault-Tolerant Services Using State Machines: Beyond Replication

Vijay K. Garg*

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA

Abstract. This paper describes a method to implement fault-tolerant services in distributed systems based on the idea of fused state machines. The theory of fused state machines uses a combination of coding theory and replication to ensure efficiency as well as savings in storage and messages during normal operations. Fused state machines may incur higher overhead during recovery from crash or Byzantine faults, but that may be acceptable if the probability of fault is low. Assuming n different state machines, pure replication based schemes require $n(f+1)$ replicas to tolerate f crash faults in a system and $n(2f+1)$ replicas to tolerate f Byzantine faults. For crash faults, we give an algorithm that requires the optimal f backup state machines for tolerating f faults in the system of n machines. For Byzantine faults, we propose an algorithm that requires only $nf+f$ additional state machines, as opposed to $2nf$ state machines. Our algorithm combines ideas from coding theory with replication to provide low overhead during normal operation while keeping the number of copies required to tolerate f faults small.

1 Introduction

The replicated state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. This approach proposed by Lamport in [1,2] and further elaborated by Schneider in [3] is considered the standard solution to the problem of fault-tolerance in distributed systems. Note that replication has been considered wasteful in the context of fault-tolerance of data (in communication and storage) for many decades, but in the distributed systems replication continues to be the dominant approach for fault-tolerance [4]. In this paper, we give an alternate method for fault-tolerance that combines ideas from replication with coding theory [5,6] to get main advantages of both the approaches. We use (sufficient) replication to guarantee low overhead during normal operations and coding theory to reduce the number of copies to get space and message savings.

* This research was supported in part by the NSF Grants CNS-0718990, CNS-0509024, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 450–464, 2010.

We depart from the standard model of fault-tolerance in distributed systems in which the problem is to tolerate faults in functioning of a single state machine. We will be concerned with fault-tolerance in a *set* of state machines where the size of the set will usually be greater than one. While this assumption makes the problem different from the usual set-up, we argue that our set-up is practically useful. Any large system is generally constructed as a set of state machines rather than a single monolithic state machine. Even when the server is constructed as a single state machine, it is quite natural to have multiple instances of the state machines deployed for different departments of the organization.

In this paper, we show how services in a distributed system can be made fault-tolerant using fusion. Given n *different* state machines running on different servers, we focus on tolerating f faults. We focus on two types of faults: *crash* faults and *Byzantine* faults. For crash faults, faulty state machines lose their state. We assume that crash faults are detectable and the problem that remains is to recover the lost state of state machines. For Byzantine faults [7], the state machine may go to an incorrect state spontaneously and the algorithm must continue to provide correct responses to the client in spite of these faults.

For *crash* faults, we give a technique to construct additional f state machines (called fused state machines) such that the system of $n + f$ machines can tolerate crash of any f machines in the system. We illustrate our technique on the resource allocation service from [3], a causal ordering algorithm [8] and a distributed mutual exclusion algorithm [9]. The fused state machines use a combination of erasure coding and replication to ensure that during normal operations, the message and computation overhead on primary state machines is close to that for replicated state machines. The updates of fused state machines are made efficient using linearity of erasure coding scheme employed and sufficient replication.

For *Byzantine* faults, the problem of detection is harder from the perspective of computation and communication complexity. Here we use a hybrid of replication and coding theory. In particular, we give an algorithm that keeps the overhead of the replicated state machine approach during normal operations but requires only $nf + f$ additional state machines (as opposed to $2nf$ state machines). Our algorithm is based on the following observation that if there are $f + 1$ copies of a state machine, then at least one of them is correct. In case of a fault, we only need to determine which of these copies is correct. The traditional method of keeping $2f + 1$ copies (and then using majority) is wasteful for the task. We introduce the notion of *efficient liar detection* based on fused state machines. This allows us to prove the following main result in this paper (in Section 3). Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $nf + f$ state machines that can tolerate f Byzantine faults and has the same overhead as the Replicated State Machine approach during the normal operation and additional $O(mf + nt^2)$ overhead during recovery where t is the actual number of faults that occurred in the system.

In our earlier work, we have given algorithms for fusible data structures. In particular, [10] gives algorithms for arrays, stacks, queues, linked lists etc. to handle *crash* faults. This work has been generalized to tolerate multiple *crash* faults in [11]. In contrast, the goal of the current work is to focus on the differences between the replicated state machine approach and the fused state machine approach and also tackle *Byzantine* faults. Furthermore, we show that our approach is applicable to many distributed algorithms including a causal ordering algorithm [8], and Ricart and Agrawala's mutual exclusion algorithm [9]. For both of these algorithms, we get n-fold savings in space. We also get savings in messages for Ricart and Agarwala's algorithm because of aggregation that is possible in fused state machines. In [12], an algorithm has been provided to generate fused *finite* state machines. That algorithm assumes that the state space of the primary machines is finite. In this paper, techniques are suitable even for infinite state space.

In data storage and communication, coding theory is extensively used to recover from faults. For example, RAID disks use disk striping and parity based schemes (or erasure codes) to recover from the disk faults [13,14,15]. As another example, network coding [16,17] has been used for recovering from packet loss or to reduce communication overhead for multicast. In these applications, the data is viewed as a set of data entities such as disk blocks for storage applications and packets for network applications. Coding theory techniques [6] are oblivious to the structure of the data. The details of actual operations on the data are ignored and the codes are simply recomputed after any write update. To tolerate crash failures for servers, one can view the memory of the server as a set of pages and apply coding theory to maintain code words. This approach, however, may not be practical because a small change in data may require recomputation of the backup for one or more pages. This results in a high computational and communication overhead. We show in this paper that with data structure-aware programming and partial state replication, backup machines can be designed so that they provide fault-tolerance in an efficient manner.

2 Fusible State Machines

There are n *deterministic* primary state machines, $P(i)$, where i ranges from 1 to n. Each state machine receives an input from the client (or environment). On receiving the input, the state machine applies the state transition function to change its state. The set of states and inputs may be infinite.

We require state machines to be deterministic just as required by the replicated state machine approach. Given the state of a machine and the sequence of inputs, the behavior of the state machine is required to be unique. This assumption is crucial in both the replicated state machine (RSM) and the fused state machine (fused-SM) approaches.

Throughout this paper we assume that channels are reliable and FIFO and that there is a fixed upper bound for all message delivery. We also assume that crashes of processes are reliably detected.

2.1 Event Counter

To concretize our discussion, we start with n simple state machines, $P(i)$'s, shown in Fig. 1. Each of these n machines accept two types of input: $entry(v)$ and $exit(v)$. These state machines may, for example, be counting the number of people of type i entering a room. Each state machine has a variable *count* with domain as non-negative integers. When $P(i)$ receives an event $entry(v)$, it increments its count if v is equal to i and decrements it when it receives a similar $exit(v)$ event.

$P(i) :: i = 1..n$ int $count_i = 0$; On event $entry(v)$: **if** $(v == i)$ $count_i = count_i + 1$; On event $exit(v)$: **if** $(v == i)$ $count_i = count_i - 1$;	$F(j) :: j = 1..f$ int $fCount_j = 0$; On event $entry(i)$, for any i $fCount_j = fCount_j + i^{j-1}$; On event $exit(i)$ for any i $fCount_j = fCount_j - i^{j-1}$;

Fig. 1. Fusion of Counter State Machines

To tolearte f faults, the Replicated State Machine (RSM) approach requires f additional state machines for each of $P(i)$ resulting in the total of nf additional state machines. For fusion, we add just f additional machines, $F(1)..F(f)$ as shown in Fig. 1. $F(j)$ increases its count by i^{j-1} for any event $entry(i)$ and decrements by the same amount for $exit(i)$. It can be seen that the fused-SM $F(1)$ tracks the sum of all counts. It increments the variable $fCount_1$ on $entry(i)$ for any i and decrements it for any $exit(i)$. $F(2)$ maintains $fCount_2 = \sum_i i * count_i$. More generally, $fCount_j = \sum_i i^{j-1} * count_i \quad for\ all\ j = 1..f$.

The recovery procedure for fusible SM is more complex than for replication. It crucially depends on the fact that if any f machines crash, the rest of the machines are still available. $F(1)$ is sufficient to recover from one crash fault. If $P(c)$ has failed, then its state $count_c$ can be recovered as $fCount_1 - \sum_{i \neq c} count_i$. In general, we can recover states of any f failed state machines using the remaining machines. For example, consider the case when f is two and the machines that crashed are $P(c)$ and $P(d)$. Using fusion machine $F(1)$ and remaining counts we can get the value of $count_c + count_d$. Using $fCount_2$, we can get the value of $c * count_c + d * count_d$. We have two linearly independent equations in two variables which can be solved to get the values of $count_c$ and $count_d$. More generally, recovery from f faults reduces to solving f linearly independent equations in f variables.

A reader well-versed in coding theory would realize that if $(count_1, count_2, , count_n)$ is viewed as data, $(count_1, count_2, ..count_n, fCount_1, fCount_2, ..fCount_f)$ can be viewed as a code word. The code word obtained is equivalent to one obtained by multiplying data vector by the identity matrix adjoined with the transpose of the Vandermonde matrix [5]. The unique solvability of all the counts is easy to show; the proof is given in [18] for completeness sake.

Theorem 1. *Suppose* $\mathbf{x} = (count_1, count_2, , count_n)$ *is the state of the primary state machines. Assume* $fCount_j = \sum_i i^{j-1} * count_i$ *for all* $j = 1..f$. *Given any* n *values out of* $\mathbf{y} = (count_1, count_2, ..count_n, fCount_1, fCount_2, ..fCount_f)$ *the remaining values in* $\mathbf{x}$ *can be uniquely determined.*

It is important to note the distinction between a server and a state machine. In the event counter example, to tolerate f crash faults among n state machines, the RSM approach need not run all nf on distinct servers. We could, for example, run one copy of each of the state machines for all $P(1)..P(n)$, on one server. Thus, the number of servers required to tolerate f crash faults can still be considered to be f for the RSM approach. However, the fused state machine approach provides upto n-fold savings in the space required for keeping backups. We now show that the fused-SM approach also yields benefits in computation and communication when events are shared between primary state machines. Suppose that each $P(i)$ has an additional event called *incr* which increases $count_i$ by 1. When the event *incr* happens, all P_i increment their counts. In the RSM approach the event *incr* would be communicated to nf state machines, and will be executed nf times. In the Fused-SM approach, we require $F(j)$ to execute *incr* as $fCount_j = fCount_j + \sum_{i=1}^{i=n} i^{j-1}$. The total number of events that are executed is exactly f, one for each fused-SM. Thus, when events are shared across primary state machines, we get the advantage of *aggregation* thereby reducing the message and computation complexity for backup.

Note that we do not require the fused-SMs to be synchronized with primary state machines. The only requirement is that all updates from primary state machines are applied in the same order at all the fused-SMs. The messages at fused-SMs may be buffered because the primary state machines never wait for fused-SMs to finish their updates. In case of a failure of a primary machine, all the pending updates at the fused-SMs must be applied before the recovery.

So far we had assumed that by adding numbers we do not get overflow. If overflow is possible, there are two approaches to tackle it. The first approach is to do all the arithmetic, i.e. addition (subtraction), and multiplication (division) in finite Galois field as typically done in coding theory [5]. In that case the matrix G can either be chosen as a Cauchy Matrix or a Vandermonde matrix reduced using elementary transformations so that the first n rows form an identity matrix [19]. The other possibility is to guarantee that there is never any overflow in any computation. This can be done, for example, by using `BigInteger` package in Java.

2.2 Causal Ordering

We now generalize the Event Counter example to primary state machines that contain not one variable but a set of data structures. Whenever a primary state machine receives an event from a client and updates it data structures, it also sends a message to the fused state machines with the list of variables and the incremental change in their values. We illustrate this method for a *causal ordering* algorithm [20] in a group of n processes.

Consider the version described by Raynal, Schiper, and Toueg [8]. Each process maintains a matrix M of integers. The entry $M[q, r]$ at $P(i)$ records the number of messages sent by process $P(q)$ to process $P(r)$ as known by process $P(i)$. Whenever a message is sent from $P(i)$ to $P(r)$, the matrix M is piggybacked with the message. A message is eligible to be received when the number of messages sent from any process $P(q)$ to $P(i)$, as indicated by the matrix W received in the message, is less than or equal to the number recorded in the matrix M.

Suppose, we would like the system to be able to tolerate f crash faults, i.e., recover matrices for processes that have crashed. We require $P(i)$'s to send an "M-Update" message with incremental changes in entries of the matrix to the fused processes $F(j)$. Instead of maintaining f copies of the matrix for each primary process, the fused-SM algorithm requires a single (fused) matrix for every fault. Thus, the storage requirement for fused processes is $O(fn^2)$ as opposed to $O(fn^3)$ required by a replication based algorithm. A similar algorithm can be used to recover vector clocks of faulty processes in distributed systems.

2.3 Resource Allocator

The technique outlined in previous section may not be practical when a simple change in data structure results in a significant change in the state. We show that by analysis of the data structure, and by selective replication the size of the messages from primary messages to fusion processes can be reduced significantly.

To illustrate this point, we apply the method of fusion to the resource allocator state machine in [3]. Assume that there are n different type of resources that can only be used in mutually exclusive fashion. The state machine $P(i)$ shown in Fig. 2 handles clients requesting resource i. It maintains two variables: *user*, an integer which records the current user of the resource if any, and *waiting*, a queue of integers which stores the id's of clients waiting for the resource.

user: int initially 0;// resource idle	**On receiving** release
waiting: queue of int initially null;	if (*waiting*.isEmpty())
On receiving acquire from client *pid*	user = 0;
if (*user* == 0) {	else { user = *waiting*.head();
send(OK) to client *pid*; *user* = *pid*;}	send(OK) to *user*;
else *waiting*.append(*pid*);	*waiting*.removeHead(); }

Fig. 2. Resource Allocator State Machine from [3] $P(i) :: i = 1..n$

Suppose that we want to tolerate one fault in any of these n machines. Whenever, the variable *user* changes we can send the incremental change to fusion processes. But, what should we do about the waiting list? If we view the bit representation of waiting list as an integer (a big integer), then computing the code at fusion processes after every change would be very inefficient. We use the technique from fusible data structures[10]. Instead of sending the change in

state, we send the event that allows the fused structure to be maintained efficiently. The primary state machine that uses fused-SM approach is shown in Fig. 3. Whenever any data structure changes, it sends to the fused machines the change that needs to be made in the data structure in a manner that is tailored to the data structure. Note that the primary machine does not send the changed queue or even the incremental difference from the old queue and the new queue. It only sends enough information so that the fused queues can carry out the state change.

$P(i) :: i = 1..n$
On receiving acquire from client pid
if ($user == 0$){send(OK) to client pid;
    $user = pid$;
    send(USER, i, $user$) to $F(j)$'s;}
else { $waiting$.append(pid);
    send(ADD-WAITING,i, pid) to $F(j)$'s;}

On receiving release
if (waiting.isEmpty()){$olduser = user$;
    $user = 0$;
    send(USER, i, $user - olduser$) to $F(j)$ }
else { $olduser = user$;
    $user = waiting.head()$;
    send(OK) to $waiting.head()$;
    $waiting.removeHead()$;
    send(USER, i, $user - olduser$) to $F(j)$'s
    send(DEL-WAITING, i, $user$) to $F(j)$'s
}

$F(j) :: j = 1..f$
    $fuser$:int initially 0;
    $fwaiting$:fused queue initially 0;

On receiving (USER, i, val)
$fuser = fuser + i^{j-1} * val$;

On receiving (ADD-WAITING, i, pid)
$fwaiting.append(i, pid)$;

On receiving (DEL-WAITING, i, $user$)
$fwaiting.deleteHead(i, user)$;

Fig. 3. Algorithm A: Fused State Machine for Resource Allocation

The code for the fused-SM is shown in Fig. 3. In $F(j)$ we have used $fwaiting$ as a fused queue. For simplicity, we use a circular array based implementation (a linked list based implementation is in [10]).

The above method has reduced the number of backup state machines nf to f and yet it can tolerate any f faults from $P(1)..P(n)$. The recovery process is more complex than replication but the significant savings (n-fold) in the reduced number of active state machines may justify this added complexity especially when the probability of faults is small.

Remark: So far we had assumed that the clients interact only with the primary machines which, in turn, interacted with fusion machines to keep them up-to-date. In many examples, an alternate design is possible in which the commands to the primary state machines are also issued to the fused-SMs. The pseudo-code for such a design is shown in [18].

We now do overhead analysis for both RSM and the fused-SM approach.

Overhead Under Normal Operation: For replication, we require additional nf state machines, f replicas for each of the primary state machine. Each operation

```
fQueue: array[0..M − 1] of int initially
0;
head, tail, size:array[1..n] of int init 0;

append(i, v):
if (size[i] == M)
    throw Exception("Full Queue");
fQueue[tail[i]] = fQueue[tail[i]] + i^(j−1) ∗
v;
tail[i] = (tail[i] + 1)%M;
size[i] = size[i] + 1;
```

```
deleteHead(i, v):
if (size[i] == 0)
    throw Exception("Empty Queue");
fQueue[head[i]] = fQueue[head[i]] −
i^(j−1) ∗ v;
head[i] = (head[i] + 1)%M;
size[i] = size[i] − 1;

isEmpty(i):
return (size[i] == 0);
```

Fig. 4. Fused Queue Implementation at $F(j)$

requires a message to the primary state machine and f replicas. For fused-SM approach, we require additional f machines. Each operation still requires $f + 1$ messages, one to the primary state machine and f messages from the primary to fused-SMs. The message to the primary state machine is same as for the RSM approach, however messages to the fused-SMs may contain additional state information so that fused machines can execute the event despite availability only of fused data structures.

Assume that the waiting list can have size at most $O(m)$. The RSM approach requires $O(nfm)$ space to tolerate f faults among n machines. The fused-SM approach requires $O(fm+nf)$ space. The component $O(nf)$ is required because we allow $O(1)$ state information for each of the n state machines at the fused-SMs. In the example, we kept $head[i], tail[i]$ and $size[i]$ for each state machine.

The number of events and messages required to be processed at the fused-SM is n times more than the number of events processed by a replica. Thus, if n is large the fused-SMs may become bottleneck. In these cases, one could easily use a hybrid of replicated and fused-SM approach.

Complexity for Recovery after Failure: The RSM approach has minimal overhead for recovery after failure. As soon as a primary machine is detected to be crashed, the replica with the highest id that survives can take over and start functioning as primary.

The recovery overhead in the fused-SM approach is crucially dependent on the number of actual faults t. Let the state of any primary state machine be $O(m)$. First consider the case when t equals 1. The recovery algorithm will require $O(n)$ messages, one from each of the surviving machines of size $O(m)$. It will take $O(nm)$ time to recover the state of the crashed machine. For $t > 1$ faults, we would be required to solve t linearly independent equations. Equivalently, it can be viewed as multiplying the fusion vector with the inverse of the equation matrix. Since m is large compared to t, we ignore the one time cost of computing the inverse. Thus, we get the overall cost as $O(m(nt + t^2))$.

2.4 Application to Ricart and Agrawala's Algorithm

The state machine for the resource allocator example was based on a centralized algorithm for mutual exclusion. We now show that the technique is also applicable to distributed algorithms such as Ricart and Agarwala's algorithm[9]. Suppose that there are n primary processes $P(1)..P(n)$ that are coordinating access to a *single* critical section. For the RSM based approach, each P_i would need f backups and will result in nf additional state machines (even if they are run on only f additional servers). Since each state machine requires $O(n)$ space to keep track of pending requests, the total space requirement is $O(fn^2)$. The code for the fused-SM based Ricart and Agrawals's algorithm is presented in [18]. With the fused-SM approach, we use f additional state machines with total of $O(fn)$ space. Any request message is also sent to the fused processes which update the fused data structures on behalf of all the processes in the system. Similarly, *okay* messages are also sent to the fused processes.

The non fault-tolerant algorithm requires $2n$ messages per CS invocation. With the RSM approach, every message needs to be sent to f backup processes resulting in $2n(f+1)$ messages. The Fused-SM approach requires an additional request message and $n-1$ okays to be sent to any fused process. Thus, the total message requirement is only $2n+nf$, which results in savings of nf messages.

3 Byzantine Faults

So far we had assumed crash faults. We now discuss Byzantine faults where any state machine may change its state arbitrarily. The RSM approach requires that there be $2f$ backup replicas for each primary state machine. Since there are $2f+1$ values available, even if f of them are faulty, the majority will always be correct. When this approach is applied to n different servers, the RSM approach requires additional $2nf$ replicas. For data coding, it is well known that by appending $2f$ parity check symbols, one can recover from f unknown data errors [6]. Can the same ideas be applied to fault-tolerance of state machines?

The additional constraint we have for tolerating Byzantine faults in state machines is that during normal (fault-free) operation, we would like to have as little overhead as possible. Specifically, we would like to avoid the overhead of decoding the state during normal operations. To achieve this goal, we give an algorithm that combines replication with coding theory. We first consider the case of a single Byzantine fault. Next we generalize the algorithm to tolerate f Byzantine faults but assume that each state machine has $O(1)$ state. Finally, we give the algorithm that tolerates f Byzantine faults and each primary state machine may have $O(m)$ state.

3.1 Tolerating Single Byzantine Fault

We start with the case of detecting and tolerating a single Byzantine fault among n primary state machines. The pure RSM approach requires two replicas for

every primary machine resulting in $3n$ state machines in all. The pure Fused-SM approach would require $n + 2$ machines in all. However, in the pure Fused-SM approach, even the normal operations may be inefficient. For crash faults, the decoding was required only when there was a failure, a low probability event. For Byzantine faults, a pure Fused-SM approach would require decoding even during normal operations just to detect if one of the primary machines is faulty. We now show a hybrid approach that is efficient during normal operation and still requires less number of processes than the RSM approach.

Our algorithm is based on two observations. First, if we have two copies of a primary state machine $P(i)$, then one of these copies is guaranteed to be correct. The RSM approach relies on keeping an additional copy so that majority can be used to determine which is correct. In our approach, we use the concept of *liar detection*. We use the fused-SMs to determine which of the two copies is faulty. The liar detection approach is more efficient in terms of the total number of copies required. The second observation we use is that if two copies of $P(i)$ agree on some value, then that value is guaranteed to be correct (because, there can be at most one Byzantine fault).

Theorem 2. *Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $n + 1$ state machines that can tolerate a single Byzantine fault and has the same overhead as the RSM approach during normal operation and additional $O(m + n)$ overhead during recovery.*

Proof. We keep one replica $Q(i)$ for every primary state machine $P(i)$ and a fused-SM $F(1)$ for the entire system. Thus, we keep $2n + 1$ state machines in all. During normal operation (when there is no fault), the value of any output at $P(i)$ and $Q(i)$ must be identical. In this case, we do not decode the value from $F(1)$. As soon as $P(i)$ and $Q(i)$ differ for any i, we have detected Byzantine fault in the system. Note that we do not observe the state of $P(i)$ and $Q(i)$ at all events. We only look at the response of $P(i)$ and $Q(i)$ for input events and take action when the response (output) at $P(i)$ differs from $Q(i)$. At this point, we know that either $P(i)$ is correct or $Q(i)$ is correct, but do not know the identity of the liar yet. We now invoke the liar detection algorithm as follows. Given the state of $P(i)$ and $Q(i)$, in $O(m)$ time we can locate the first data of size $O(1)$ that is different in them. We use the fused process $F(1)$ to determine which of these values is correct. This step will require messages of size $O(1)$ from other $n-1$ primary processes. It also requires that the system ensures that all operations that have been performed on the primary state machines have been applied to $F(1)$. Now, in $O(n)$ time the correct value of the data can be determined; therefore, we have the identity and the state of the correct process. The liar process can be killed and a new copy of the correct process can be started.

Observe that in the above algorithm we never decode the data structure at the fused-SM. During normal operations, we only do the encoding. Whenever there is Byzantine fault detected, we use $F(1)$ only to determine which of the copies is correct. We can encode $O(1)$ crucial information to determine whether $P(i)$ or

$Q(i)$ is a liar. Also observe that if the fault occurs in the fused machine, it does not affect the overall operation of the system and it is not even detected. If early detection of fault in the fused machine is important for some application, then periodically (or during off-peak period) one could simply reset and recompute the fused process data. Thus, decoding of the fused-SM is not required.

3.2 Tolerating f Byzantine Faults in State Machines with $O(1)$ State

To generalize the above algorithm for f faults, we maintain the invariant that there is at least one correct copy in spite of f faults. Therefore, we keep f copies of each of the primary server and f fused copies. Thus, we have total of $n * f + f$ state machines in addition to n primary machines. The only requirement on the fused copies $\{H(j), j = 1..f\}$is that if $H(j)$ is not faulty and if we have $n - 1$ correct values of the primary machines, then the remaining one can be determined using $H(j)$. Thus, a simple xor or sum based fused-SM is sufficient. Even though we are tolerating f faults, the requirement on the fused copy is only for a single fault (because we are also using replication).

The primary copy together with its f replicas are called *unfused* copies. If any of $f + 1$ unfused copies differ, we call the primary server *mismatched*. Let the value of one of the copies be v. The number of unfused copies with value v is called the *multiplicity* of that copy.

We now generalize Theorem 2 for $f \geq 1$ faults. At first, we will assume that the state space of each of the state machines is small. Later, we generalize it to the case when each of the state machine has $O(m)$ state.

Theorem 3. *There exists an algorithm with $fn + f$ backup state machines that can tolerate f Byzantine faults and has the same overhead as the RSM approach during normal operation and additional $O(nf)$ overhead during recovery.*

Proof. We keep f copies for each primary state machine and f overall fused machines. This results in additional $nf + f$ state machines in the system. If there are no faults among unfused copies, all $f + 1$ copies will result in the same output and therefore the system will incur same overhead as the RSM approach.

Our algorithm first checks the number of primary state machines that are mismatched. First consider the case when there is a mismatch between a primary state machine $P(i)$ and its replica for at most one value of $i = 1..n$. Let that primary machine be $P(c)$. Since there are at most f faults, we can conclude that we have the correct state of all other primary state machines $P(i), i \neq c$. Now given the correct state of all other primary machines, we can successively retrieve the state of $P(c)$ from fused machines $H(j), j = 1..f$ till we find one of the unfused machine that has $f + 1$ multiplicity. We have to decode at most f fused machines each at cost of $O(n)$.

Now consider the case when there is a mismatch for at least two primary state machines, say $P(c)$ and $P(d)$. Let $\alpha(c)$ and $\alpha(d)$ be the largest multiplicity among unfused copies of $P(c)$ and $P(d)$ respectively. Without loss of generality, assume that $\alpha(c) \geq \alpha(d)$. We show that the copy with multiplicity $\alpha(c)$ is correct.

Unfused Copies:
On receiving any message from client
Update local copy;
send state update to fused processes;
send response to the client;

Client:
send event to all unfused $f+1$ copies;
if (all $f+1$ responses identical)
    use the response;
else invoke recovery algorithm;

Fused Copies:
On receiving update from unfused copy
if (all $f+1$ updates identical)
    carry out the update
else invoke recovery algorithm;

Recovery Algorithm:
Let t be the number of mismatched SMs;
while $t > 1$ do
    choose the copy with largest multiplicity;
    kill all incorrect unfused copies;
    restart them with the chosen copy;
    $t = t - 1$;

// Can assume that t equals one.
// Let $P(c)$ be the mismatched SM
for $(j = 1; j \leq f; j++)$
    create a new copy using $H(j)$ and $P(i), i \neq c$;
    if (any copy has multiplicity $f+1$)
        recover to that copy and return;

Fig. 5. Algorithm C: Tolerating f Byzantine faults

If this copy is not correct, then there are at least $\alpha(c)$ liars among unfused copies of $P(c)$. We now claim that there are at least $f+1-\alpha(d)$ liars among unfused copies of $P(d)$ which gives us the total number of liars as $\alpha(c)+f+1-\alpha(d) \geq f+1$ contradicting the assumption on the maximum number of liars. Consider the copy among unfused copies of $P(d)$ with multiplicity $\alpha(d)$. If this copy is correct we have $f+1-\alpha(d)$ liars. If this value is false, we know that the correct value has multiplicity less than or equal to $\alpha(d)$ and therefore there are at least $f+1-\alpha(d)$ liars among unfused copies of $P(d)$.

By identifying the correct value, we have reduced the number of mismatched primary state machines by 1. By repeating this argument, we get to the case when there is exactly one mismatched primary machine.

Based on the proof of Theorem 3, we get the Algorithm C shown in Figure 5, to tolerate f Byzantine faults with nf replicated and f fused-SMs.

In Algorithm C, we had to decode the fused-SMs during the recovery algorithm. The algorithm requires at most f fusion processes to be decoded in the worst case. If there are $t \leq f$ faults, we are guaranteed that after decoding t fused-SMs we will have $f+1+t$ unfused copies. At least one of these copies will have multiplicity of $f+1$ or more. Alternatively, we can try out all the values of unfused copies of $P(c)$ and $\{P(i), i \neq c\}$ to compute H and thereby determine multiplicity of various copies.

3.3 Tolerating f Byzantine Faults for State Machines with $O(m)$ State

We now extend the algorithm to the case when each of the primary state machine has $O(m)$ state. We would like to avoid decoding or encoding the entire fused

```
Z:set of copies initially {1..f + 1};
while (all unfused copies in Z not identical)
    w = min{r : ∃p, q ∈ Z : state_p[r] ≠ state_q[r]};
    j = 1;
    while (no copy with multiplicity f + 1)
        create state[w] using H(j) and P(i), i ≠ c;
        j = j + 1;
    endwhile;
    delete other copies from Z;
endwhile;
return any copy from Z;
```

Fig. 6. Locating a Correct Unfused Copy for mismatched $P(c)$: *locate(int c)*

process. As observed earlier, one of the $f+1$ unfused copies is guaranteed to be correct and it is sufficient to locate this copy using fused copies. We give an algorithm with $O(mf+nt^2)$ time complexity to locate the correct copy. Assume that we are trying to locate the correct copy among unfused copies of $P(c)$.

In the algorithm shown in Fig. 6, the set Z maintains the invariant that it includes all the correct unfused copies (and may include incorrect copies as well). The invariant is initially true because all indices from $1..f+1$ are in Z. Since the set has $f+1$ indices and there are at most f faults, we know that the set Z always contains at least one correct copy.

The outer *while* loop iterates until all copies are identical. If all copies in Z are identical, from the invariant it follows that all of them must be correct and we can simply return any of the copies in Z. Otherwise, there exist at least two different copies in Z, say p and q. Let w be the first index in which states of copies p and q differ [1]. Either copy p or the copy q (or both) are liars. We now use the fused machines to recreate copies of $state[w]$. Since we have the correct copies of all other primary machines $P(i), i \neq c$, we can use them with the fused copies $H(j), j = 1..f$. Note that the fused copies may themselves be wrong so it is necessary to get enough multiplicity for any value to determine if some copy is faulty. Suppose that for some v, we get multiplicity of $f+1$. This implies that any copy with $state[w] \neq v$ must be faulty and therefore can safely be deleted from Z. We are guaranteed to get a value with multiplicity $f+1$ out of total $2f+1$ copies. Further, since copies p and q differ in $state[w]$, we are guaranteed to delete at least one of them in each iteration of while. Eventually, the set Z would either be singleton or will contain only identical copies. In either case, the *while* loop terminates and we have located a correct copy.

We now analyze the time complexity of the procedure *locate*. Assume that there are $t \leq f$ actual faults that occurred. We delete at least one index in each iteration of the outer *while* loop and there are at most t faulty processes giving us the bound of t for the number of iterations of the while loop. In each

[1] For simplicity, we view the state of machines as an $O(m)$ array (though in practice it could be any structure with size $O(m)$).

iteration, creating $state[w]$ requires at most $O(1)$ state to be decoded for each fusion process at the cost of $O(n)$. The maximum number of fused processes that would be required is t. Thus, $O(nt)$ work is required for a single iteration before a copy is deleted from Z. To determine w in incremental fashion requires $O(mf)$ work cumulative over all iterations. Combining these costs we get the complexity of the algorithm to be $O(mf + nt^2)$.

By using the method *locate*, in the recovery algorithm we get the following result – the main result of the paper.

Theorem 4. *Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $nf + f$ state machines that can tolerate f Byzantine faults and has the same overhead as the RSM approach during the normal operation and additional $O(mf + nt^2)$ overhead during recovery where t is the actual number of faults that occurred in the system.*

Theorem 4 combines advantages of replication and coding theory. We have enough replication to guarantee that there is at least one correct copy at all times and therefore we do not need to decode the entire state machine but only locate the correct copy. We have also taken advantage of coding theory to reduce the number of copies from $2f$ to f.

It can be seen that our algorithm is optimal in the number of unfused and fused copies it maintains to guarantee that there is at least one correct unfused copy and that faults of any f machines can be tolerated. The first requirement dictates that there be at least $f+1$ unfused copies and the recovery from Byzantine fault requires that there be at least $2f + 1$ fused or unfused copies in all.

4 Conclusions

We have presented efficient distributed algorithms to tolerate crash and Byzantine faults of state machines in distributed systems. Our algorithms use a combination of replication and coding theory to achieve efficiency in detection and correction of faults. Our algorithms use fewer backup state machines (and therefore smaller space, and fewer messages in many cases) while providing the same level of fault-tolerance.

Acknowledgements. I am thankful to Bharath Balasubramanian, Vinit Ogale and Yogish Sabharwal for discussions on the topic.

References

1. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. ACM Commun. 21(7), 558–565 (1978)
2. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Program. Lang. Syst. 6(2), 254–280 (1984)
3. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)

4. Sivasubramanian, S., Szymaniak, M., Pierre, G., van Steen, M.: Replication for web hosting systems. ACM Comput. Surv. 36(3), 291–334 (2004)
5. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Publishing Company, Amsterdam (1981)
6. van Lint, J.H.: Introduction to Coding Theory. Springer, Heidelberg (1998)
7. Pease, M., Shostak, R., Lamport, L.: Reaching agreements in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)
8. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Information Processing Letters 39(6), 343–350 (1991)
9. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM 24 (1981)
10. Garg, V.K., Ogale, V.A.: Fusible data structures for fault-tolerance. In: ICDCS, p. 20. IEEE Computer Society, Los Alamitos (2007)
11. Balasubramanian, B., Garg, V.K.: A fusion-based approach for handling multiple faults in data structures. Technical Report ECE-PDS-2009-001, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin (2009)
12. Ogale, V.A., Balasubramanian, B., Garg, V.K.: A fusion-based approach for tolerating faults in finite state machines. In: IPDPS, pp. 1–11. IEEE, Los Alamitos (2009)
13. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). In: SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, pp. 109–116. ACM Press, New York (1988)
14. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: Raid: high-performance, reliable secondary storage. ACM Comput. Surv. 26(2), 145–185 (1994)
15. Plank, J.S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software – Practice & Experience 27(9), 995–1012 (1997)
16. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A., Spielman, D.A., Stemann, V.: Practical loss-resilient codes. In: STOC '97: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 150–159. ACM Press, New York (1997)
17. Byers, J.W., Luby, M., Mitzenmacher, M., Rege, A.: A digital fountain approach to reliable distribution of bulk data. SIGCOMM Comput. Commun. Rev. 28(4), 56–67 (1998)
18. Garg, V.K.: Implementing fault-tolerant services using fused state machines. Technical Report ECE-PDS-2010-001, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin (2010)
19. Plank, J.S., 0002, Y.D.: Note: Correction to the 1997 tutorial on reed-solomon coding. Softw., Pract. Exper. 35(2), 189–194 (2005)
20. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems 5(1), 47–76 (1987)

Low Communication Self-stabilization through Randomization

Shay Kutten and Dmitry Zinenko*

Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract. Most self-stabilizing protocols rely on checking every neighbor of a node continuously to detect failures. Such protocols have a high communication cost, especially in dense graphs. Conceivably, one can check neighbors less often, reducing the amount of communication per round. However, delaying the checking delays the fault detection and the stabilization, and therefore has the potential of increasing the total amount of communication overhead until stabilization.

In this paper, we strive to reduce "after stabilization" overhead, without increasing the "before stabilization" overhead. For that, we investigate the potential effect of randomization on the communication efficiency of self-stabilizing protocols. We present randomized low communication self-stabilizing algorithms for several major tasks, namely, spanning tree construction, distributed reset, and unison. We study this approach in a complete graph, since there the communication overhead of checking seems the highest when one strives for protocols that are also fast.

1 Introduction

In his seminal paper in 1973, Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [16]. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." A strong motivation was demonstrated by [30], who showed that crash failures can drive protocols to arbitrary states.

In many early self stabilizing protocols every node collects complete global information (e.g. [24]). Even in later algorithms, using the local checking and detection paradigm [2, 8], a node collects the state of all its neighbors. Masuzawa et al. [14, 26] showed that a complete knowledge of the neighborhood is not always necessary, especially after stabilization. In this work, we show that this is sometimes the case even before stabilization.

We study this approach in a complete graph, since there the communication overhead of checking seems the highest when one strives for protocols that are also fast. The results may be meaningful also for a virtual complete graph, e.g. the application layer of the Internet, that is an overlay over a less dense graph.

* Supported in part by a grant from the ISF.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 465–479, 2010.

We present randomized low communication self-stabilizing algorithms for several major tasks, namely, the tasks of a spanning tree construction, distributed reset, and unison. This implies results for many other important tasks, such as naming, routing, etc. Our results compare favorably to previous papers studying our problems in the communication efficiency measures developed by [14], as well in the traditional measure of the number of bits sent.

When applied in a complete graph, all the known self-stabilizing spanning tree algorithms require $\Omega(n^2)$ messages until stabilization. Similarly, previous solutions for distributed reset and unison in dense networks are time efficient, but have a high communication overhead. By comparison, our algorithms stabilize after only $O(n \log n)$ messages and $O(\log n)$ rounds with high probability[1] until stabilization, and $O(n)$ messages per round later. Our reset and unison protocols can also be used (although with a different round complexity) in more general classes of synchronous networks. For example, in bounded degree networks their round complexity is $O(D+\log n)$ with high probability (D is the graph diameter), while still sending one message per node per round.

Besides reducing communication, the "smooth" and uniform behavior of our protocols should be especially desirable for the implementation of middleware, since it avoids hard-to-predict fluctuations in the network load. A middleware protocol that uses unbalanced and hard to predict patterns of communications can potentially cause network congestions and negatively affect not only its own performance, but also that of other applications on the same network.

1.1 Model

A distributed system is a set of cooperating computing elements (processors), interconnected by a network. It is represented by a graph, with processors represented by vertices (also called nodes), and the links between processors represented by graph edges. Our algorithms do not use any assumption about unique identifiers, that is, our network is *anonymous*. In this work, we concentrate on the case of the complete graph topology, where a communication link exists between every pair of nodes. However, the order of outgoing communication links as seen by each processor is arbitrary. It is impossible to determine which link leads to which processor until a message is received from that link directly.

An execution of a *synchronous* distributed system proceeds in *rounds*. Each round, the processors make their pending state transitions and may send and/or receive messages over their communication links. We use a standard model (e.g., [1]), where all the processors are active each round, and all the messages sent during a particular round are received before the beginning of the next round.

At any point of time transient adversarial failures may alter arbitrarily the state of any process variables. However, the adversary is oblivious of, and unable to affect, the random bits generated but the processes. As is common for self-stabilization protocols, when dealing with the algorithms' time complexity, we

[1] An event occurs with high probability if it occurs with probability $1 - O(n^{-\alpha})$ for any fixed $\alpha \geq 1$. All our algorithms converge with probability 1, the w.h.p. part relates to the number of rounds (and therefore messages) until the convergence.

always count the number of rounds *after the last failure has occurred*. Therefore, our model of failures is equivalent to the case when the algorithm is initialized to arbitrary values, possibly chosen by an adversary, but no further failures occur when the execution time is measured.

1.2 Related Work

Intuitively, one can identify two sources for the high communication complexity. One is the desire to be fast, and hence to communicate with many neighbors at the same time. The other arises from the self stabilization context: even assuming we already stabilized, a node must verify that this is indeed the case. For example, in a leader election protocol, every two nodes need to be compared from time to time, to make sure they do not both consider themselves to be the leaders. Electing a leader in a complete network can be achieved in one round. However, for a deterministic algorithm, this would cost $\Omega(n^2)$ messages. Similarly, the time efficient unison algorithms of [7, 10] would stabilize fast when applied to a complete graph, but would have used $\Omega(n^2)$ messages per round.

In [13, 14], the authors ask whether it is possible to lower the communication complexity of self-stabilizing protocols below the need of checking every neighbor forever. In many cases, their answer is negative for the deterministic algorithms. However, randomized algorithms, such as the ones we present here, could potentially overcome this limitation. Another difference between our papers is that they concentrate mainly on reducing communication after the stabilization is complete. We, on the other hand, are interested in developing protocols that are message-efficient at any time, and not only after the stabilization.

For general graphs, multiple message-driven algorithms were presented for the spanning tree construction in the context of self stabilization [11, 13, 20, 22]. Some protocols for self-stabilizing distributed reset appear in [2, 4, 5], and for the stabilizing unison problem in [6, 7, 10, 21]. All those algorithms may use $\Omega(n^2)$ communication until convergence when applied in a dense graph.

The spanning tree protocol presented in [22] has a similar motivation, and also uses a bounded degree spanning tree to reduce communication and memory requirements. The idea of contacting only one neighbor at a time was used for reducing memory requirements in various papers, such as in [9]. For deterministic algorithms, this approach often increases the stabilization time. The technique of repeated tree coloring to detect competing trees was presented in [2, 4], and several other publications. We use it with a slight adaptation in Sect. 5.1-5.2. Afek and Matias have already used randomization in [3] to solve an equivalent problem (leader election) with $O(n)$ messages (w.h.p.). It was adapted to the self stabilizing context in [4] to achieve improved time complexity. We use the identifier selection mechanism of [3] in our spanning tree algorithm.

This work is also related to randomized gossip algorithms, such as those described in [25, 31], and random branching processes [15]. The branching process used by our spanning tree algorithm bears a similarity to a simpler random gossip process analyzed by Pittel [28]. The exact random process we describe, to our best knowledge, has never been presented before.

2 The Random Process

In this section, we describe a simple random process that serves as a starting point for our construction. It grows a spanning tree with degree (maximum branching factor) $d \geq 2$ in a complete graph with n nodes, *given a root node*.

The tree starts with exactly one node, the predefined root of the tree. Every node in the tree can have at most d children. Every *request* round, every node v that belongs to the tree and has $d(v) < d$ children, selects $d - d(v)$ additional nodes as its children independently at random[2]. After the request round is complete, all the nodes that were selected this way join the tree during the *confirmation* round. A newly joining node chooses one of the tree nodes that selected it arbitrarily and notifies its new parent.

Let T_i be the number of nodes in the tree after i request and i confirmation rounds have completed, starting with $T_0 = 1$.

The number of leaves in a d-regular tree with T_i internal nodes is always exactly $T_i(d-1)+1$. Therefore, this is the number of *selections* $\sum_v (d - d(v))$ made during request round i, regardless of the precise tree structure. The number of selections $T_i(d-1)+1 > T_i(d-1)$ is almost linear in the number of tree nodes. Since the number of joining nodes is non-decreasing in the number of selections, we may bound the number of tree nodes from below by a random gossip model, where every tree member sends a message uniformly at random to $d-1$ nodes each round. It is well known (see, e.g. [23]) that even for $d = 2$, messages from the root will reach all the nodes with high probability after only $O(\log n)$ rounds.

To generalize, let $\mathcal{A}$ be a synchronous algorithm on n nodes, with a state that can be defined in terms of the number of nodes in a particular structure. In our case this structure will always be a tree or a forest. Define $T_i^{\mathcal{A}}$ to be the number of nodes in that structure (by analogy with T_i).

Lemma 1. *If* $\mathbb{E}(T_{i+1}^{\mathcal{A}} | T_i^{\mathcal{A}} = x) \geq \mathbb{E}(T_{i+1} | T_i = x)$ *for all* $x > 0$, *and* $T_{i+1}^{\mathcal{A}} \in [T_i^{\mathcal{A}}, (d+1)T_i^{\mathcal{A}}]$ *for constant* d, *then for some* $k = O(\log n)$, $T_k^{\mathcal{A}} = n$ *w.h.p.*.

The proof of the last lemmas is rather lengthy and mathematical, and is based on several standard probability concentration bounds. Due to the lack of space, its presentation is deferred to the full paper.

3 Spanning Forest Algorithm

First, we show a simple self-stabilizing extension of the protocol in Sect. 2. This is not the final algorithm, we present it mainly to prove some basic properties that are used later on.

Like many other spanning tree algorithms that are based on local checking and detection (e.g., [2, 18]), for maintaining a tree, we provide every node with two variables: `treeid` and `treedistance`. All the error-free nodes in a

[2] We assume that a node selects new children independently, including selecting itself, to simplify the analysis. In real-world implementations, it should not select the same node more than once, or select nodes which are already in its child/parent set.

single tree share the same `treeid`, and all, except for the roots, have a positive `treedistance` which is larger by one than that of their parent. Nodes with `treedistance` equal to 0 consider themselves roots. The algorithm ensures that all the nodes eventually join the trees with the largest `treeid` in the graph. The pseudocode run by every node is the same and appears on Fig. 1–3.

Figure 1. Spanning forest algorithm, main loop

```
1: loop
2:     R ← received request messages
3:     C ← received confirmation messages
4:     if request then // this is the request round
5:         run request procedure (Fig. 2)
6:     else
7:         run confirmation procedure (Fig. 3)
8:     request ← ¬request
```

Figure 2. Spanning forest algorithm, request procedure

```
1: children ← C
2: while |children| < d do
3:     children ← children ∪ random neighbor
4: send request ⟨treeid, treedepth⟩ to the first d children
```

Figure 3. Spanning forest algorithm, confirmation procedure

```
1: candidates ← {r ∈ R : r > ⟨treeid, treedepth⟩}
2: if |candidates| > 0 then
3:     parent ← max(candidates) // an arbitrary maximal candidate if not unique
4:     treeid ← parent.treeid
5:     treedepth ← parent.treedepth + 1
6:     send confirmation ⟨⟩ to parent
7: else // we are orphaned, become a root
8:     treedepth ← 0
```

When one node captures another node that already has children, we can use this fact to capture the subtree of the new node as fast as possible. A newly joined node does not start with an empty `children` set, like in the process in Sect. 2. Its "effort" from the earlier rounds is preserved to some degree — it is more likely to receive confirmation from its old children on the following round. We were unable to account for this optimization in the analysis, but our simulations show that it has a measurable impact on the convergence speed.

To be able to capture subtrees quickly, we would also like our trees to be of a small height. A node prefers, among all potential parents with the same `treeid`, the ones with the smallest `treedistance`. The trees "contract", because new nodes join as close to the root as possible, and also move closer to the

root within the same tree whenever offered such a possibility. For the simplicity of description, we define an order on `<treeid, treedistance>` pairs to be as above: first increasing by `treeid`, and then decreasing by `treedistance`. Any comparisons in the algorithm definitions use this order.

Note that the algorithm of Sect. 2 alternated between request and confirmation rounds. In the context of self-stabilization, a round may be a request round at one node and a confirmation round at another. We say that two nodes are *synchronized* if their request rounds coincide. Handling the lack of synchronization directly in the spanning forest algorithm would make it and the related proofs much less elegant. We prefer to ignore this problem for now, by not claiming anything about the algorithm's properties when the configuration is not synchronized; it will be solved later using a separate unison algorithm (Sect. 4.1).

3.1 Algorithm Analysis

For this section we define the *proper parent* of a non-root node v to be the node to which v sent the confirmation message during its last confirmation round, but only if it has the same `treeid` as v. Otherwise v has no current proper parent. A node's children are the nodes in its `children` set. The following lemma is given without a proof due to the lack of space.

Lemma 2. *At every point of time after the first 3 rounds, a node u is the proper parent of a node v only if v is a child of u. The graph formed by the (directed) proper parent relationships is a directed forest with the branching factor $\leq d$.*

Lemma 3. *Assume that there is only one node with the maximum `treeid` in the graph, and all the nodes are synchronized. Then, when running the spanning forest algorithm, that node's tree absorbs all the nodes in the graph within $O(\log n)$ rounds with high probability.*

Proof. We base our proof on Lemma 1, comparing our algorithm to the process in Sect. 2. The choices to which new nodes to send request messages are made independently and uniformly at random. Therefore both processes can be analyzed using the standard balls-in-bins model (see, for example, [27]). Such a model involves two independent factors: the number of balls thrown (messages sent), and the probability per ball of hitting a bin of a particular type (a node which is not already in the tree).

In order to show that $\mathbb{E}(T_{i+1}^{\mathcal{A}} | T_i^{\mathcal{A}} = x) \geq \mathbb{E}(T_{i+1} | T_i = x)$, it is enough to prove that when the tree contains T_i nodes: (1) at least $T_i(d-1)$ selections are made; and (2) every such selection is independent and hits a node in the tree with probability at most $\frac{T_i}{n}$. The first item is correct for the same reason as in Sect. 2: this is the number of leaves in a d-regular tree with T_i internal nodes.

The second item would be correct if all the selections would be new random selections (line 3 in the request procedure), simply because the probability of randomly hitting a tree node is $\frac{T_i}{n}$. However, as we have mentioned earlier, new nodes that already had children before joining the tree may preserve them. This did not happen in the original process. First, note that those children cannot

be in the tree when they send the confirmations, otherwise they would not send confirmations for their parent's previous, inferior, `treeid`.

We assume that all the nodes are synchronized, therefore the nodes receive any request messages together with the request message from their current parent. Assume that one such child node receives k such messages. Only one message, the one from its current parent, was sent to it deterministically. The other $k-1$ messages must be new random selections, made uniformly and independently from each other.

Now consider the event that k new random selections messages arrive at some node. For k independent uniformly distributed random variables $U_1, \ldots, U_k$, it holds that $\Pr(U_2 = U_1, \ldots, U_k = U_1) = \Pr(U_2 = x, \ldots, U_k = x)$ for any x s.t. $\Pr(U_1 = x) \neq 0$. Fixing one of those k random selections, the probability of this event is the same as the probability of the event described in the previous paragraph. The difference is, in the former, the node was already in the tree with probability 0, and in the latter with a probability of $\frac{T_i}{n}$. We conclude, therefore, that in the former case the expected number of captured nodes is at least as large as if all the selections were new independent random selections.

Finally, $T_{i+1}^{\mathcal{A}} \in [T_i^{\mathcal{A}}, (d+1)T_i^{\mathcal{A}}]$ for the same reasons as in Sect. 2.

We define the *leader forest* to be the forest of nodes with the largest `treeid` value in the graph (it is a forest because of Lemma 2)[3]. We do not claim anything about the speed of convergence when the nodes are not synchronized, since our final spanning tree algorithm will make sure this is the case eventually.

Lemma 4. *Assume that all the nodes are synchronized. When running the spanning forest algorithm, the leader forest absorbs all the nodes in the graph within* $O(\log n)$ *rounds with high probability.*

Proof. It is not hard to see, that eventually all the nodes in the graph must be captured by one of the trees in the leader forest. We compare its growth with the growth of a single dominating tree from Lemma 3.

Let q be the number of different roots in the leader forest, and $T_{i,j}^{\mathcal{A}}$ be the number of nodes in the tree of root j after i confirmation rounds. While the single tree in Lemma 3 performs exactly $T_i(d-1)+1$ selections during the following request round, the leader forest with the same total number of nodes tries at least $\sum_j \left(T_{i,j}^{\mathcal{A}}(d-1)+1\right) = T_i^{\mathcal{A}}(d-1)+q$. The probability of hitting a node outside the leader forest is the same in both cases, as it depends only on the local tree structure and the total number of nodes.

As the leader forest makes at least as many selections with at least the same probability of success as in Lemma 3, we can apply Lemma 1 to it as well.

As the reader can see, the algorithm of this section is not very useful, unless we can ensure that there is only one root with the highest `treeid` and synchronize all the nodes to the same round. We show how to do it in the following sections.

[3] There may be several trees with the largest `treeid`, if node identities are not unique.

4 Low-Bandwidth Self-stabilizing Reset and Unison

In a regular random gossip [23], every node with a "knowledge" sends it to a single random neighbor every round. In a complete graph, the knowledge becomes known to all the nodes with high probability after $\mathrm{COV} = O(\log n)$ rounds.

We add one additional variable, `reset`, to the node state. Any node can start a reset procedure by setting its `reset` to 2COV. The only exception are the nodes that already have a non-zero `reset` variable; those are the nodes that already participate in a reset. The `reset` value is used as a countdown "time-to-live" timer: every round when the timer is still positive, the node decreases it by 1 and sends its value to a random neighbor. When the value goes from 1 to 0, the node resets. The intuition behind using 2COV is that it takes COV rounds for all the nodes to become aware that a reset is in progress and another COV rounds to agree on when to perform the reset.

When a node receives a `reset` timer value from another node, it takes the maximum between all the received values and its own as the new `reset` value. However, the nodes never set their timer to values larger than 2COV, since those were clearly introduced by an adversary.

Theorem 1. *Let i be a round when one of the nodes initializes a reset procedure. Then, with high probability, during some round t, such that $i + 2COV \leq t < i + 3COV$, all the nodes in the graph reset simultaneously (and on round $t + 1$ all the* `reset` *variables are 0).*

Proof. Let v be a node that initialized a reset on round i, and let j be the current round. As v counts down its timer, there could be new resets initialized by nodes other than v; there can also be older resets initialized before round i.

Consider the way messages from v spread among the nodes in the graph. Whenever a message reaches a node u that does not participate in a reset, or participates in a reset that started before round i, that node has a smaller `reset`. In this case, u adopts the new timer value. Whenever a message reaches a node u with the same, or a higher `reset` value (i.e. u participates in a reset that was initialized during round i or later), u ignores it. In both cases, u starts or continues to spread messages with $reset \geq 2\mathrm{COV} - (j - i)$.

Because every value spreads to all the nodes w.h.p. within COV rounds, during round $i{+}\mathrm{COV}$, all the nodes have $reset \geq \mathrm{COV}$ with high probability. Therefore, no new resets are initialized during rounds $i{+}\mathrm{COV} \ldots i{+}2\mathrm{COV}$, and the *maximal* value of `reset` remains consistent during those rounds (the maximal value of $reset + j$ is constant).

Applying the same theorem again, tells us that, with high probability, all the nodes adopt this maximum within the remaining COV rounds. Since no resets could be initialized after round $i{+}\mathrm{COV}$ (w.h.p.), this maximum must be between $2\mathrm{COV} - (j - i)$ and $3\mathrm{COV} - (j - i)$.

Corollary 1. *A distributed reset can be performed in a complete or bounded degree graph in a self-stabilizing fashion with high probability within $O(\log n)$ rounds while using $O(n \log n)$ messages.*

Our reset procedure works *in any graph* when substituting the appropriate COV value. For example, in bounded degree networks COV $= O(D + \log n)$, where D is the graph diameter [19].

4.1 The Unison Problem

Assume that we have n processors connected in a complete synchronous network. Every processor is running a clock counting the number of rounds that passed since some event in the past. The *unison* problem [21] is to ensure that all the clocks are synchronized, i.e. their values are equal after every increment.

Theorem 2. *The unison problem can be solved in a complete or bounded degree graph in a self-stabilizing fashion with high probability within* $O(COV + \log n)$ *rounds while using* $O(n(COV + \log n))$ *messages.*

Proof. In order to solve the unison problem, the reset algorithm is used as follows. Every processor sends one message to a random neighbor every round. The message contains both its clock value and its `reset` value. A processor that receives a message with an unsynchronized clock value initializes a reset. When a processor resets, it sets its clock to a predefined value (e.g., 0).

When not all the clocks are synchronized, at least one node has a minority clock value. At least half of the other nodes will initialize a reset if they receive a message from this node, so the probability of reset in this case is at least $1/2$. In the case of the bounded degree graph, this probability is at least $1/\Delta = O(1)$, because there must be two neighboring nodes whose clocks differ. Therefore, a reset will be initiated after $O(\log n)$ rounds with high probability. When a reset is initialized, all the nodes are guaranteed to execute it simultaneously within $O(\mathrm{COV})$ rounds with high probability, and set their clocks to a common value. Since every node sends one message each round, the message complexity of this procedure is $O(n(\mathrm{COV} + \log n))$ w.h.p..

5 Spanning Tree Algorithm

5.1 Tree Recoloring

We use the technique of repeatedly recoloring trees (see, e.g., [2, 17, 4, 18]) to allow the roots to detect a situation when there are several roots with the same `treeid` in the graph. Repeated tree recoloring works as follows: from time to time, the root of the tree chooses a new random color to use and sends it to its children. Propagation with feedback (see, e.g. [29]) is then used to make sure the whole tree is colored in this color. When the root has received completion reports from all its direct children, it chooses a new random color. Lemma 5 ensures that a new color is chosen at least once every $O(\log n)$ rounds:

Lemma 5. *Let* $CAP = O(\log n)$ *be the time required for the leader forest to capture all the nodes in the graph w.h.p. (as in Lemma 4). Then, with high probability, at any time after the first* $i \geq CAP$ *request rounds, the height of every tree in the graph (or equivalently* $\max(treedepth)$*) is at most* CAP*.*

Proof. Define the *restricted leader forest* $\mathcal{F}$ to be the subgraph of the leader forest that contains only the nodes with `treedistance` less than the last request round number. During the first request and confirmation rounds, those are the nodes that have the maximum `treeid` and are roots. During the second — those that have the maximum `treeid` and are either roots or their immediate children, etc.

We would like to prove that $\mathcal{F}$ captures every node in the graph within CAP rounds. In order to do so, consider the leader forest $\mathcal{F}'$ that would result from increasing the `treeid` of all the original leader forest roots by 1. Clearly, Lemma 4 applies to $\mathcal{F}'$.

$\mathcal{F}$ and $\mathcal{F}'$ contain the same nodes during the first round. A node in $\mathcal{F}'$ is always able to capture any node not in $\mathcal{F}'$. Additionally, a node is in $\mathcal{F}'$ after round pair i if and only if it either was in $\mathcal{F}'$ before, or received a request message from one of the nodes in $\mathcal{F}'$ during that round pair. Observe that the same holds for $\mathcal{F}$. We can therefore create a coupling between the random choices of the nodes in $\mathcal{F}$ and $\mathcal{F}'$, so that the selections are made with the same probabilities and the same results. Therefore, on any particular round, the distribution of $|\mathcal{F}|$ is the same as the distribution of $|\mathcal{F}'|$, and Lemma 4 applies to $\mathcal{F}$ as well.

We conclude that after CAP rounds, all the nodes in the graph are in $\mathcal{F}$ with high probability. After CAP request rounds, the height of any tree in $\mathcal{F}$ is at most CAP by the definition of $\mathcal{F}$, and after a node is captured by one of the leader trees, its height can only decrease.

Because of Lemma 5 we only allow the `treedistance` of any node to be less than or equal to CAP. This way we avoid very deep trees set up by the adversary. Any node that discovers that its `treedistance` is greater than this value resets it to 0 and becomes a root. Since in the proof neither $\mathcal{F}$ nor $\mathcal{F}'$ had any nodes with treedistance $>$ CAP, this modification does not affect algorithm correctness (more specifically the ability of the leader forest to capture every node in the graph within CAP rounds w.h.p.).

The next corollary follows from the fact that the time required to recolor a tree is at most twice its depth:

Corollary 2. *Completely recoloring any tree in the leader forest using propagation with feedback requires at most* $O(\log n)$ *rounds.*

5.2 Competitor Detection and Synchronization

Let CAP $= O(\log n)$ be the time required for a single root with the highest `treeid` to capture all the nodes, as in Lemma 4. After CAP rounds have passed, with high probability there are multiple roots with the highest `treeid` if and only if there are multiple trees.

In order to detect the presence of multiple trees, we use tree recoloring combined with a round counter. In our algorithm, only the roots perform the detection based on the request messages they receive. Whenever a root receives a request message from a node colored differently than its last two colors, that

originating node must belong to a different tree. If a root receives such a message after enough rounds have passed, it starts a global reset procedure, which we have specified in Sect. 4.

The purpose of the reset is for all the nodes to choose new random `treeid`s simultaneously. We use the ID selection procedure from [3] to ensure that after every such reset, an unique leader exists with high probability regardless of n:

Lemma 6 (Lemma 6 from [3]). *If n nodes run the procedure Choose(treeid), there is a unique node with the largest `treeid`, with probability at least $1-\epsilon$, where $\epsilon = 1/r$.*

For our purpose we take $r = n^\alpha$. The procedure consists of drawing two random integers as the `treeid`: the primary key is drawn from a Geometric distribution with $p = 1/2$, and the secondary key is uniformly selected from $[1, 36r\log(4r)]$. The largest primary key is less than $(\alpha+1)\log n$ with high probability, however the secondary key may be as large as $36\alpha n^\alpha \log(4n)$, and therefore the size of a node `treeid` is $O(\log n)$ bits.

Lemma 7. *If there are $R > 1$ trees in the leader forest, then the probability that no root receives a request message from a competing tree over $CAP + \log n$ request rounds is less than $2n^{-1}$.*

Proof. By Lemma 4, the probability that the the leader forest has not absorbed all the nodes after CAP request rounds is less than n^{-1}.

If all the trees belong to the leader forest, as long as no detection occurs, R cannot change. At least $n(d-1)+R$ request messages are sent every request round (recall the proof of Lemma 4), and each one of them arrives at one of the roots of a different tree with a probability $(R-1)/n$. The probability that no such event occurs during the remaining $\log n$ request rounds is:

$$\left(1-\frac{R-1}{n}\right)^{(n(d-1)+R)\log n} < e^{-(d-1)(R-1)\log n} \leq n^{-(d-1)(R-1)} \leq n^{-1} \qquad (1)$$

The probability that any of those two unfavorable events occur is therefore at most $2n^{-1}$.

Let the competition detection mechanism use $\Omega(n)$ colors. The probability that a message arriving at a root from a different tree has either the root's current or previous color is $O(1/n)$. Divide the execution into epochs of $\mathrm{CAP} + \log n = O(\log n)$ request rounds. The following corollary follows from Corollary 2 combined with Lemma 7.

Corollary 3. *If there is more than one root in the leader forest, and the detection mechanism is using $\Omega(n)$ colors, then a reset is triggered within $O(\log n)$ request rounds with high probability.*

The detection of lack of synchronization between node request rounds is even easier. It is enough that a node receives a request message after its confirmation round or vice versa. Alternatively, we could reduce the phase synchronization to unison and use Theorem 2.

Lemma 8. *If not all the nodes are synchronized, this is detected by some node w.h.p. within $O(1)$ rounds.*

Proof. Let there be cn nodes that consider the current round to be a request round. Assume that $c \leq 1/2$, otherwise wait for the next round.

The cn nodes send at least $cn(d-1)$ messages to random nodes during their request round. For each of these messages, the probability of triggering detection is $1-c$. Therefore, the probability that none of the $cn(d-1)$ messages trigger detection is $c^{cn(d-1)}$. By taking a derivative in c, there is one minimum at $c = 1/e$. Since $c \in [1/n, 1/2]$, $c^{cn(d-1)} \leq \max\left(n^{-(d-1)}, 2^{-\frac{n}{2}(d-1)}\right) = O(n^{-1})$.

5.3 Algorithm Description and Analysis

The pseudocode for the final spanning tree algorithm appears on Fig. 4-7. The local state variable `roundcounter` counts the number of rounds since the beginning of the algorithm or since the last reset.

Figure 4. Spanning tree algorithm, main loop

```
1: loop
2:     R ← received request messages
3:     C ← received confirmation messages
4:     T ← received reset messages
5:     run reset handler procedure (Fig. 5)
6:     if request then
7:         if (R ≠ ∅) ∧ (reset = 0) then // not synchronized
8:             reset ← 2COV
9:         run request procedure (Fig. 6)
10:    else
11:        if (C ≠ ∅) ∧ (reset = 0) then // not synchronized
12:            reset ← 2COV
13:        run confirmation procedure (Fig. 7)
14:    request ← ¬request
```

Note that in order to achieve bounded memory/message size, we must limit the values of the drawn `treeid`s. This also makes our algorithms more practical to implement. The probability that *any* node draws a value larger than $(\alpha+2)\log n$ in $O(\log n)$ attempts (resets) is just $o(n^{-\alpha})$. Therefore, with high probability, the execution of the algorithm with truncated entries is identical to its execution if the `treeid`s weren't limited. When the execution is different, it leads to $tO(\log n)$ rounds increase in time complexity with a probability of $tn^{-\alpha-1}$, therefore the expectation of this increase is $O(1)$.

All the resets introduced by an adversary must cease within $O(\log n)$ rounds. It does not matter that some nodes may reset earlier than others, or that the reset breaks some of the invariants we relied on earlier. After the first reset initialized by the algorithm (if such is needed), all the nodes are synchronized. After every such reset, with high probability there exists only one leader tree

Figure 5. Spanning tree algorithm, reset handler procedure

```
1: if 𝒯 ≠ ∅ then
2:     reset ← max(reset, max(𝒯))
3: if reset > 0 then
4:     reset ← min(reset, 2COV) − 1
5:     if reset = 0 then
6:         ℛ, 𝒞 ← ∅
7:         roundcounter ← 0
8:         treeid ← ⟨min(Geo(1/2), (α + 2) log n), 1 + ⌊U(36αn^α log(4n))⌋⟩
9:         request ←true
10:    else
11:        send reset to a random neighbor
```

Figure 6. Spanning tree algorithm, request procedure

```
1: roundcounter ← min(roundcounter + 1, CAP)
2: coloringdone ← ∄m ∈ 𝒞 : ((m.color ≠ color) ∨ (m.coloringdone = false))
3: children ← {confirm.source : confirm ∈ 𝒞}
4: while |children| < d do
5:     children ← children ∪ random neighbor
6: send request ⟨treeid, treedepth, color⟩ to the first d children
```

Figure 7. Spanning tree algorithm, confirmation procedure

```
1: candidates ← {r ∈ ℛ : (r > ⟨treeid, treedepth⟩) ∧ (r.treedepth < CAP)}
2: if |candidates| > 0 then
3:     parent ← max(candidates)
4:     treeid ← parent.treeid
5:     treedepth ← parent.treedepth + 1
6:     if color ≠ parent.color then
7:         color ← parent.color
8:         coloringdone ← false
9:     send confirmation ⟨color, coloringdone⟩ to parent.source
10: else // we are orphaned, become a root
11:    treedepth ← 0
12:    if (roundcounter ≥ CAP) ∧ (∃r ∈ ℛ : (r.color ∉ {color, oldcolor})) ∧ (reset =
       0) then
13:        reset ← 2COV
14:    if coloringdone then
15:        oldcolor ← color
16:        color ←random color
```

(Lemma 6). Therefore, there are only $O(1)$ resets w.h.p.. By Theorem 1, every reset takes at most $O(\log n)$ rounds. The time between resets is also $O(\log n)$ because of Corollary 3 and Lemma 8. Finally, after the last reset, there is only one node with the highest `treeid`, so by Lemma 3, a spanning tree is constructed within $O(\log n)$ rounds. Every round, at most nd messages are sent. The size of all the variables is $O(\log n)$ bits. As a result, we have the following theorem:

Theorem 3. *On average and with high probability the spanning tree algorithm converges within $O(\log n)$ rounds and uses $O(n \log n)$ messages. The algorithm uses $O(\log n)$ memory bits at every node, and this is also the size of its messages.*

6 Conclusion and Further Work

An interesting question is whether there is a gap in the communication efficiency between randomized protocols and deterministic ones. [1] showed a lower bound of $\Omega(n \log n)$ messages for any deterministic (even non self-stabilizing) leader election. However, in their model not all the processors may be initially active.

Our reset protocol works in any graph. It is possible that our other algorithms can be configured or improved to work in a more general topology as well (expander graphs look particularly promising). Some of this work seems extensible to asynchronous networks, or at least asynchronous bounded networks [12]. We have also done some work on a generalization of the random process that can be applied, with some additional work, to allow handling crash failures.

Another direction for improvement is a tighter mathematical analysis. Intuitively, re-capturing an existing sub-tree of a new node should be easier than hitting new nodes randomly. After running some simulations, we learned that is has a real effect on the algorithm's time complexity, but were unable to find a method that would allow us to account for its effects in the theoretical analysis.

References

[1] Afek, Y., Gafni, E.: Time and message bounds for election in synchronous and asynchronous complete networks. In: PODC'85, pp. 186–195. ACM, New York (1985)
[2] Afek, Y., Kutten, S., Yung, M.: Memory-efficient self-stabilization on general networks. In: Toueg, S., Kirousis, L.M., Spirakis, P.G. (eds.) WDAG 1991. LNCS, vol. 579, pp. 15–28. Springer, Heidelberg (1992)
[3] Afek, Y., Matias, Y.: Elections in anonymous networks. Information and Computation Journal 113(2), 312–330 (1994)
[4] Aggarwal, S., Kutten, S.: Time optimal self-stabilizing spanning tree algorithm. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 400–410. Springer, Heidelberg (1993)
[5] Arora, A., Gouda, M.: Distributed reset. IEEE Transactions on Computing 43(9), 1026–1038 (1994)
[6] Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. In: Toueg, S., Kirousis, L.M., Spirakis, P.G. (eds.) WDAG 1991. LNCS, vol. 579, pp. 71–79. Springer, Heidelberg (1992)
[7] Awerbuch, B., Kutten, S., Mansour, Y., Patt-Shamir, B., Varghese, G.: Time optimal self-stabilizing synchronization. In: STOC'93, pp. 652–661 (1993)
[8] Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset. In: Tel, G., Vitányi, P.M.B. (eds.) WDAG 1994. LNCS, vol. 857, pp. 326–339. Springer, Heidelberg (1994)
[9] Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: A new self-stabilizing minimum spanning tree construction with loop-free property. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 407–422. Springer, Heidelberg (2009)
[10] Boulinier, C., Petit, F., Villain, V.: When graph theory helps self-stabilization. In: PODC'04, pp. 150–159 (2004)

[11] Burman, J., Kutten, S.: Time optimal asynchronous self-stabilizing spanning tree. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 92–107. Springer, Heidelberg (2007)
[12] Chou, C.-T., Cidon, I., Gopal, I.S., Zaks, S.: Synchronizing asynchronous bounded delay networks. In: van Leeuwen, J. (ed.) WDAG 1987. LNCS, vol. 312, pp. 212–218. Springer, Heidelberg (1988)
[13] Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Masuzawa, T., Tixeuil, S. (eds.) SSS 2007. LNCS, vol. 4838, pp. 219–233. Springer, Heidelberg (2007)
[14] Devismes, S., Masuzawa, T., Tixeuil, S.: Communication efficiency in self-stabilizing silent protocols. In: ICDCS'09, pp. 474–481. IEEE, Los Alamitos (2009)
[15] Devroye, L.: Branching processes and their applications in the analysis of tree structures and tree algorithms. In: Probabilistic Methods for Algorithmic Discrete Mathematics, vol. 16, pp. 249–314. Springer, Heidelberg (1998)
[16] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
[17] Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. In: PODC'90, pp. 103–117 (1990)
[18] Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems 8(4), 424–440 (1997)
[19] Feige, U., Peleg, D., Raghavan, P., Upfal, E.: Randomized broadcast in networks. In: Asano, T., Imai, H., Ibaraki, T., Nishizeki, T. (eds.) SIGAL 1990. LNCS, vol. 450, pp. 128–137. Springer, Heidelberg (1990)
[20] Gärtner, F.C.: A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Ecole Polytechnique Fèdèrale de Lausanne (2003)
[21] Gouda, M.G., Herman, T.: Stabilizing unison. Information Processing Letters 35(4), 171–175 (1990)
[22] Hérault, T., Lemarinier, P., Peres, O., Pilard, L., Beauquier, J.: A model for large scale self-stabilization. In: IPDPS'07, pp. 1–10 (2007)
[23] Karp, R., Schindelhauer, C., Shenker, S., Vocking, B.: Randomized rumor spreading. In: FOCS'00, p. 565. IEEE Computer Society, Los Alamitos (2000)
[24] Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distributed Computing 7(1), 17–26 (1993)
[25] Koldehofe, B.: Simple gossiping with balls and bins. Studia Informatica Universalis 3(1), 43–60 (2004)
[26] Masuzawa, T., Izumi, T., Katayama, Y., Wada, K.: Brief announcement: Communication-efficient self-stabilizing protocols for spanning-tree construction. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 219–224. Springer, Heidelberg (2009)
[27] Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
[28] Pittel, B.: On spreading a rumor. SIAM Journal of Applied Mathematics 47(1), 213–223 (1987)
[29] Segall, A.: Distributed network protocols. IEEE Transactions on Information Theory IT-29, 23–35 (1983)
[30] Varghese, G., Jayaram, M.: The fault span of crash failures. Journal of the ACM 47, 47–52 (1997)
[31] Vogels, W., van Renesse, R., Birman, K.: The power of epidemics: robust communication for large-scale distributed systems. SIGCOMM Comput. Commun. Rev. 33(1), 131–135 (2003)

Fast Self-stabilizing Minimum Spanning Tree Construction

Using Compact Nearest Common Ancestor Labeling Scheme

Lélia Blin[1,3], Shlomi Dolev[4],
Maria Gradinariu Potop-Butucaru[2,3,5], and Stephane Rovedakis[1,6]

[1] Université d'Evry-Val d'Essonne, 91000 Evry, France
[2] Université Pierre & Marie Curie - Paris 6, 75005 Paris, France
[3] LIP6-CNRS UMR 7606, France
{lelia.blin,maria.gradinariu}@lip6.fr
[4] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il
[5] INRIA REGAL, France
[6] Laboratoire IBISC-EA 4526, 91000 Evry, France
stephane.rovedakis@ibisc.fr

Abstract. We present a novel self-stabilizing algorithm for minimum spanning tree (MST) construction. The space complexity of our solution is $O(\log^2 n)$ bits and it converges in $O(n^2)$ rounds. Thus, this algorithm improves the convergence time of all previously known self-stabilizing asynchronous MST algorithms by a multiplicative factor $\Theta(n)$, to the price of increasing the best known space complexity by a factor $O(\log n)$. The main ingredient used in our algorithm is the design, for the first time in self-stabilizing settings, of a labeling scheme for computing the nearest common ancestor with only $O(\log^2 n)$ bits.

1 Introduction

Since its introduction in a centralized context [15,14], the minimum spanning tree (or MST) problem gained a benchmark status in distributed computing thanks to the seminal work of Gallager, Humblet and Spira [9].

The emergence of large scale and dynamic systems, often subject to transient faults, revives the study of scalable and self-stabilizing algorithms. A *scalable* algorithm does not rely on any global parameter of the system (*e.g.* upper bound on the number of nodes or the diameter). *Self-stabilization* introduced first by Dijkstra in [6] and later publicized by several books [7,8] deals with the ability of a system to recover from catastrophic situation (i.e., the global state may be arbitrarily far from a legal state) without external (*e.g.* human) intervention in finite time.

Although there already exists self-stabilizing solutions for the MST construction, none of them considered the extension of the Gallager, Humblet and Spira

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 480–494, 2010.

algorithm (GHS) to self-stabilizing settings. Interestingly, this algorithm unifies the best properties for designing large scale MSTs: it is fast and totally decentralized and it does not rely on any global parameter of the system. Our work proposes an extension of this algorithm to self-stabilizing settings. Our extension uses only logarithmic memory and preserves all the good characteristics of the original solution in terms of convergence time and scalability.

Gupta and Srimani presented in [13] the first self-stabilizing algorithm for the MST problem. The MST construction is based on the computation of all shortest paths (for a certain cost function) between all pairs of nodes. While executing the algorithm, every node stores the cost of all paths from it to all the other nodes. To implement this algorithm, the authors assume that every node knows the number n of nodes in the network, and that the identifiers of the nodes are in $\{1, \ldots, n\}$. Every node u stores the weight of the edge $e_{u,v}$ placed in the MST for each node $v \neq u$. Therefore the algorithm requires $\Omega(\sum_{v \neq u} \log w(e_{u,v}))$ bits of memory at node u. Since all the weights are distinct integers, the memory requirement at each node is $\Omega(n \log n)$ bits. The main drawback of this solution is its lack of scalability since each node has to know and maintain information for all the nodes in the system. Note also that the time complexity announced by the authors, $O(n)$ stays only in the particular synchronous settings considered by the authors. In asynchronous setting the complexity is $\Omega(n^2)$ rounds. A different approach for the message-passing model, was proposed by Higham and Liang [11]. The algorithm performs roughly as follows: every edge checks whether it eventually belongs to the MST or not. To this end, every non tree-edge e floods the network to find a potential cycle, and when e receives its own message back along a cycle, it uses the information collected by this message (*i.e.*, the maximum edge weight of the traversed cycle) to decide whether e could potentially be in the MST or not. If the edge e has not received its message back after the time-out interval, it decides to become tree edge. The memory used by each node is $O(\log n)$ bits, but the information exchanged between neighboring nodes is of size $O(n \log n)$ bits, thus only slightly improving that of [13]. This solution also assume that each node has access to a global parameter of the system: the diameter. Its computation is expensive in large scale systems and becomes even harder in dynamic settings. The time complexity of this approach is $O(mD)$ rounds where m and D are the number of edges and the diameter of the network respectively, i.e., $O(n^3)$ rounds in the worst case.

In [2] we proposed a self-stabilizing loop-free algorithm for the MST problem. Contrary to previous self-stabilizing MST protocols, this algorithm does not make any assumption on the network size (including upper bounds) or the unicity of the edge weights. The proposed solution improves on the memory space usage since each participant needs only $O(\log n)$ bits while preserving the same time complexity as the algorithm in [11].

Clearly, in the self-stabilizing implementation of the MST algorithms there is a trade-off between the memory complexity and their time complexity (see Table 1, where a boldface denotes the most useful (or efficient) feature for a particular criterium). The challenge we address in this paper is to design fast and scal-

Table 1. Distributed Self-Stabilizing algorithms for the MST problem

	a priori knowledge	space complexity	convergence time
[13]	network size and the nodes in the network	$O(n \log n)$	$\Omega(n^2)$
[11]	upper bound on diameter	$O(\log n)$ +messages of size $O(n \log n)$	$O(n^3)$
[2]	**none**	$\mathbf{O(\log n)}$	$O(n^3)$
This paper	**none**	$O(\log^2 n)$	$\mathbf{O(n^2)}$

able self-stabilizing MST with little memory. Our approach brings together two worlds: the time efficient MST constructions and the memory compact informative labeling schemes. Therefore, we extend the GHS algorithm to self-stabilizing settings and keep compact its memory space by using a self-stabilizing extension of the nearest common ancestor labeling scheme of [1]. Note that labeling schemes have already been used in order to infer a broad set of information such as vertex adjacency, distance, tree ancestry or tree routing [5], however none of these schemes have been studied in self-stabilizing settings (except the last one).

Our contribution is therefore twofold. We propose for the first time in self-stabilizing settings a $O(log^2 n)$ bits scheme for computing the nearest common ancestor. Furthermore, based on this scheme, we describe a new self-stabilizing algorithm for the MST problem. Our algorithm does not make any assumption on the network size (including upper bounds) or the existence of an a priori known root. Moreover, our solution is the best space/time compromise over the existing self-stabilizing MST solutions. The convergence time is $O(n^2)$ asynchronous rounds and the memory space per node is $O(\log^2 n)$ bits. Interestingly, our work is the first to prove the effectiveness of an informative labeling scheme in self-stabilizing settings and therefore opens a wide research path in this direction.

2 Model and Notations

We consider an undirected weighted connected network $G = (V, E, w)$ where V is the set of nodes, E is the set of edges and $w : E \to \mathbb{R}^+$ is a positive cost function. Nodes represent processors and edges represent bidirectional communication links. Additionally, we consider that $G = (V, E, w)$ is a network in which the weight of the communication links may change value.

The processors asynchronously execute their programs consisting of a set of variables and a finite set of rules. The variables are part of the shared register which is used to communicate with the neighbors. A processor can read and write its own registers and can read the shared registers of its neighbors. Each processor executes a program consisting of a sequence of guarded rules. Each *rule* contains a *guard* (Boolean expression over the variables of a node and its neighborhood) and an *action* (update of the node variables only). Any rule whose

guard is *true* is said to be *enabled*. A node with one or more enabled rules is said to be *privileged* and may make a *move* executing the action corresponding to the chosen enabled rule.

A *local state* of a node is the value of the local variables of the node and the state of its program counter. A *configuration* of the system $G = (V, E)$ is the cross product of the local states of all nodes in the system. The transition from a configuration to the next one is produced by the execution of an action at a node. A *computation* of the system is defined as a *weakly fair, maximal* sequence of configurations, $e = (c_0, c_1, \ldots c_i, \ldots)$, where each configuration c_{i+1} follows from c_i by the execution of a single action of at least one node. During an execution step, one or more processors execute an action and a processor may take at most one action. *Weak fairness* of the sequence means that if any action in G is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of G is enabled in the final global state.

In the sequel we consider the system can start in any configuration. That is, the local state of a node can be corrupted. Note that we don't make any assumption on the bound of corrupted nodes. In the worst case all the nodes in the system may start in a corrupted configuration. In order to tackle these faults we use self-stabilization techniques.

Definition 1 (self-stabilization). *Let $\mathcal{L}_{\mathcal{A}}$ be a non-empty* legitimacy predicate[1] *of an algorithm $\mathcal{A}$ with respect to a specification predicate Spec such that every configuration satisfying $\mathcal{L}_{\mathcal{A}}$ satisfies Spec. Algorithm $\mathcal{A}$ is* self-stabilizing *with respect to Spec iff the following two conditions hold:*
*(i) Every computation of $\mathcal{A}$ starting from a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ preserves $\mathcal{L}_{\mathcal{A}}$ (*closure*).*
*(ii) Every computation of $\mathcal{A}$ starting from an arbitrary configuration contains a configuration that satisfies $\mathcal{L}_{\mathcal{A}}$ (*convergence*).*

3 Overview of our Solution

We propose to extend the Gallager, Humblet and Spira (GHS) algorithm, [9], to self-stabilizing settings via a compact informative labeling scheme. Thus, the resulting solution presents several advantages appealing for large scale systems: it is compact since it uses only logarithmic memory in the size of the network, it scales well since it does not rely on any global parameter of the system, it is fast — its time complexity is the better known in self-stabilizing settings. Additionally, it self-recovers from any transient fault.

The central notion in the GHS approach is the notion of *fragment*. A fragment is a partial spanning tree of the graph, i.e., a fragment is a tree which spans a subset of nodes. Note that a fragment can be limited to a single node. An outgoing edge of a fragment F is an edge with a unique endpoint in F.

[1] A legitimacy predicate is defined over the configurations of a system and is an indicator of its correct behavior.

The minimum-weight outgoing edge of a fragment F is denoted in the following as $\texttt{ME}_F$. In the GHS construction, initially each node is a fragment. For each fragment F, the GHS algorithm in [9] identifies the $\texttt{ME}_F$ and merges the two fragments endpoints of $\texttt{ME}_F$. Note that, with this scheme, more than two fragments may be merged concurrently. The merging process is recursively repeated until a single fragment remains. The result is a MST. The above approach is often called "blue rule" for MST construction.

This approach is particularly appealing when transient faults yield to a forest of fragments (which are sub-trees of a MST). The direct application of the blue rule allows the system to reconstruct a MST and to recover from faults which have divided the existing MST. However, when more severe faults hit the system the process variables may be corrupted leading to a configuration of the network where the set of fragments are not sub-trees of some MST. That is, it may be a spanning tree but not of minimum weight, or it can contain cycles. In this case, the application of the blue rule only is not sufficient to reconstruct a MST. To overcome this difficulty, we combine the blue rule with another method, referred in the literature as the "red rule". The red rule removes the heaviest edge from every cycle. The resulting configuration contains a MST. We use the red rule as follows: given a spanning tree T of G, every edge e of G that is not in T is added to T, thus creating a (unique) cycle in $T \cup \{e\}$. This cycle is called a *fundamental cycle*, denoted by C_e. If e is not the edge of maximum weight in C_e, then, according to the red rule, there exists an edge $f \neq e$ in C_e with $w(f) > w(e)$. The edge of maximum weight can be removed since it is not part of any MST.

Our `MST` construction combines both the blue and red rules. The blue rule application needs that each node identifies its own fragment. The red rule requires that nodes identify the fundamental cycle corresponding to every adjacent non-tree-edge. In both cases, we use a self-stabilizing labeling scheme, called `NCA-L`, which provides at each node a distinct informative label such that the nearest common ancestor of two nodes can be identified based only on the labels of these nodes (see Section 3.1). Thus, the advantage of this labeling is twofold. First the labeling helps nodes to identify their fragments. Second, given any non-tree edge $e = (u, v)$, the path in the tree going from u to the nearest common ancestor of u and v, then from there to v, and finally back to u by traversing e, constitute the fundamental cycle C_e.

To summarize, our algorithm will use the blue rule to construct a spanning tree, and the red rule to recover from invalid configurations. In both cases, it uses our algorithm `NCA-L` to identify both fragments and fundamental cycles. Note that, in [3,4] distributed algorithms using the blue and red rules to construct a MST in a dynamic network are proposed, however these algorithms are not self-stabilizing.

Variables used by `NCA-L` *and* `MST` *modules* For any node $v \in V(G)$, we denote by $N(v)$ the set of all neighbors of v in G. We use the following notations:

- p_v: *the parent* of v in the current spanning tree, an integer pointer to a neighbor;

- ℓ_v: *the label* of v composed of a list of pairs of integers where each pair is an identifier and a distance (the size of ℓ_v is bounded by $O(\log^2 n)$ bits);
- $size_v$: a pair of variables, the first one is an integer *the number of nodes in the sub-tree* rooted at v and the second one is the identifier of the child u of v with the maximum number of nodes in the sub-tree rooted at u;
- mwe_v: the *minimum weighted edge* composed by a pair of variables, the first one is an integer, the weight of the edge and the second one is the label of a node u stored in ℓ_u.

3.1 Self-stabilizing Nearest Common Ancestor Labeling

Our labeling scheme, called in the following `NCA-L`, uses the notions of *heavy* and *light* edges introduced in [10]. In a tree, a *heavy* edge is an edge between a node u and one of its children v of maximum number of nodes in its sub-tree. The other edges between u and its other children are tagged as *light* edges. We extend this edge designation to the nodes, a node v is called *heavy node* if the edge between v and its parent is a heavy edge, otherwise v is called *light node*. Moreover, the root of a tree is a heavy node. The idea of the scheme is as follows. A tree is recursively divided into disjoint paths: the heavy and the light paths which contain only heavy and light edges respectively.

- $\text{Child} \equiv \{u \in N(v) : p_u = \mathsf{Id}_v\}$
- $\text{SizeC}(v) \equiv \text{Leaf}(v) \vee (size_v = (1 + \sum_{u \in \text{Child}(v)} size_u, \arg\max\{size_u : u \in \text{Child}(v)\}))$
- $\text{Leaf}(v) \equiv (\nexists u \in N(v), p_u = \mathsf{Id}_v) \wedge size_v = (1, \perp)$
- $\text{Label}(v) \equiv \text{Label}_R(v) \vee \text{Label}_{Nd}(v)$
- $\text{Label}_R(v) \equiv (p_v = \emptyset \wedge \ell_v = (\mathsf{Id}_v, 0))$
- $\text{Label}_{Nd}(v) \equiv p_v \in N(v) \wedge (\text{Heavy}(v) \vee \text{Light}(v))$
- $\text{Heavy}(v) \equiv size{p_v}[1] = \mathsf{Id}_v \wedge size_v[0] < size{p_v}[0] \wedge \mathsf{last}(\ell p_v)[1] + 1 = \mathsf{last}(\ell_v)[1]$
- $\text{Light}(v) \equiv size{p_v}[1] \neq \mathsf{Id}_v \wedge size_v[0] \leq size{p_v}[0]/2 \wedge \ell_v = \ell p_v.(\mathsf{Id}_v, 0)$
- $nca(u,v) \equiv \begin{cases} \ell.(a_0,a_1) \text{ s.t. } \ell_u \cap \ell_v = \ell, \ell_u = \ell.(a_0,a_1).\ell'_u \text{ and } \ell_v = \ell.(b_0,b_1).\ell'_v & \textbf{if } (a_0 = b_0 \vee \ell \neq \emptyset) \wedge \ell_u \prec \ell_v \\ \ell.(b_0,b_1) \text{ s.t. } \ell_u \cap \ell_v = \ell, \ell_u = \ell.(a_0,a_1).\ell'_u \text{ and } \ell_v = \ell.(b_0,b_1).\ell'_v & \textbf{if } (a_0 = b_0 \vee \ell \neq \emptyset) \wedge \ell_v \prec \ell_u \\ \emptyset & \textbf{otherwise} \end{cases}$
- $\text{Cycle}(v) \equiv \ell_v \subset \ell p_v \vee \ell_v \prec \ell p_v$
- $\text{MinEnabled}(v) \equiv \text{Enabled}(v) \wedge (\forall u \in N(v), \text{Enabled}(u) \wedge \mathsf{Id}_v < \mathsf{Id}_u)$

Fig. 1. Predicates used by the algorithm `NCA-L` for the labeling procedure

To label the nodes in a tree T, the size of each subtree rooted at each node of T is needed to identify heavy edges leading the heaviest subtrees at each level of T. To this end, each node v maintains a variable named $size_v$ which is a pair of integers. The first integer is the local estimation of the number of nodes in the subtree rooted at v. The second integer is the identifier of a child

of v with maximum number of nodes. That is, it indicates the heavy edge. The computation of $size_v$ is processed from the leaves to the root (see Predicate SizeC(v) in Figure 1). A leaf has no child, therefore $size_v = (1, \perp)$ for a leaf node v (see Predicate Leaf(v) in Figure 1 and Rule R_ℓ).

Based on the heavy and light nodes in a tree T indicated by variable $size_v$ at each node $v \in T$, each node of T can compute its label. The label of a node v stored in ℓ_v is a list of pair of integers. Each pair of the list contains the identifier of a node and a distance to the root of the heavy path (i.e., a path including only heavy edges). For the root v of a fragment, the label ℓ_v is the following pair $(\mathsf{Id}_v, 0)$, respectively the identifier of v and the distance to itself, i.e., zero (see Rule $R_\odot$). When a node u is tagged by its parent as a heavy node (i.e., $size_{p_v}[1] = \mathsf{Id}_u$), then the node u takes the label of its parent but it increases by one the distance of the last pair of the parent label. Examples of theses cases are given in Figure 2, where integers inside the nodes are node identifiers and lists of pairs of values are node labels. When a node u is tagged by its parent v as a light node (i.e., $size_{p_v}[1] \neq \mathsf{Id}_u$), then the node u becomes the root of a heavy path and it takes the label of its parent to which it adds a pair of integers composed of its identifier and a zero distance (see Figure 2).

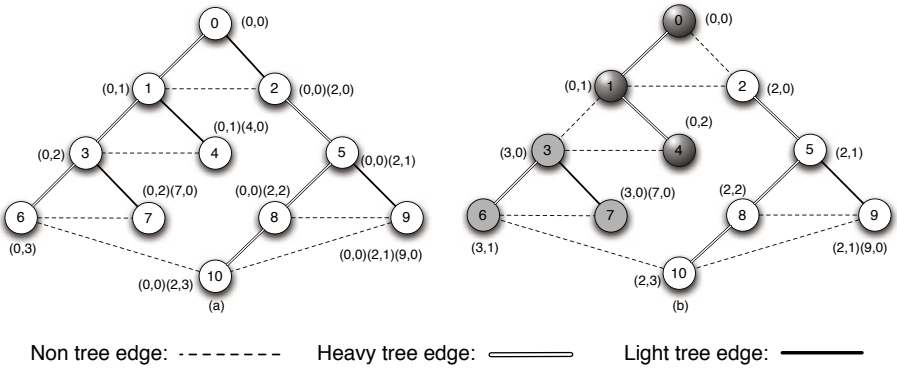


Fig. 2. Labeling scheme

This labeling scheme is used in second part of this article in `MST` algorithm to find the minimum weighted edges, but it is also used to detect and destroy cycles since the initial configuration may not be a spanning tree. To this end, we define an order $\prec$ on the labels of nodes. Let a and b be two nodes and ℓ_a and ℓ_b be their respective labels such that $\ell_a = \ell.(a_0, a_1).\ell'_a$ and $\ell_b = \ell.(b_0, b_1).\ell'_b$ with $\ell_a \cap \ell_b = \ell$. The label of a node a is lower than the label of node b, noted $\ell_a \prec \ell_b$, if (1) $(a_0, a_1).\ell'_a = \emptyset$ and $(b_0, b_1).\ell'_b \neq \emptyset$, or (2) $a_0 < b_0$ or (3) $a_0 = b_0$ and $a_1 < b_1$.

A node u can detect the presence of a cycle by only comparing its label with the label of its parent. That is, if its label is contained in the label of its parent, or it is inferior to the one of its parent then u is part of a cycle (see Predicate

Cycle(v) in Figure 1). In this case, the node u becomes the root of its fragment in order to break the cycle (see below Rule $R_{\odot}$).

Algorithm NCA-L is composed of two rules. Rule $R_{\odot}$ creates a root or breaks cycles while rule R_{ℓ} produces a proper labeling. Note that the last predicates in rules $R_{\odot}$ and R_{ℓ} (the part in gray) are used only for insuring the exclusivity of rules execution when the labeling scheme works together with the MST scheme.

A node v with an incoherent parent (which is not one of its neighbors) or present in a cycle executes $R_{\odot}$. Following the execution of this rule node v becomes a root node, it sets its parent to void and its label to $(\mathsf{Id}_v, 0)$.

Rule R_{ℓ} helps a node v to compute the number of nodes in its sub-tree (stored in variable $size_v$) and provides to v a coherent label.

$R_{\odot}$**:** [**Root creation**]
If $p_v \notin N(v) \vee (p_v = \emptyset \wedge \ell_v \neq (\mathsf{Id}_v, 0) \vee (\text{Cycle}(v) \wedge \neg\text{NeedReorientation}(v))$
Then $p_v := \emptyset$; $\ell_v = (\mathsf{Id}_v, 0)$;

R_{ℓ}**:** [**Label correction**]
If $\neg\text{Cycle}(v) \wedge (\neg\text{SizeC}(v) \vee \neg\text{Label}(v)) \wedge \text{MinEnabled}(v) \wedge \neg\text{TreeMerg}(v)$
Then If Leaf(v) **then** $size_v = (1, \perp)$
Else $size_v := (1 + \sum_{u \in \text{Child}(v)} size_u, \arg\max\{size_u : u \in \text{Child}(v)\})$;
If $size_{p_v}[1] = \mathsf{Id}_v$ **then** $\ell_v := \ell_{p_v}$; $last(\ell_v)[1] := last(\ell_v)[1] + 1$;
Else $\ell_v = \ell_{p_v}.(\mathsf{Id}_v, 0)$

3.2 Self-stabilizing MST

In this section we describe our self-stabilizing MST algorithm. The algorithm executes two phases: the MST correction and the MST fragments merging. Recall that our algorithm uses the blue rule to construct a spanning tree and the red rule to recover from invalid configurations. In both cases, it uses the nearest-common ancestor labeling scheme to identify fragments and fundamental cycles. We assume in the following that the *merging* operations have a higher priority than the *recovering* operations. That is, the system recovers from an invalid configuration if and only if no merging operation is possible. In the worst case, after a failure hit the system, a merging phase will be followed by a recovering phase and finally by a final merging phase.

The minimum weighted edge and MST correction. Note that the scope of our labeling scheme is twofold. First, it allows a node to identify the neighbors that share the same fragment and consequently to select the outgoing edges of a fragment. Second, the labeling scheme may be used to identify cycles and to repair the tree. To this end, the algorithm uses the nearest common ancestor predicate *nca* depicted in Figure 1. For two nodes u and v with $e = (u, v)$ a non tree edge (i.e., $p_u \neq v$ and $p_v \neq u$), if the nearest common ancestor does not exist then u and v are in two distinct fragments (i.e., if we have $nca(u, v) = \emptyset$). Otherwise u and v are in the same fragment F and the addition of e to

- $\text{MergeChild}(v) \equiv \min\{mwe_u : u \in \text{Child}(v) \wedge mwe_u[1] = \emptyset\}$
- $\text{MergeAdj}(v) \equiv (\min\{w(u,v) : u \in N(v) \setminus \text{Child}(v) \setminus \{p_v\} \wedge nca(u,v) = \emptyset\}, \emptyset)$
- $\text{MergeEdge}(v) \equiv \min\{\text{MergeChild}(v), \text{MergeAdj}(v)\}$
- $\text{FarLcaChild}(v) \equiv \arg\min_{\prec}\{mwe_u[1] : u \in \text{Child}(v) \wedge mwe_u[1] \preceq \ell_v\}$
- $\text{RecoverChild}(v) \equiv mwe_u$ such that $\text{FarLcaChild}(v) = u$
- $\text{FarLca}(v) \equiv \arg\min_{\prec}\{nca(u,v) : u \in N(v) \setminus \text{Child}(v) \setminus \{p_v\} \wedge mwe_v[1] \neq \emptyset \wedge nca(u,v) \succ mwe_v[1] \wedge nca(u,v) \neq \emptyset\}$
- $\text{RecoverAdj}(v) \equiv \begin{cases} (w(u,v), nca(u,v)) \text{ s.t. } \text{FarLca}(v) = u & \textbf{if } \text{FarLca}(v) \neq \emptyset \\ (w(u,v), nca(u,v)) & \textbf{otherwise} \\ \text{s.t. } u = \arg\min_{\prec}\{nca(u,v) : \\ u \in N(v) \setminus Child(v) \setminus \{p_v\} \wedge nca(u,v) \neq \emptyset\} \end{cases}$
- $\text{RecoverEdge}(v) \equiv \begin{cases} \text{RecoverChild}(v) \textbf{ if } \text{RecoverAdj}(v)[1] \prec \text{RecoverChild}(v) \\ \text{RecoverAdj}(v) \quad \textbf{otherwise} \end{cases}$
- $\text{MinEdge}(v) \equiv \begin{cases} \text{MergeEdge}(v) \quad \textbf{if } \text{MergeEdge}(v) \neq \emptyset \\ \text{RecoverEdge}(v) \textbf{ otherwise} \end{cases}$
- $\text{NeedReorientation}(v) \equiv \text{Label}_R(v) \vee (\ell_{p_v} = (\perp, \perp) \wedge p_{p_v} = \mathsf{Id}_v)$
- $\text{EndReorientation}(v) \equiv \ell_{p_v} = (\emptyset, \emptyset) \wedge (\ell_v = (\perp, \perp) \vee \ell_v \neq (\emptyset, \emptyset))$
- $\text{TreeMerg}(v) \equiv \text{NeedReorientation}(v) \vee \text{EndReorientation}(v)$
- $\text{NewFrag}(v) \equiv mwe_v = mwe_{p_v} \wedge mwe_v[1] \neq \mathsf{Id}_v \wedge w(v, p_v) > mwe_v[0]$

Fig. 3. Predicates used by the MST for the tree correction or the fusion fragments

F generates a cycle. Let $\texttt{path}(x, y)$ be the set of edges on the unique path between x and y in F, with $x, y \in F$. The fundamental cycle C_e is the following: $C_e = \texttt{path}(u, nca(u,v)) \cup \texttt{path}(nca(u,v), v) \cup e$. Consider the example depicted on Figure 2(b). The labels of nodes 10 and 6 are respectively $\ell_{10} = (2, 3)$ and $\ell_6 = (3, 1)$. In this case $nca(10, 6) = \emptyset$ so the edge $(10, 6)$ is an outgoing edge because the nodes 10 and 6 are in two distinct fragments and they have no common ancestor. If the edge $(10, 6)$ is of minimum weight then $(10, 6)$ can be used for a merging between the fragment rooted in 2 and the fragment rooted in 3. For the case of nodes 10 and 9 the labels are $\ell_{10} = (2, 3)$ and $\ell_9 = (2, 1)(9, 0)$ and $nca(10, 9) = (2, 1)$. Consequently, 10 and 9 are in the same fragment. The fundamental cycle C_e with $e = (9, 10)$ goes through the node with the label $nca(10, 9)$, in other word the node 5 in Figure 2(b).

Predicate MinEdge(v) (see Figure 3) computes both the minimum weight outgoing edge used in a merging phase and the internal edges used in a recovering phase. Our algorithm gives priority to the computation of minimum outgoing edges via Predicate MinEdge(v). A recovering phase is initiated if there exists a unique tree or if a sub-tree of one fragment has no outgoing edge.

The computation of the minimum weight outgoing edge is done in a fragment F_u if and only an adjacent fragment F_v is detected by F_u, i.e., if we have Predicate MergeEdge(v) $\neq \emptyset$. In this case, using Rule R_{Min} each node collects from the leaves to the root the outgoing edges leading to an adjacent fragment F_v. At each level in a fragment, a node selects the outgoing edge of minimum

weight among the outgoing edges selected by its children and its adjacent outgoing edges. Thus, this allows to the root of a fragment to select the minimum outgoing edge e of the fragment leading to an adjacent fragment. Then, the edge e can be used to perform a merging between two adjacent fragments using an edge belonging to a MST.

Let us explain Rule $R_{\lightning}$ which allows to correct a tree (or a fragment). In this case, the information about the non-tree edges are sent to the root as follows. Among all its non-tree edges, a node u sends the edge $e = (u, v)$ with the $nca(u, v)$ nearest to the root (see Figure 4(a)). The information about the edge e is stored in variable mwe_u. If the parent x of the node u has the same information and the weight of the edge $w(u, x) > w(e)$ then the edge (u, v) is removed from the tree (see Figure 4(a-b) for the nodes 6 and 10). We use the red rule in an intensive way, because we remove all the edges with a weight upper than $w(e)$ in fundamental cycle of e. This interpretation of the red rule allows to insure that after a recovering phase the remaining edges belong to a MST.

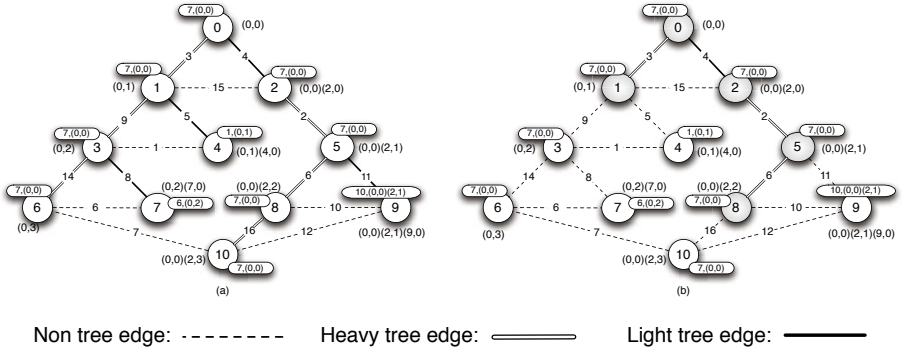


Fig. 4. Minimum weighted edge computation and Tree correction. The bubble at each node v corresponds to the weight and the label of the common ancestor of the edge stored on variable mwe_v.

R_{Min}: [**Minimum computation**]
If $\neg\text{Cycle}(v) \wedge \text{Label}(v) \wedge [(mwe_v \neq \text{MinEdge}(v) \wedge \text{MergeEdge}(v) \neq \emptyset) \vee (mwe_{p_v} = mwe_v \wedge \text{MergeEdge}(v) = \emptyset)]$
Then $mwe_v := \text{MinEdge}(v)$;

$R_{\lightning}$: [**MST Correction**]
If $\neg\text{Cycle}(v) \wedge \text{Label}(v) \wedge \text{NewFrag}(v)$
Then $p_v := \emptyset;\ \ell_v := (\text{Id}_v, 0)$;

To summarize, in this section we explained how to compute the outgoing-edges and the fundamental cycles (Rule R_{Min}), and how to recover from a false tree (Rule $R_{\lightning}$). The next section addresses the fragments merging operation (Rules $R_{\triangleright\triangleleft}$ and $R_{\bigcirc}$).

Fragments Merging

In this phase two rules are executed: $\mathrm{R}_{\triangleright\triangleleft}$ and $\mathrm{R}_{\bigcirc}$. Note that Rule R_{Min} (described in the previous section) computes from the leaves to the root the minimum outgoing edge $e = (u, v)$ of the fragment F_u, with $u \in F_u$. The information about e are stored in the variable mwe, i.e., the weight of the edge and a common ancestor equal to ∞ to indicate that these information concern an outgoing edge. When a root r of F_u has stabilized its variable mwe_r, it starts a merging phase (Rule $\mathrm{R}_{\triangleright\triangleleft}$). To this end, the nodes in the path between r and u are reoriented from r to v. During this reorientation the labels are locked. That is, each node x on the path between r and u (including r and excluding v) changes its label to: $\ell_v := (\bot, \bot)$. When a node u becomes the root of the fragment F_u it can merge with the fragment F_v. After the addition of the outgoing edge e, the labeling process is re-started (see Rule $\mathrm{R}_{\bigcirc}$). The merging phase is repeated until a single fragment is obtained.

$\mathrm{R}_{\triangleright\triangleleft}$: [**Merging**]
If NeedReorientation$(v) \wedge mwe_v = $ MergeEdge(v)
Then
If $(\exists u \in N(v) \backslash Child(v) \backslash \{p_v\}, w(u, v) = \text{MergeEdge}(v) \wedge \mathsf{Id}_v > \mathsf{Id}_u)$
Then
$p_v := \min\{\mathsf{Id}_u : u \in N(v) \backslash Child(v) \backslash \{p_v\} \wedge w(u, v) = \text{MergeEdge}(v)\}$;
$\ell_v := (\emptyset, \emptyset)$;
If $(\exists u \in N(v) \backslash Child(v) \backslash \{p_v\}, w(u, v) = \text{MergeEdge}(v) \wedge \mathsf{Id}_v < \mathsf{Id}_u \wedge p_u = \mathsf{Id}_v)$
Then $\ell_v := (\emptyset, \emptyset)$;
Else $p_v := \min\{\mathsf{Id}_u : u \in Child(v) \wedge mwe_v = \text{MergeEdge}(v)\}$;
$\ell_v := (\bot, \bot)$;

$\mathrm{R}_{\bigcirc}$: [**End Merging**]
If $\neg$NeedReorientation$(v) \wedge$ EndReorientation(v)
Then $\ell_v := (\emptyset, \emptyset)$;

4 Complexity Proofs

In the following we discuss the complexity issues of our solution. Due to space limitation the correctness proofs are omitted, see [16].

Lemma 1. *Algorithms* `NCA-L` *and* `MST` *have a space complexity of* $O(\log^2 n)$ *bits.*

Proof. Algorithm `NCA-L` uses three variables : $p_v, \ell_v, size_v$. The first and the last one are respectively a pointer to a neighbor node and a pair of integers, each one needs $O(\log n)$ bits. However, the variable ℓ_v is a list of pairs of integers. A new pair of integers is added to the list when a light edge is created in the tree. As noticed in [1], there are at most $\log n$ light edges on the path from a leaf to the root, i.e., at most $\log n$ pairs of integers. Thus, the variable ℓ_v uses $\log n \times \log n$ bits.

Algorithm MST uses an additional variable mwe_v which is a pair composed of an integer and the label of a node. The label of a node is stored in variable ℓ_v which uses $\log^2 n$ bits. Thus, the variable mwe_v needs $\log^2 n$ bits.

Therefore, Algorithms NCA-L and MST use $O(\log^2 n)$ bits of memory at each node. □

Lemma 2. *Starting from any configuration, all cycles are removed from the network in at most $O(n^2)$ rounds, with n the number of nodes in the network.*

Proof. As explained in the proof of Lemma 6, to break a cycle C_k a part of the nodes in C_k must compute their new labels, that is a label computation must be initiated from one node and then this process must cross C_k. Thus, the worst case is a configuration in which all the nodes in C_k have to compute their new labels using Rule R_ℓ to detect the presence of cycle C_k. Therefore, at most $O(n)$ rounds are needed to compute the new label of the nodes in C_k based on the label of one node x in C_k. According to Lemma 6, when this computation is done the cycle C_k is detected and removed by the node x. At most $O(n)$ additional rounds are needed to break the cycle C_k.

Since there is at most $n/2$ cycles in a network, at most $O(n^2)$ rounds are needed to remove all the cycles from the network. □

Lemma 3. *Starting from a configuration which contains a tree T, using Algorithm* NCA-L *any node $v \in V_T$ has a correct label in at most $O(n)$ rounds.*

Proof. According to the description of Algorithm NCA-L, the correction of node labels is done using a bottom-up computation followed by a top-down computation in the tree $T = (V_T, E_T)$.

The bottom-up computation is started by the leaves of T, when leaf nodes $v \in V_T$ have corrected their variable $size_v$ to $(1, \perp)$ then internal nodes $u \in V_T$ can start to correct their variable $size_u$. An internal node $v \in V_T$ computes a correct value in its variable $size_v$ using Rule R_ℓ when all its children u have a correct value in their variable $size_u$. Since the computation is done in a tree sub-graph then in at most $O(n)$ rounds each node $v \in V_T$ has corrected its variable $size_v$.

The top-down computation is started by the root of the tree T. When the root v has a correct value in variable $size_v$ then the computation of correct labels can start. Thus, if the parent of a node v has a correct value in its variable $size_{p_v}$ and ℓ_{p_v} then v can compute its correct label in ℓ_v using Rule R_ℓ. As for the bottom-up computation, the top-down computation is done in at most $O(n)$ rounds since it is performed in a tree sub-graph.

Therefore, in at most $O(n)$ rounds each node v in the tree T has a correct label stored in variable $size_v$. □

Lemma 4. *Starting from any configuration, Algorithm* NCA-L *reaches a legitimate configuration in at most $O(n^2)$ rounds.*

Proof. The initial configuration $\mathcal{C}$ could contain one or more cycles, so according to Lemma 2 in at most $O(n^2)$ rounds the system reaches a new configuration

$\mathcal{C}'$ which contains no cycle. Moreover according to Lemma 3, the nodes v in each tree T in the configuration $\mathcal{C}'$ have a correct label in at most $O(n)$ rounds. Therefore, starting from an arbitrary configuration each node $v \in V$ computes its correct label in at most $O(n^2)$ rounds. □

Lemma 5. *Starting from any configuration, Algorithm* `MST` *reaches a legitimate configuration in at most* $O(n^2)$ *rounds.*

Proof. According to Lemma 2, starting from any configuration after at most $O(n^2)$ rounds all the cycles are removed from the network, i.e., it remains a forest of trees after at most $O(n^2)$ rounds. Moreover, according to Lemma 3 in at most $O(n)$ additional rounds each node $v \in V$ has a correct label since each node belongs to a unique tree.

According to the description of Algorithm `MST` and Macro MinEdge(v), when it is possible to make a merging between two distinct trees in the forest a merging phase is started. This merging phase is done in three steps: (1) information corresponding to the minimum outgoing edge is propagated in a bottom-up fashion in each tree, (2) the orientation is reversed from the root of a tree until reaching the node in the tree adjacent to the minimum outgoing edge, and (3) the node labels are changed to inform of the end of the merging phase, followed by a propagation of the new correct node labels in the new tree resulting from the merging phase.

The first step is a propagation of information in a bottom-up fashion in a tree which is done in at most $O(n)$ rounds. The second step reverses and propagates new node labels on a part of the tree (between the root and the node adjacent to the minimum outgoing edge) which is done in at most $O(n)$ rounds too. Step 3 modifies the label of the nodes which have changed their parent pointer in step 2, so this last step takes also at most $O(n)$ rounds and the relabeling of the nodes in the new tree is done in at most $O(n)$ rounds according to Lemma 3. Thus, a merging phase is accomplished in at most $O(n)$ rounds and as there are in the worst case n trees then in at most $O(n^2)$ rounds a spanning tree is constructed.

When there is no possible merging for a given fragment (or tree) T_i in the forest then the correction phase concerning T_i is started. In a tree T_i, the internal edges (i.e., whose two endpoints are in T_i) are sent upward in T_i in order to detect incorrect tree edges. The internal edges e are sent following an order on the distance between the common ancestor $nca(e)$ and the root of T_i, by sending first the edge e with the nearest common ancestor $nca(e)$ from the root. Let $h(T_i)$ be the height of tree T_i and $d(v)$ be the distance from $v \in T_i$ to the root of T_i. Thus, an internal (resp. leaf) node has at most $d(v) - 2$ (resp. $d(v) - 1$) adjacent internal edges. Since a leaf node could have a lower priority (compared to its ancestors) to send all its adjacent internal edges, then the worst case to correct a tree is the case of a chain. Indeed, if the last internal edge of a leaf node x must be used to detect an incorrect tree edge then x may have to wait that all its ancestors in the chain have sent their internal edges of higher priority. Thus, starting from any configuration after at most $O(h(T_i)^2)$ rounds T_i contains no incorrect edges. Note that this is the worst case time to detect the farthest incorrect tree edge from the root of T_i, otherwise the correction phase

is stopped earlier for nearest incorrect tree edges because the merging phase has a higher priority than the correction phase. Moreover, after $O(h(T_i)^2)$ rounds all the new edges used by T_i for a merging are correct tree edges for T_i. So, T_i does not remove another tree edge in a new correction phase. Hence starting from any configuration, a correction phase deletes all the incorrect tree edges of a spanning tree after at most $O(n^2)$ rounds and no new tree edges are removed by a correction phase.

Therefore, starting from an arbitrary configuration Algorithm `MST` constructs a minimum spanning tree in at most $O(n^2)$ rounds. □

Lemma 6. *Let $\mathcal{C}$ a configuration where the set of variables $p_v, v \in V$, form at least one cycle in the network. In a finite time, Algorithm* `NCA-L` *removes all the cycles from the network.*

Proof. If a node v has a parent which is not in its neighborhood or if v has no parent then the parent and the label variable of v is modified to $\emptyset$ and $(\mathsf{Id}_v, 0)$ respectively with Rule $R_\odot$.

A node v identifies a cycle with Predicate Cycle(v) which uses v's label and the label of its parent. In a legitimate configuration, v's label is smaller than the label of its parent and is constructed using the label of its parent, i.e., the label of the parent of v is included to v's label. Thus, if the label of v is included or is smaller than the label of its parent then a cycle is detected and Predicate Cycle(v) is true. In this case, v reinitiates its parent and label variable using Rule $R_\odot$.

In order to detect a cycle the label computation process must cross a part or all the nodes of the cycle. However, since we consider a distributed scheduler then all the nodes in a cycle can be activated and we can have a rotation of the labels of the nodes in the cycle. This may lead to a new configuration in which the labels cannot be used to detect a cycle, because the label of one node is not used to compute some other labels and to detect a cycle. To break this symmetry, we use the node identifiers with Predicate MinEnabled(v). This predicate allows to activate the node v iff v has no neighbor u such that u is activated and u's identifier is lower than v. Therefore, there is at least a node x in the cycle which is not activated and when the label of x is used by some other nodes to compute their labels then Predicate Cycle(x) is true and x breaks the cycle using Rule $R_\odot$. □

5 Conclusion

We extended the Gallager, Humblet and Spira (GHS) algorithm, [9], to self-stabilizing settings via a compact informative labeling scheme. Thus, the resulting solution presents several advantages appealing for large scale systems: it is compact since it uses only logarithmic memory in the size of the network, it scales well since it does not rely on any global parameter of the system, it is fast — its time complexity is the better known in self-stabilizing settings. Additionally, it self-recovers from any transient fault. The time complexity is $O(n^2)$ rounds and the space complexity is $O(log^2 n)$.

References

1. Stephen, A., Cyril, G., Haim, K., Theis, R.: Nearest common ancestors: a survey and a new algorithm for a distributed environment. Theory of Computing Systems 37(3), 441–456 (2004)
2. Blin, L., Potop-Butucaru, M., Rovedakis, S., Tixeuil, S.: A New Self-stabilizing Minimum Spanning Tree Construction with Loop-Free Property. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 407–422. Springer, Heidelberg (2009)
3. Park, J., Masuzawa, T., Hagihara, K., Tokura, N.: Distributed Algorithms for Reconstructing MST after Topology Change. In: van Leeuwen, J., Santoro, N. (eds.) WDAG 1990. LNCS, vol. 486, pp. 122–132. Springer, Heidelberg (1991)
4. Park, J., Masuzawa, T., Hagihara, K., Tokura, N.: Efficient distributed algorithm to solve updating minimum spanning tree problem. Systems and Computers in Japan 23(3), 1–12 (1992)
5. Bein, D., Datta, A.K., Villain, V.: Self-Stablizing Pivot Interval Routing in General Networks. In: ISPAN, pp. 282–287 (2005)
6. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. ACM Commun. 17(11), 643–644 (1974)
7. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
8. Tel, G.: Introduction to distributed algorithm, 2nd edn. Cambridge University Press, Cambridge (2000)
9. Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. 5(1), 66–77 (1983)
10. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal Computing 13(2), 338–355 (1984)
11. Higham, L., Liang, Z.: Self-stabilizing minimum spanning tree construction on message-passing networks. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 194–208. Springer, Heidelberg (2001)
12. Katz, S., Perry, K.J.: Self-stabilizing extensions for message-passing systems. Distributed Computing 7, 17–26 (1993)
13. Gupta, S.K.S., Srimani, P.K.: Self-stabilizing multicast protocols for ad hoc networks. J. Parallel Distrib. Comput. 63(1), 87–96 (2003)
14. Kruskal, J.B.: On the shortest spanning subtree of a graph and the travelling salesman problem. Proc. Amer. Math. Soc. 7, 48–50 (1956)
15. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Tech. J, 1389–1401 (1957)
16. Blin, L., Dolev, S., Potop-Butucaru, M.G., Rovedakis, S.: Fast Self-Stabilizing Minimum Spanning Tree Construction. Research Report, hal-00492398, HAL (2010)

The Impact of Topology on Byzantine Containment in Stabilization*

Swan Dubois[1], Toshimitsu Masuzawa[2], and Sébastien Tixeuil[1]

[1] LIP6 - UMR 7606 Université Pierre et Marie Curie - Paris 6 & INRIA, France
[2] Osaka University, Japan

Abstract. Self-stabilization is a versatile approach to fault-tolerance since it permits a distributed system to recover from any transient fault that arbitrarily corrupts the contents of all memories in the system. Byzantine tolerance is an attractive feature of distributed systems that permits to cope with arbitrary malicious behaviors.

We consider the well known problem of constructing a maximum metric tree in this context. Combining these two properties proves difficult: we demonstrate that it is impossible to contain the impact of Byzantine nodes in a self-stabilizing context for maximum metric tree construction (strict stabilization). We propose a weaker containment scheme called *topology-aware strict stabilization*, and present a protocol for computing maximum metric trees that is optimal for this scheme with respect to impossibility result.

Keywords: Byzantine fault, Distributed protocol, Fault tolerance, Stabilization, Spanning tree construction.

1 Introduction

The advent of ubiquitous large-scale distributed systems advocates that tolerance to various kinds of faults and hazards must be included from the very early design of such systems. *Self-stabilization* [1,2,3] is a versatile technique that permits forward recovery from any kind of *transient* faults, while *Byzantine Fault-tolerance* [4,5] is traditionally used to mask the effect of a limited number of *malicious* faults. Making distributed systems tolerant to both transient and malicious faults is appealing yet proved difficult [6,7,8] as impossibility results are expected in many cases.

Two main paths have been followed to study the impact of Byzantine faults in the context of self-stabilization:
- *Byzantine fault masking* (any correct processes eventually satisfy its specification). In completely connected synchronous systems, one of the most studied problems in the context of self-stabilization with Byzantine faults is that of *clock*

* This work has been supported in part by ANR projects SHAMAN, ALADDIN, SPADES, by MEXT Global COE Program and by JSPS Grant-in-Aid for Scientific Research ((B) 22300009).

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 495–509, 2010.

synchronization. In [9,6], probabilistic self-stabilizing protocols were proposed for up to one third of Byzantine processes, while in [10,11] deterministic solutions tolerate up to one fourth and one third of Byzantine processes, respectively.

- *Byzantine containment*. For *local* tasks (*i.e.* tasks whose correctness can be checked locally, such as vertex coloring, link coloring, or dining philosophers), the notion of *strict stabilization* was proposed [8,12,13,14]. Strict stabilization guarantees that there exists a *containment radius* outside which the effect of permanent faults is masked, provided that the problem specification makes it possible to break the causality chain that is caused by the faults. As many problems are not local, it turns out that it is impossible to provide strict stabilization for those. Note that a strictly stabilizing algorithm with a radius of 0 which runs on a completely connected system provides a masking approach.

Our Contribution. In this paper, we investigate the possibility of Byzantine containment in a self-stabilizing setting for tasks that are global (*i.e.* for which there exists a causality chain of size r, where r depends on n the size of the network), and focus on a global problem, namely maximum metric tree construction (see [15,16]). As strict stabilization is impossible with such global tasks, we weaken the containment constraint by relaxing the notion of containment radius to containment area, that is Byzantine processes may disturb infinitely often a set of processes which depends on the topology of the system and on the location of Byzantine processes.

The main contribution of this paper is to present new possibility results for containing the influence of unbounded Byzantine behaviors. In more details, we define the notion of *topology-aware strict stabilization* as the novel form of the containment and introduce *containment area* to quantify the quality of the containment. The notion of topology-aware strict stabilization is weaker than the strict stabilization but is stronger than the classical notion of self-stabilization (*i.e.* every topology-aware strictly stabilizing protocol is self-stabilizing, but not necessarily strictly stabilizing).

To demonstrate the possibility and effectiveness of our notion of topology-aware strict stabilization, we consider *maximum metric tree construction*. It is shown in [8] that there exists no strictly stabilizing protocol with a constant containment radius for this problem. In this paper, we provide a topology-aware strictly stabilizing protocol for maximum metric tree construction and we prove that the containment area of this protocol is optimal.

2 Distributed System

A *distributed system* $S = (V, E)$ consists of a set $V = \{v_1, v_2, \ldots, v_n\}$ of processes and a set E of bidirectional communication links (simply called links). A link is an unordered pair of distinct processes. A distributed system S can be regarded as a graph whose vertex set is V and whose link set is E, so we use graph terminology to describe a distributed system S.

Processes u and v are called *neighbors* if $(u, v) \in E$. The set of neighbors of a process v is denoted by N_v, and its cardinality (the *degree* of v) is denoted

by $\Delta_v(= |N_v|)$. The degree Δ of a distributed system $S = (V, E)$ is defined as $\Delta = \max\{\Delta_v \mid v \in V\}$. We do not assume existence of a unique identifier for each process. Instead we assume each process can distinguish its neighbors from each other by locally arranging them in some arbitrary order: the k-th neighbor of a process v is denoted by $N_v(k)$ $(1 \leq k \leq \Delta_v)$. The *distance* between two processes u and v is the length of the shortest path between u and v.

In this paper, we consider distributed systems of arbitrary topology. We assume that a single process is distinguished as a *root*, and all the other processes are not distinguishable.

We adopt the *shared state model* as a communication model in this paper, where each process can directly read the states of its neighbors.

The variables that are maintained by processes denote process states. A process may take actions during the execution of the system. An action is simply a function that is executed in an atomic manner by the process. The actions executed by each process are described by a finite set of guarded actions of the form $\langle \text{guard} \rangle \longrightarrow \langle \text{statement} \rangle$. Each guard of process u is a boolean expression involving the variables of u and its neighbors.

A global state of a distributed system is called a *configuration* and is specified by a product of states of all processes. We define C to be the set of all possible configurations of a distributed system S. For a process set $R \subseteq V$ and two configurations ρ and ρ', we denote $\rho \overset{R}{\mapsto} \rho'$ when ρ changes to ρ' by executing an action of each process in R simultaneously. Notice that ρ and ρ' can be different only in the states of processes in R. For completeness of execution semantics, we should clarify the configuration resulting from simultaneous actions of neighboring processes. The action of a process depends only on its state at ρ and the states of its neighbors at ρ, and the result of the action reflects on the state of the process at ρ'.

A *schedule* of a distributed system is an infinite sequence of process sets. Let $Q = R^1, R^2, \ldots$ be a schedule, where $R^i \subseteq V$ holds for each i $(i \geq 1)$. An infinite sequence of configurations $e = \rho_0, \rho_1, \ldots$ is called an *execution* from an initial configuration ρ_0 by a schedule Q, if e satisfies $\rho_{i-1} \overset{R^i}{\mapsto} \rho_i$ for each i $(i \geq 1)$. Process actions are executed atomically, and we also assume that a *distributed daemon* schedules the actions of processes, i.e. any subset of processes can simultaneously execute their actions. A more constrained daemon is the *central* one which must choose only one enabled process at each step. Note that, as the central daemon allows executions that are also allowed under the distributed daemon, an impossibility result under the central daemon is stronger than one under the distributed one. In the same way, a possibility result under the distributed daemon is stronger than one under the central one.

The set of all possible executions from $\rho_0 \in C$ is denoted by E_{ρ_0}. The set of all possible executions is denoted by E, that is, $E = \bigcup_{\rho \in C} E_\rho$. We consider *asynchronous* distributed systems where we can make no assumption on schedules except that any schedule is *weakly fair*: every process is contained in infinite number of subsets appearing in any schedule.

In this paper, we consider (permanent) *Byzantine faults*: a Byzantine process (i.e. a Byzantine-faulty process) can make arbitrary behavior independently from its actions. In other words, a Byzantine process has always an enabled rule and the daemon arbitrarily chooses a new state for this process when this process is activated. If v is a Byzantine process, v can repeatedly change its variables arbitrarily. The only restriction we do on Byzantine processes is that the root process can never be Byzantine.

3 Self-Stabilizing Protocol Resilient to Byzantine Faults

Problems considered in this paper are so-called *static problems*, i.e. they require the system to find static solutions. For example, the spanning-tree construction problem is a static problem, while the mutual exclusion problem is not. Some static problems can be defined by a *local specification predicate* (shortly, specification), $spec(v)$, for each process v: a configuration is a desired one (with a solution) if every process $v \in V$ satisfies $spec(v)$ in this configuration. A specification $spec(v)$ is a boolean expression on variables of V_v ($\subseteq V$) where V_v is the set of processes whose variables appear in $spec(v)$. The variables appearing in the specification are called *output variables* (shortly, *O-variables*). In what follows, we consider a static problem defined by a local specification predicate.

Self-Stabilization. A *self-stabilizing protocol* ([1]) is a protocol that eventually reaches a *legitimate configuration*, where $spec(v)$ holds at every process v, regardless of the initial configuration. Once it reaches a legitimate configuration, every process never changes its O-variables and always satisfies $spec(v)$. From this definition, a self-stabilizing protocol is expected to tolerate any number and any type of transient faults since it can eventually recover from any configuration affected by the transient faults. However, the recovery from any configuration is guaranteed only when every process correctly executes its action from the configuration, i.e., we do not consider existence of permanently faulty processes.

Strict stabilization. When (permanent) Byzantine processes exist, Byzantine processes may not satisfy $spec(v)$. In addition, correct processes near the Byzantine processes can be influenced and may be unable to satisfy $spec(v)$. Nesterenko and Arora [8] define a *strictly stabilizing protocol* as a self-stabilizing protocol resilient to unbounded number of Byzantine processes.

Given an integer c, a *c-correct process* is a process defined as follows.

Definition 1 (c-correct process). *A process is c-correct if it is correct (*i.e. *not Byzantine) and located at distance more than c from any Byzantine process.*

Definition 2 ((c, f)-containment). *A configuration ρ is* (c, f)-contained *for specification spec if, given at most f Byzantine processes, in any execution starting from ρ, every c-correct process v always satisfies $spec(v)$ and never changes its O-variables.*

The parameter c of Definition 2 refers to the *containment radius* defined in [8]. The parameter f refers explicitly to the number of Byzantine processes, while [8] dealt with unbounded number of Byzantine faults (that is $f \in \{0 \dots n\}$).

Definition 3 ((c, f)-strict stabilization). *A protocol is* (c, f)-strictly stabilizing *for specification spec if, given at most f Byzantine processes, any execution $e = \rho_0, \rho_1, \ldots$ contains a configuration ρ_i that is (c, f)-contained for spec.*

An important limitation of the model of [8] is the notion of *r-restrictive* specifications. Intuitively, a specification is r-restrictive if it prevents combinations of states that belong to two processes u and v that are at least r hops away. An important consequence related to Byzantine tolerance is that the containment radius of protocols solving those specifications is at least r. For any global problem, such as the spanning tree construction we consider in this paper, r can not be bounded to a constant. The results of [8] show us that there exists no $(o(n), 1)$-strictly stabilizing protocol for these problems and especially for the spanning tree construction.

Topology-aware strict stabilization. In the former paragraph, we saw that there exist a number of impossibility results on strict stabilization due to the notion of r-restrictive specifications. To circumvent this impossibility result, we define here a new notion, which is weaker than the strict stabilization: the *topology-aware strict stabilization* (denoted by TA-strict stabilization for short). Here, the requirement to the containment radius is relaxed, *i.e.* the set of processes which may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine processes but can be defined depending on the topology of the system and on Byzantine processes location.

In the following, we give formal definition of this new kind of Byzantine containment. From now, B denotes the set of Byzantine processes and S_B (which is a function of B) denotes a subset of V (intuitively, this set gathers all processes which may be disturbed by Byzantine processes).

Definition 4 (S_B-correct node). *A node is* S_B-correct *if it is a correct node (*i.e. *not Byzantine) which does not belong to S_B.*

Definition 5 (S_B-legitimate configuration). *A configuration ρ is* S_B-legitimate *for spec if every S_B-correct node v is legitimate for spec (*i.e. *if spec(v) holds).*

Definition 6 ((S_B, f)-topology-aware containment). *A configuration ρ_0 is* (S_B, f)-topology-aware contained *for specification spec if, given at most f Byzantine processes, in any execution $e = \rho_0, \rho_1, \ldots$, every configuration is S_B-legitimate and every S_B-correct process never changes its O-variables.*

The parameter S_B of Definition 6 refers to the *containment area*. Any process which belongs to this set may be infinitely disturbed by Byzantine processes. The parameter f refers explicitly to the number of Byzantine processes.

Definition 7 ((S_B, f)-topology-aware strict stabilization). *A protocol is* (S_B, f)-topology aware strictly stabilizing *for specification spec if, given at most f Byzantine processes, any execution $e = \rho_0, \rho_1, \ldots$ contains a configuration ρ_i that is (S_B, f)-topology-aware contained for spec.*

Note that, if B denotes the set of Byzantine processes and $S_B = \{v \in V | min\{d(v,b), b \in B\} \leq c\}$, then a (S_B, f)-topology-aware strictly stabilizing protocol is a (c, f)-strictly stabilizing protocol. Then, a TA-strictly stabilizing protocol is generally weaker than a strictly stabilizing one, but stronger than a classical self-stabilizing protocol (that may never meet its specification in the presence of Byzantine processes).

The parameter S_B is introduced to quantify the strength of fault containment, we do not require each process to know the actual definition of the set. Actually, the protocol proposed in this paper assumes no knowledge on this parameter.

4 Maximum Metric Tree Construction

In this work, we deal with maximum (routing) metric spanning trees as defined in [16] (note that [15] provides a self-stabilizing solution to this problem). Informally, the goal of a routing protocol is to construct a tree that simultaneously maximizes the metric values of all of the nodes with respect to some total ordering $\prec$. In [16], authors give a general definition of a routing metric and provide a characterization of *maximizable metrics*, that is metrics which always allow to construct a maximum (routing) metric spanning trees. In the following, we recall all definitions and notations introduced in [16].

Definition 8 (Routing metric). *A* routing metric *is a five-tuple* $(M, W, met, mr, \prec)$ *where:*
- M is a set of metric values,
- W is a set of edge weights,
- met is a metric function whose domain is $M \times W$ and whose range is M,
- mr is the maximum metric value in M with respect to $\prec$ and is assigned to the root of the system,
- $\prec$ is a less-than total order relation over M that satisfies the following three conditions for arbitrary metric values m, m', and m'' in M:
  - irreflexivity: $m \not\prec m$,
  - transitivity : if $m \prec m'$ and $m' \prec m''$ then $m \prec m''$,
  - totality: $m \prec m'$ or $m' \prec m$ or $m = m'$.

Any metric value $m \in M \setminus \{mr\}$ satisfies the utility condition *(that is, there exist $w_0, \ldots, w_{k-1}$ in W and $m_0 = mr, m_1, \ldots, m_{k-1}, m_k = m$ in M such that $\forall i \in \{1, \ldots, k\}, m_i = met(m_{i-1}, w_{i-1})$).*

For instance, we provide the definition of three classical metrics with this model: the shortest path metric ($\mathcal{SP}$), the flow metric ($\mathcal{F}$), and the reliability metric ($\mathcal{R}$).

$\mathcal{SP} = (M_1, W_1, met_1, mr_1, \prec_1)$
where $M_1 = \mathbb{N}$
$W_1 = \mathbb{N}$
$met_1(m, w) = m + w$
$mr_1 = 0$
$\prec_1$ is the classical $>$ relation

$\mathcal{F} = (M_2, W_2, met_2, mr_2, \prec_2)$
where $mr_2 \in \mathbb{N}$
$M_2 = \{0, \ldots, mr_2\}$
$W_2 = \{0, \ldots, mr_2\}$
$met_2(m, w) = min\{m, w\}$
$\prec_2$ is the classical $<$ relation

$$\begin{aligned} \mathcal{R} &= (M_3, W_3, met_3, mr_3, \prec_3) \\ \text{where} \quad & M_3 = [0,1] \\ & W_3 = [0,1] \\ & met_3(m,w) = m * w \\ & mr_3 = 1 \\ & \prec_3 \text{ is the classical } < \text{ relation} \end{aligned}$$

Definition 9 (Assigned metric). *An* assigned metric *over a system S is a six-tuple $(M, W, met, mr, \prec, wf)$ where $(M, W, met, mr, \prec)$ is a metric and wf is a function that assigns to each edge of S a weight in W.*

Let a rooted path (from v) be a simple path from a process v to the root r. The next set of definitions are with respect to an assigned metric $(M, W, met, mr, \prec, wf)$ over a given system S.

Definition 10 (Metric of a rooted path). *The* metric of a rooted path *in S is the prefix sum of met over the edge weights in the path and mr.*

For example, if a rooted path p in S is $v_k, \ldots, v_0$ with $v_0 = r$, then the metric of p is $m_k = met(m_{k-1}, wf(\{v_k, v_{k-1}\}))$ with $\forall i \in \{1, k-1\}, m_i = met(m_{i-1}, wf(\{v_i, v_{i-1}\}))$ and $m_0 = mr$.

Definition 11 (Maximum metric path). *A rooted path p from v in S is called a* maximum metric path *with respect to an assigned metric if and only if for every other rooted path q from v in S, the metric of p is greater than or equal to the metric of q with respect to the total order $\prec$.*

Definition 12 (Maximum metric of a node). *The* maximum metric of a node $v \neq r$ *(or simply* metric value *of v) in S is defined by the metric of a maximum metric path from v. The maximum metric of r is mr.*

Definition 13 (Maximum metric tree). *A spanning tree T of S is a* maximum metric tree *with respect to an assigned metric over S if and only if every rooted path in T is a maximum metric path in S with respect to the assigned metric.*

The goal of the work of [16] is the study of metrics that always allow the construction of a maximum metric tree. More formally, the definition follows.

Definition 14 (Maximizable metric). *A metric is* maximizable *if and only if for any assignment of this metric over any system S, there is a maximum metric tree for S with respect to the assigned metric.*

Note that [15] provides a self-stabilizing protocol to construct a maximum metric tree with respect to any maximizable metric. Moreover, [16] provides a fully characterization of maximazable metrics as follow.

Definition 15 (Boundedness). *A metric $(M, W, met, mr, \prec)$ is* bounded *if and only if: $\forall m \in M, \forall w \in W, met(m,w) \prec m$ or $met(m,w) = m$*

Definition 16 (Monotonicity). *A metric* $(M, W, met, mr, \prec)$ *is* monotonic *if and only if:* $\forall (m,\ m') \in M^2, \forall w \in W, m \prec m' \Rightarrow (met(m,w) \prec met(m',w)$ *or* $met(m,w) = met(m',w))$

Theorem 1 (Characterization of maximizable metrics [16]). *A metric is maximizable if and only if this metric is bounded and monotonic.*

Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, the aim of this work is to construct a maximum metric tree with respect to $\mathcal{M}$ which spans the system in a self-stabilizing way in a system subject to permanent Byzantine failures. It is obvious that these Byzantine processes may disturb some correct processes. It is why, we relax the problem in the following way: we want to construct a maximum metric forest with respect to $\mathcal{M}$. The root of any tree of this forest must be either the real root or a Byzantine process.

Each process v has three O-variables: a pointer to its parent in its tree ($prnt_v \in N_v \cup \{\perp\}$), a level which stores its current metric value ($level_v \in M$), and a variable which stores its distance to the root of its tree ($dist_v \in \{0, \ldots, D\}$). Obviously, Byzantine process may disturb (at least) their neighbors. We use the following specification of the problem.

We introduce new notations as follows. Given an assigned metric $(M, W, met, mr, \prec, wf)$ over the system S and two processes u and v, we denote by $\mu(u, v)$ the maximum metric of node u when v plays the role of the root of the system. If u and v are two neighor processes, we denote by $w_{u,v}$ the weight of the edge $\{u, v\}$ (that is, the value of $wf(\{u, v\})$).

Definition 17 ($\mathcal{M}$-path). *Given an assigned metric* $\mathcal{M} = (M, W, mr, met, \prec, wf)$ *over a system* S*, a* $\mathcal{M}$-path *is a path* $(v_0, \ldots, v_k)$ $(k \geq 1)$ *such that: (i)* $prnt_{v_0} = \perp$*,* $level_{v_0} = mr$*,* $dist_{v_0} = 0$*, and* $v_0 \in B \cup \{r\}$*, (ii)* $\forall i \in \{1, \ldots, k\}, prnt_{v_i} = v_{i-1}$*,* $level_{v_i} = met(level_{v_{i-1}}, w_{v_i,v_{i-1}})$*, and* $dist_{v_i} = i$*, (iii)* $\forall i \in \{1, \ldots, k\}, met(level_{v_{i-1}}, w_{v_i,v_{i-1}}) = max_{\prec}\{met(level_u, w_{v_i,u}), u \in N_{v_i}\}$*, and (iv)* $level_{v_k} = \mu(v_k, v_0)$.

We define the specification predicate $spec(v)$ of the maximum metric tree construction with respect to a maximizable metric $\mathcal{M}$ as follows.

$$spec(v) : \begin{cases} prnt_v = \perp, level_v = mr, \text{ and } dist_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a } \mathcal{M}\text{-path } (v_0, \ldots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

Following discussion of Section 3 and results from [8], it is obvious that there exists no strictly stabilizing protocol for this problem. It is why we consider the weaker notion of topology-aware strict stabilization. First, we show an impossibility result in order to define the best possible containment area. Then, we provide a maximum metric tree construction protocol which is (S_B, f)-TA-strictly stabilizing where $f \leq n - 1$ which matches these optimal containment area. From now, S_B denotes this optimal containment area, *i.e.*:

$$S_B = \{v \in V \setminus B \,|\mu(v, r) \preceq max_{\prec}\{\mu(v, b), b \in B\}\} \setminus \{r\}$$

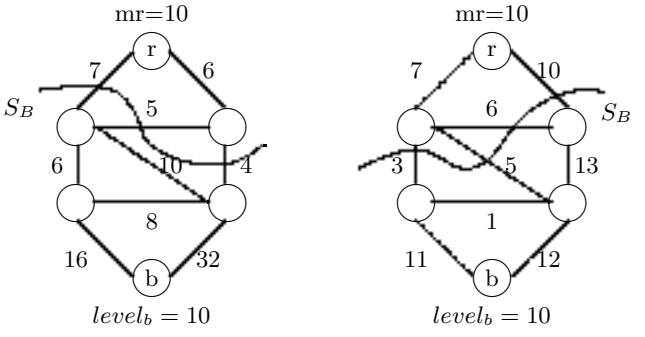


Fig. 1. Examples of containment areas for flow spanning tree construction

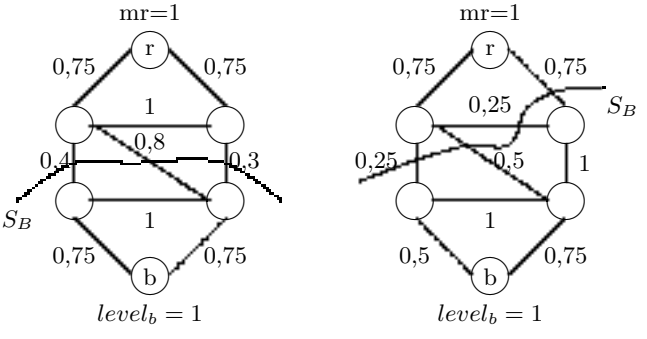


Fig. 2. Examples of containment areas for reliability spanning tree construction

Intuitively, Byzantine faults may disturb only processes that are (non strictly) closer from a Byzantine process than the root with respect to the metric. Figures 1 and 2 provide some examples of containment areas with respect to two maximizable metrics.

We introduce here a new definition that is used in the following.

Definition 18 (Fixed point). *A metric value m is a* fixed point *of a metric $\mathcal{M} = (M, W, mr, met, \prec)$ if $m \in M$ and if for all value $w \in W$, we have: $met(m, w) = m$.*

4.1 Impossibility Result

In this section, we show that there exist some constraints on the containment area of any topology-aware strictly stabilizing protocol for the maximum metric tree construction depending on the metric.

Theorem 2. *Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, even under the central daemon, there exists no $(A_B, 1)$-TA-strictly stabilizing protocol for maximum metric spanning tree construction with respect to $\mathcal{M}$ where $A_B \subsetneq S_B$.*

Proof. Let $\mathcal{M} = (M, W, mr, met, \prec)$ be a maximizable metric and $\mathcal{P}$ be a $(A_B, 1)$-TA-strictly stabilizing protocol for maximum metric spanning tree construction protocol with respect to $\mathcal{M}$ where $A_B \subsetneq S_B$. We must distinguish the following cases:

Case 1: $|M| = 1$. Denote by m the metric value such that $M = \{m\}$. For any system and for any process $v \neq r$, we have $\mu(v, r) = min_{\prec}\{\mu(v, b), b \in B\} = m$. Consequently, $S_B = V \setminus (B \cup \{r\})$ for any system. Consider the following system: $V = \{r, u, v, b\}$ and $E = \{\{r, u\}, \{u, v\}, \{v, b\}\}$ (b is a Byzantine process). As $S_B = \{u, v\}$ and $A_B \subsetneq S_B$, we have: $u \notin A_B$ or $v \notin A_B$. Consider now the following configuration ρ_0^0: $prnt_r = prnt_b = \perp$, $prnt_v = b$, $prnt_u = v$, $level_r = level_u = level_v = level_b = m$, $dist_r = dist_b = 0$, $dist_v = 1$ and $dist_u = 2$ (other variables may have arbitrary values). Note that ρ_0^0 is A_B-legitimate for *spec* (whatever A_B is). Assume now that b behaves as a correct process with respect to $\mathcal{P}$. Then, by convergence of $\mathcal{P}$ in a fault-free system starting from ρ_0^0 which is not legitimate (remember that a strictly-stabilizing protocol is a special case of a self-stabilizing protocol), we can deduce that the system reaches in a finite time a configuration ρ_1^0 in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = u$, $prnt_b = v$, $level_r = level_u = level_v = level_b = m$, $dist_r = 0$, $dist_u = 1$, $dist_v = 2$ and $dist_b = 3$. Note that processes u and v modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA-strict stabilization of $\mathcal{P}$ (whatever A_B is).

Case 2: $|M| \geq 2$. By definition of a bounded metric, we can deduce that there exist $m \in M$ and $w \in W$ such that $m = met(mr, w) \prec mr$. Then, we must distinguish the following cases:

Case 2.1: m is a fixed point of $\mathcal{M}$. Consider the following system: $V = \{r, u, v, b\}$, $E = \{\{r, u\}, \{u, v\}, \{v, b\}\}$, $w_{r,u} = w_{v,b} = w$, and $w_{u,v} = w'$ (b is a Byzantine process). As for any $w' \in W$, $met(m, w') = m$ (by definition of a fixed point), we have: $S_B = \{u, v\}$. Since $A_B \subsetneq S_B$, we have: $u \notin A_B$ or $v \notin A_B$. Consider now the following configuration ρ_0^1: $prnt_r = prnt_b = \perp$, $prnt_v = b$, $prnt_u = v$, $level_r = level_b = mr$, $level_u = level_v = m$, $dist_r = dist_b = 0$, $dist_v = 1$ and $dist_u = 2$ (other variables may have arbitrary values). Note that ρ_0^1 is A_B-legitimate for *spec* (whatever A_B is). Assume now that b behaves as a correct process with respect to $\mathcal{P}$. Then, by convergence of $\mathcal{P}$ in a fault-free system starting from ρ_0^1 which is not legitimate (remember that a strictly-stabilizing protocol is a special case of a self-stabilizing protocol), we can deduce that the system reaches in a finite time a configuration ρ_1^1 in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = u$, $prnt_b = v$, $level_r = mr$, $level_u = level_v = level_b = m$ (since m is a fixed point), $dist_r = 0$, $dist_u = 1$, $dist_v = 2$ and $dist_b = 3$. Note that processes u and v modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA-strict stabilization of $\mathcal{P}$ (whatever A_B is).

Case 2.2: m is not a fixed point of $\mathcal{M}$. This implies that there exists $w' \in W$ such that: $met(m, w') \prec m$ (remember that $\mathcal{M}$ is bounded). Consider the following system: $V = \{r, u, v, v', b\}$, $E = \{\{r, u\}, \{u, v\}, \{u, v'\}, \{v, b\}, \{v', b\}\}$, $w_{r,u} = w_{v,b} = w_{v',b} = w$, and $w_{u,v} = w_{u,v'} = w'$ (b is a Byzantine process). We can see that $S_B = \{v, v'\}$. Since $A_B \subsetneq S_B$, we have: $v \notin A_B$ or $v' \notin A_B$. Consider

now the following configuration ρ_0^2: $prnt_r = prnt_b = \perp$, $prnt_v = prnt_{v'} = b$, $prnt_u = r$, $level_r = level_b = mr$, $level_u = level_v = level_{v'} = m$, $dist_r = dist_b = 0$, $dist_v = dist_{v'} = 1$ and $dist_u = 1$ (other variables may have arbitrary values). Note that ρ_0^2 is A_B-legitimate for *spec* (whatever A_B is). Assume now that b behaves as a correct process with respect to $\mathcal{P}$. Then, by convergence of $\mathcal{P}$ in a fault-free system starting from ρ_0^2 which is not legitimate (remember that a strictly-stabilizing protocol is a special case of a self-stabilizing protocol), we can deduce that the system reaches in a finite time a configuration ρ_1^2 in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = prnt_{v'} = u$, $prnt_b = v$ (or $prnt_b = v'$), $level_r = mr$, $level_u = m$ $level_v = level_{v'} = met(m, w') = m'$, $level_b = met(m', w) = m''$, $dist_r = 0$, $dist_u = 1$, $dist_v = dist_{v'} = 2$ and $dist_b = 3$. Note that processes v and v' modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA-strict stabilization of $\mathcal{P}$ (whatever A_B is).

4.2 Topology-Aware Strict Stabilizing Protocol

In this section, we provide our self-stabilizing protocol that achieves optimal containment areas to permanent Byzantine failures for constructing a maximum metric tree for any maximizable metric $\mathcal{M} = (M, W, met, mr, \prec)$. More formally, our protocol is (S_B, f)-strictly stabilizing, that is optimal with respect to the result of Theorem 2. Our protocol is borrowed from the one of [15] (which is self-stabilizing). The key idea of this protocol is to use the distance variable (upper bounded by a given constant D) to detect and break cycles of processes which have the same maximum metric. The main modifications we bring to this protocol follow. In the initial protocol, when a process modifies its parent, it chooses arbitrarily one of the "better" neighbors (with respect to the metric). To achieve the (S_B, f)-TA-strict stabilization, we must ensure a fair selection along the set of its neighbor. We perform this fairness with a round-robin order along the set of neighbors. The second modification is to give priority to rules $(\boldsymbol{R_2})$ and $(\boldsymbol{R_3})$ over $(\boldsymbol{R_1})$ for any correct non root process (that is, such a process which has $(\boldsymbol{R_1})$ and another rule enabled in a given configuration always executes the other rule if it is activated). Our solution is presented as Algorithm 4.1.

In the following, we provide a sketch[1] of the proof of the TA-strict stabilization of $\mathcal{SSMAX}$. Remember that the real root r can not be a Byzantine process by hypothesis. Note that the subsystem whose set of nodes is $(V \setminus S_B) \setminus B$ is connected respectively by boundedness of the metric.

Given $\rho \in C$ and $m \in M$, let us define the following predicate:

$$IM_m(\rho) \equiv \forall v \in V, level_v \preceq max_{\prec}\{m, max_{\prec}\{\mu(v, u), u \in B \cup \{r\}\}\}$$

If we take a configuration $\rho \in \mathcal{C}$ such that $IM_m(\rho)$ holds for a given $m \in M$, then we can prove that the boundedness of $\mathcal{M}$ implies that, for any step $\rho \mapsto \rho'$ of $\mathcal{SSMAX}$, $IM_m(\rho')$ holds. Hence, we can deduce that:

Lemma 1. *For any metric value $m \in M$, the predicate IM_m is closed by actions of $\mathcal{SSMAX}$.*

[1] Due to the lack of place, formal proofs are omitted. A full version of this work is available in the companion technical report (see [17]).

algorithm 4.1. $\mathcal{SSMAX}$: A TA-strictly stabilizing protocol for maximum metric tree construction.

Data:
N_v: totally ordered set of neighbors of v.
D: upper bound of the number of processes in a simple path.
Variables:
$prnt_v \begin{cases} = \perp \text{ if } v = r \\ \in N_v \text{ if } v \neq r \end{cases}$: pointer on the parent of v in the tree.
$level_v \in \{m \in M | m \preceq mr\}$: metric of the node.
$dist_v \in \{0, \ldots, D\}$: distance to the root.
Macro:
For any subset $A \subseteq N_v$, $choose(A)$ returns the first element of A which is bigger than $prnt_v$ (in a round-robin fashion).
Rules:
$(\boldsymbol{R_r})$ :: $(v = r) \wedge ((level_v \neq mr) \vee (dist_v \neq 0)) \longrightarrow level_v := mr;\ dist_v := 0$
$(\boldsymbol{R_1})$ :: $(v \neq r) \wedge (prnt_v \in N_v) \wedge$
$((dist_v \neq min(dist_{prnt_v} + 1, D)) \vee (level_v \neq met(level_{prnt_v}, w_{v,prnt_v})))$
$\longrightarrow dist_v := min(dist_{prnt_v} + 1, D); level_v := met(level_{prnt_v}, w_{v,prnt_v})$
$(\boldsymbol{R_2})$ :: $(v \neq r) \wedge (dist_v = D) \wedge (\exists u \in N_v, dist_u < D - 1)$
$\longrightarrow prnt_v := choose(\{u \in N_v | dist_v < D - 1\});\ dist_v := dist_{prnt_v} + 1;$
$level_v := met(level_{prnt_v}, w_{v,prnt_v})$
$(\boldsymbol{R_3})$ :: $(v \neq r) \wedge (\exists u \in N_v, (dist_u < D - 1) \wedge (level_v \prec met(level_u, w_{u,v})))$
$\longrightarrow prnt_v := choose\Big(\Big\{u \in N_v \Big|$
$(level_u < D - 1) \wedge (met(level_u, w_{u,v}) = \underset{q \in N_v / level_q < D-1}{max_{\prec}} \{met(level_q, w_{q,v})\})\Big\}\Big);$
$level_v := met(level_{prnt_v}, w_{prnt_v,v});\ dist_v := dist_{prnt_v} + 1$

Given an assigned metric to a system S, observe that the set of metrics value M is finite and that we can label elements of M by $m_0 = mr, m_1, \ldots, m_k$ such that $\forall i \in \{0, \ldots, k-1\}, m_{i+1} \prec m_i$. We introduce the following notations:

$$\forall m_i \in M,\ P_{m_i} = \left\{v \in (V \setminus S_B) \setminus B \middle| \mu(v, r) = m_i\right\}$$

$$\begin{aligned}
\forall m_i \in M, \quad V_{m_i} &= \bigcup_{j=0}^{i} P_{m_j} \\
\forall m_i \in M, \quad I_{m_i} &= \left\{v \in V \middle| max_{\prec}\{\mu(v, u), u \in B \cup \{r\}\} \prec m_i\right\} \\
\forall m_i \in M, \mathcal{LC}_{m_i} &= \left\{\rho \in \mathcal{C} \middle| (\forall v \in V_{m_i}, spec(v)) \wedge (IM_{m_i}(\rho))\right\} \\
\mathcal{LC} &= \mathcal{LC}_{m_k}
\end{aligned}$$

If we consider a configuration $\rho \in \mathcal{LC}_{m_i}$ for a given metric value m_i and a process $v \in V_{m_i}$, then we can show from the closure of IM_{m_i} (established in Lemma 1), the boundedness of $\mathcal{M}$ and the construction of the protocol that v is not enabled in ρ. Then, the closure of IM_{m_i} is sufficient to conclude that:

Lemma 2. *For any $m_i \in M$, the set $\mathcal{LC}_{m_i}$ is closed by actions of $\mathcal{SSMAX}$.*

Lemma 2 applied to $\mathcal{LC} = \mathcal{LC}_{m_k}$ gives us the following result:

Lemma 3. *Any configuration of $\mathcal{LC}$ is $(S_B, n-1)$-TA contained for spec.*

This lemma establishes the closure of $\mathcal{SSMAX}$. To prove the TA strict stabilization of $\mathcal{SSMAX}$, it remains to prove its convergence. In this goal, we prove

that any execution starting from an arbitrary configuration of C converges to $\mathcal{LC}_{m_0} = \mathcal{LC}_{mr}$ and then to $\mathcal{LC}_{m_1}$ and so on until $\mathcal{LC}_{m_k} = \mathcal{LC}$.

Note that IM_{mr} is satisfied by any configuration of C and that if all processes of P_{mr} are not enabled in a configuration then this configuration belongs to $\mathcal{LC}_{mr}$. Then, we can prove that any process of P_{mr} takes only a finite number of steps in any execution. This implies the following result:

Lemma 4. *Starting from any configuration of $\mathcal{C}$, any execution of $\mathcal{SSMAX}$ reaches in a finite time a configuration of $\mathcal{LC}_{mr}$.*

Given a metric value $m_i \in M$ and a configuration $\rho_0 \in \mathcal{LC}_{m_i}$, assume that $e = \rho_0, \rho_1, \ldots$ is an execution of $\mathcal{SSMAX}$ starting from ρ_0. We define then the following variant function. For any configuration ρ_j of e, we denote by A_j the set of processes v of I_{m_i} such that $level_v = m_i$ in ρ_j. Then, we define $f(\rho_j) = min\{dist_v, v \in A_j\}$. We can prove that there exists an integer k such that $f(\rho_k) = D$. This implies the following lemma:

Lemma 5. *For any $m_i \in M$ and for any configuration $\rho \in \mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMAX}$ starting from ρ reaches in a finite time a configuration such that $\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$.*

Given a metric value $m_i \in M$, consider a configuration $\rho_0 \in \mathcal{LC}_{m_i}$ such that $\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$. Assume that $e = \rho_0, \rho_1, \ldots$ is an execution of $\mathcal{SSMAX}$ starting from ρ_0. For any configuration ρ_i of e, we define the following set $E_{\rho_i} = \{v \in I_{m_i} | level_v = m_i\}$. First, we prove that there exists an integer k such that for any integer $j \geq k$, we have $E_{\rho_{j+1}} \subseteq E_{\rho_j}$. In other words, there exists a point of the execution afterwards the set E can not grow. Moreover, we prove that if a process of E_{ρ_j} $(j \geq k)$ is activated during the step $\rho_j \mapsto \rho_{j+1}$, then it satisfies $v \notin E_{\rho_{j+1}}$. Finally, we observe that any process $v \in I_{m_i}$ such that $dist_v = D$ is activated in a finite time. In conclusion, we obtain that there exists an integer j such that $E_{\rho_j} = \emptyset$. In other words, we have:

Lemma 6. *For any $m_i \in M$ and for any configuration $\rho \in \mathcal{LC}_{m_i}$ such that $\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$, any execution of $\mathcal{SSMAX}$ starting from ρ reaches in a finite time a configuration such that $\forall v \in I_{m_i}, level_v \prec m_i$.*

A direct consequence of Lemmas 5 and 6 is the following.

Lemma 7. *For any $m_i \in M$ and for any configuration $\rho \in \mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMAX}$ starting from ρ reaches in a finite time a configuration ρ' such that $IM_{m_{i+1}}(\rho')$ holds.*

Given a metric value $m_i \in M$, consider a configuration $\rho \in \mathcal{LC}_{m_i}$. We know by Lemma 7 that any execution starting from ρ reaches in a finite time a configuration ρ' such that $IM_{m_{i+1}}(\rho')$ holds. Denote by e an execution starting from ρ'. Now, we can observe that, if all processes of $P_{m_{i+1}}$ are not enabled in a configuration of e, then this configuration belongs to $\mathcal{LC}_{m_{i+1}}$. Then, we can prove that any process of $P_{m_{i+1}}$ takes only a finite number of steps in any execution starting from ρ'. This implies the following result:

Lemma 8. *For any $m_i \in M$ and for any configuration $\rho \in \mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMAX}$ starting from ρ reaches in a finite time a configuration of $\mathcal{LC}_{m_{i+1}}$.*

Let ρ be an arbitrary configuration. We know by Lemma 4 that any execution starting from ρ reaches in a finite time a configuration of $\mathcal{LC}_{mr} = \mathcal{LC}_{m_0}$. Then, we can apply at most k times the result of Lemma 8 to obtain that any execution starting from ρ reaches in a finite time a configuration of $\mathcal{LC}_{m_k} = \mathcal{LC}$, that proves the following result:

Lemma 9. *Starting from any configuration, any execution of $\mathcal{SSMAX}$ reaches a configuration of $\mathcal{LC}$ in a finite time.*

Lemmas 3 and 9 imply respectively the closure and the convergence of $\mathcal{SSMAX}$. We can summarize our results with the following theorem.

Theorem 3. *$\mathcal{SSMAX}$ is a $(S_B, n-1)$-TA strictly stabilizing protocol for spec.*

5 Conclusion

We introduced a new notion of Byzantine containment in self-stabilization: the topology-aware strict stabilization. This notion relaxes the constraint on the containment radius of the strict stabilization to a containment area. In other words, the set of correct processes which may be infinitely often disturbed by Byzantine processes is a function depending on the topology of the system and on the actual location of Byzantine processes. We illustrated the relevance of this notion by providing a topology-aware strictly stabilizing protocol for the maximum metric tree construction problem which does not admit strictly stabilizing solution. Moreover, our protocol performs the optimal containment area with respect to the topology-aware strict stabilization.

Our work raises some opening questions. Number of problems do not accept strictly stabilizing solution. Does any of them admit a topology-aware strictly stabilizing solution ? Is it possible to give a necessary and/or sufficient condition for a problem to admit a topology-aware strictly stabilizing solution ? What happens if we consider only bounded Byzantine behavior ?

References

1. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. ACM Commun. 17(11), 643–644 (1974)
2. Dolev, S.: Self-stabilization. MIT Press, Cambridge (March 2000)
3. Tixeuil, S.: Self-stabilizing Algorithms. Chapman & Hall/CRC Applied Algorithms and Data Structures. In: Algorithms and Theory of Computation Handbook, 2nd edn., pp. 26.1–26.45. CRC Press, Taylor & Francis Group (November 2009)
4. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
5. Nesterenko, M., Tixeuil, S.: Discovering network topology in the presence of byzantine nodes. IEEE Trans. Parallel Distrib. Syst. (October 2009)

6. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. J. ACM 51(5), 780–799 (2004)
7. Daliot, A., Dolev, D.: Self-stabilization of byzantine protocols. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 48–67. Springer, Heidelberg (2005)
8. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: 21st Symposium on Reliable Distributed Systems, p. 22. IEEE Computer Society, Los Alamitos (2002)
9. Ben-Or, M., Dolev, D., Hoch, E.N.: Fast self-stabilizing byzantine tolerant digital clock synchronization. In: Bazzi, R.A., Patt-Shamir, B. (eds.) PODC, pp. 385–394. ACM, New York (2008)
10. Dolev, D., Hoch, E.N.: On self-stabilizing synchronous actions despite byzantine attacks. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 193–207. Springer, Heidelberg (2007)
11. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 350–362. Springer, Heidelberg (2006)
12. Sakurai, Y., Ooshita, F., Masuzawa, T.: A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 283–298. Springer, Heidelberg (2005)
13. Masuzawa, T., Tixeuil, S.: Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. International Journal of Principles and Applications of Information Science and Technology (PAIST) 1(1), 1–13 (2007)
14. Dubois, S., Potop-Butucaru, M.G., Nesterenko, M., Tixeuil, S.: Self-stabilizing byzantine asynchronous unison. CoRR abs/0912.0134 (2009)
15. Gouda, M.G., Schneider, M.: Stabilization of maximal metric trees. In: Arora, A. (ed.) WSS, pp. 10–17. IEEE Computer Society, Los Alamitos (1999)
16. Gouda, M.G., Schneider, M.: Maximizable routing metrics. IEEE/ACM Trans. Netw. 11(4), 663–675 (2003)
17. Dubois, S., Masuzawa, T., Tixeuil, S.: The Impact of Topology on Byzantine Containment in Stabilization. Research report inria-00481836 (May 2010), http://hal.inria.fr/inria-00481836/en/

Minimum Dominating Set Approximation in Graphs of Bounded Arboricity

Christoph Lenzen and Roger Wattenhofer

Computer Engineering and Networks Laboratory (TIK)
ETH Zurich
{lenzen,wattenhofer}@tik.ee.ethz.ch

Abstract. Since in general it is NP-hard to solve the minimum dominating set problem even approximatively, a lot of work has been dedicated to central and distributed approximation algorithms on restricted graph classes. In this paper, we compromise between generality and efficiency by considering the problem on graphs of small arboricity a. These family includes, but is not limited to, graphs excluding fixed minors, such as planar graphs, graphs of (locally) bounded treewidth, or bounded genus. We give two viable distributed algorithms. Our first algorithm employs a forest decomposition, achieving a factor $\mathcal{O}(a^2)$ approximation in randomized time $\mathcal{O}(\log n)$. This algorithm can be transformed into a deterministic central routine computing a linear-time constant approximation on a graph of bounded arboricity, without a priori knowledge on a. The second algorithm exhibits an approximation ratio of $\mathcal{O}(a \log \Delta)$, where Δ is the maximum degree, but in turn is uniform and deterministic, and terminates after $\mathcal{O}(\log \Delta)$ rounds. A simple modification offers a trade-off between running time and approximation ratio, that is, for any parameter $\alpha \geq 2$, we can obtain an $\mathcal{O}(a\alpha \log_\alpha \Delta)$-approximation within $\mathcal{O}(\log_\alpha \Delta)$ rounds.

1 Introduction

We are interested in the distributed complexity of the minimum dominating set (MDS) problem, a classic both in graph theory and distributed computing. Given a graph, a dominating set is a subset D of nodes such that each node in the graph is either in D, or has a direct neighbor in D. There are various applications where it is beneficial to find dominating sets of small cardinality, for instance in routing. We want to find a minimum dominating set (MDS)—or a dominating set that is not much larger than an MDS—fast, if possible even in constant time.

Regrettably, it has been shown that in general graphs, small dominating sets cannot be computed in constant time [15]. In some special graphs, however, this is simple. In a tree, for instance, we can get a constant approximation of the MDS problem by choosing all inner nodes, because a third of the inner nodes must be in any MDS. In fact, one can approximate an MDS quickly and up to a constant for more general graphs, for instance planar graphs [18]. What

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 510–524, 2010.

about graph classes between planar and general? One property of planar graphs is *sparsity*, i.e., the number of edges is at most linear in the number of nodes. Hence it seems natural to raise the question on which side of the boundary between "easy" and "hard" sparse graphs reside.

Unfortunately, a constant MDS approximation cannot be computed quickly in sparse graphs, as the hardness of the general MDS problem can be enclosed into a small part of the graph whose contribution to the global solution is decisive. Just take $n - \sqrt{n}$ nodes, and connect them as a star. The remaining $\sqrt{n}$ nodes can be connected arbitrarily, yielding a sparse graph with less than $n+\sqrt{n}^2 = 2n$ edges. Whereas the star is dominated by its center alone, we can just apply the lower bound [15] to the remaining $\sqrt{n}$ nodes.

Consequently, we need a different definition of sparseness, one that applies "everywhere" in the graph. In this work we consider graphs of bounded *arboricity*, subsuming planar graphs, graphs of bounded genus or treewidth, and, more generally, graphs excluding any fixed minor. The arboricity of a graph can be defined in two equivalent ways: (*i*) the minimum number of forests into which the edge set can be partitioned and (*ii*) the maximum ratio of edges to nodes in any subgraph. We present two distributed algorithms, each of which exploits one of these properties. Generally speaking, both algorithms run in logarithmic time and feature an approximation guarantee that depends on the arboricity of the graph. This can be seen as the result of a balancing act between generality of feasible inputs on the one hand and approximation quality and running time on the other hand: On graphs of small arboricity, we outperform the so far best algorithm designed for general graphs; faster routines achieving similar approximations are currently known only for severely constrained inputs like planar graphs or graphs of bounded growth.

As a corollary, we observe that based on a forest decomposition, a central algorithm can compute a constant-factor approximation in a linear number of operations in any graph of bounded arboricity. In contrast, asymptotically optimum solutions are intractable in general graphs (and thus also on sparse graphs), where it is known to be NP-hard to obtain any sublogarithmic approximation ratio [23]. To the best of our knowledge, graphs of bounded arboricity represent the so far most general family of graphs for which a constant MDS approximation can be computed efficiently. Finally, given that all our algorithms are simple and the distributed routines require only small messages, we believe them to be suitable for use in practice.

2 Related Work

It is safe to say that computing small dominating sets is one of the most classical and fundamental graph problems. The task of finding a minimum dominating set was among the first to be recognized as NP-complete [11] and—dominating set being a quite general special case of set cover—Raz and Safra proved that it is NP-hard to achieve a $c \log \Delta$-approximation, where Δ is the maximum node degree and $c > 0$ a constant [23]. Ironically, a $((1 - o(1)) \log \Delta)$-approximation

can be obtained by a naive greedy algorithm, a strategy which was shown to be optimal for polynomial-time algorithms unless NP is contained in the class of problems that can be solved deterministically in time $n^{\mathcal{O}(\log\log n)}$ (where n is the number of nodes) by Feige [10].

Arguably, for distributed algorithms things are even worse. Even if message size is unbounded and nodes may perform arbitrary local computations (in particular solve NP-hard problems!), $\Omega(\log \Delta/\log\log \Delta)$ and $\Omega(\log n/\log\log n)$ communication rounds are required to guarantee a polylogarithmic approximation [15]. Currently, the best known randomized algorithm yields an expected[1] $\mathcal{O}(\log \Delta)$-approximation in $\mathcal{O}(\log n)$ time [17]. Considering that a better approximation ratio is (supposing P≠NP, of course) intractable, this is optimal up to a factor of $\mathcal{O}(\log n \log\log \Delta/\log \Delta)$ in time complexity. However, for this algorithm no non-trivial bound on the message size holds. The authors give a second algorithm whose messages are of size at most $\mathcal{O}(\log \Delta)$, trading in for a time complexity of $\mathcal{O}(\log^2 \Delta)$. To the best of our knowledge, the deterministic distributed complexity of MDS approximations on general graphs is more or less a blind spot, as so far neither fast (polylogarithmic time) algorithms nor stronger lower bounds are known.

In light of the strong negative results, it is natural to consider restricted families of graphs. Here we get a colorful picture. The $((1-o(1))\log \Delta)$-hardness result of Feige has been extended to bipartite and split graphs by Chlebík and Chlebíková [4], and thus also to chordal graphs and their complements. The case of trees is trivial for centralized algorithms. Distributed algorithms need time in the order of the depth of the tree for an optimal solution, while a constant approximation is also trivial. For series-parallel graphs a linear-time centralized algorithm was devised by Takamizawa et al. [25]. This algorithm generalizes to a polynomial-time one in graphs of bounded treewidth.

More interesting are less extreme cases. For instance, both in planar and unit disk graphs, it remains NP-complete to solve MDS exactly [5,11], but PTAS have been given [2,13]. These graph classes are also comparatively well understood in the distributed setting. On unit disk graphs, or more generally the family of graphs of bounded growth,[2] a constant approximation can be found in $\mathcal{O}(\log^* n)$ rounds [24]. The respective algorithm outputs a maximal independent set (which must also be a dominating set); because the number of independent neighbors of any node in an optimal solution is bounded by a constant, so is the approximation ratio of the algorithm. Using the same argument, a constant approximation can be found on graphs of bounded independence, however, the so far best known maximal independent set algorithms on this graph class have randomized running time $\mathcal{O}(\log n)$ with high probability [1,21]. For deterministic algorithms on unit disk graphs, the product of approximation ratio and running time cannot be in $o(\log^* n)$ [19], implying that the upper bound from [24] is

[1] This can be improved to hold with high probability [14].

[2] "Bounded independence" means that the number of independent nodes in neighborhoods is constant. "Bounded growth" refers to the stronger property that the number of independent nodes in an r-neighborhood is polynomially bounded in r.

asympotically optimal. The same result also implies a constant lower bound on the approximation ratio of deterministic distributed algorithms running on planar graphs in $o(\log^* n)$ rounds, which was proved independently in [6]. On the other hand, a deterministic constant-time algorithm that outputs a 74-approximation on planar graphs was given in [18]. Shortly thereafter, Czygrinow et al. devised a $(1+\varepsilon)$-approximation on planar graphs [6] (for any constant $\varepsilon > 0$) in $\mathcal{O}(\log^* n)$ time. Their algorithm utilizes a different constant-time approximation for planar graphs with a constant, but much larger approximation ratio than in [18]; intriguingly, a closer look reveals that this routine does not require planarity at all, but works for any graph free of $K_{3,3}$-minors. Earlier, two of the authors of this work gave slower (polylogarithmic time with large exponent) algorithms yielding $(1+o(1))$-approximations in graphs excluding any fixed minor [7,8]. It should be mentioned, though, that the algorithms from [7,8,6] have in common that they compute optimal solutions to subproblems on subgraphs of bounded diameter, implying that NP-hard problems are solved. Therefore, the respective results are mainly theoretic statements on distributed time complexity and probably infeasible in practice.

Returning to centralized algorithms, Baker's PTAS for planar graphs [2] is based on the fact that planar graphs admit an $\mathcal{O}(D)$ tree decomposition in time $\mathcal{O}(Dn)$, where D denotes the diameter of the graph. Building on Baker's ideas, Eppstein extended the approach to minor-closed families excluding a specific apex graph [9]. Finally Grohe generalized the technique further, giving PTAS for any minor-closed family that does not contain all minors [12].

3 Contribution

In this work, we give practical approximation algorithms whose approximation ratios depend on the arboricity of the underlying graph, i.e., the minimum number of forests into which the edge set can be decomposed. The family of graphs of bounded arboricity contains all graphs excluding fixed minors, thus including planar graphs, graphs of bounded genus, and graphs of bounded tree-width. It is a proper superset of the family of graphs excluding some minor, as already graphs of arboricity 2 may have $K_{\sqrt{n}}$-minors.[3]

Our first distributed algorithm computes, given a forest decomposition into f forests, an f^2-approximation in randomized time $\mathcal{O}(\log n)$ with high probability. The algorithms of Czygrinow et al. [6,7,8] also involve forest decompositions, but they rely on additional properties of the underlying graph and are considerably slower. Barenboim and Elkin applied essentially the same technique as Czygrinow et al. to obtain forest decompositions [3]. However, they stated slightly different and more general results, better fitting our needs. More precisely, their

[3] In contrast, graphs of bounded independence are fundamentally different. A clique has maximum arboricity, but independent sets are of size one, while a star has arboricity one, but the neighborhood of the center consists of $n-1$ independent nodes. The combination of bounded independence *and* arboricity implies bounded degree and vice versa.

algorithms compute $\Theta(a)$-forest decompositions of graphs of arboricity a in time $\mathcal{O}(\log n)$ provided that an upper bound in $\mathcal{O}(a)$ on a or an upper bound in $\mathcal{O}(n^c)$ on n (for a constant $c \geq 1$) is known to the nodes. Employing their algorithm, we thus get an $\mathcal{O}(a^2)$-approximation in $\mathcal{O}(\log n)$ time on any graph of arboricity a. In particular, the resulting routine guarantees constant approximation ratios on graphs of bounded arboricity, using messages of size $\mathcal{O}(\log n)$. From a coloring lower bound by Linial [20], Barenboim and Elkin inferred a lower bound of $\Omega(\log n / \log f)$ on the time to compute a forest decomposition into f forests distributedly. Thus, no algorithm utilizing a forest decomposition can yield substantially better results. Using standard techniques, our algorithm can also be transformed into an efficient central routine. In this case, we can remove the logarithmic overhead in running time and the need for randomization, and we need no a-priori knowledge on the arboricity of the graph. Hence, we achieve a uniform, central $\mathcal{O}(a)$-approximation within $\mathcal{O}(an)$ steps, i.e., a linear time constant approximation on graphs of bounded arboricity.

We proceed by presenting a second, deterministic distributed algorithm that features a smaller running time of $\mathcal{O}(\log \Delta)$, smaller messages of size $\mathcal{O}(\log \log \Delta)$, and is uniform, i.e., does not require any bounds on a or n as input. Interestingly, this algorithm requires less symmetry breaking than the first one (which apart from the forest decomposition computes a maximal independent set). Indeed, a port numbering suffices, and if (an upper bound on) a is known to the algorithm, also this condition can be dropped, i.e., the modified algorithm does not rely on any non-topological symmetry breaking information at all. These advantages come not for free, as the approximation ratio of the second algorithm now depends on the maximum degree, being $\mathcal{O}(a \log \Delta)$. A simple modification of the algorithm permits to reduce the running time to $\mathcal{O}(\log \Delta / \log \alpha)$, but at the expense weakening the approximation guarantee to $\mathcal{O}(a\alpha \log \Delta / \log \alpha)$ (for any $\alpha \geq 2$). We give an example where the approximation ratio of the second algorithm is matched, i.e., better approximation guarantees require different techniques.

4 Constant-Factor Approximation

In this section, we present an algorithm that computes a dominating set at most $\mathcal{O}(a^2)$ larger than optimum. After introducing the employed standard model of distributed computation and some preliminary definitions, we proceed by presenting the algorithm. After proving its approximation ratio, we review how to obtain a $(2a)$-forest decomposition of a graph of arboricity a in $\mathcal{O}(an)$ central operations. From these results, we conclude that on graphs of bounded arboricity constant MDS approximations can be computed in $\mathcal{O}(\log n)$ distributed rounds and $\mathcal{O}(n)$ central steps.

4.1 Model

We model a distributed system as a simple, undirected graph $G = (V, E)$, where edges represent bidirectional communication links. Algorithms proceed in

synchronous rounds. In each round, nodes (*i*) send messages to their neighbors, (*ii*) receive messages sent by their neighbors, and (*iii*) perform arbitrary (but finite) local computations. Furthermore, we permit randomization, i.e., nodes have access to an unlimited source of unbiased random bits. Initially, any node v knows its *inclusive neighborhood* $\mathcal{N}_v^+ := \{v\} \cup \{w \in V \mid \{v, w\} \in E\}$. Similarly, by $\mathcal{N}_S^+ := \bigcup_{s \in S} \mathcal{N}_s^+$ we denote the inclusive neighborhood of a set $S \subseteq V$.

Before we describe our first algorithm, we need to formalize some well-known graph theoretic concepts.

Definition 1 (Dominating Sets). *A* dominating set *is a subset of the nodes $D \subseteq V$ such that $\mathcal{N}_D^+ = V$. A* minimum dominating set (MDS) *is a dominating set of minimum cardinality. An α-approximation to an MDS is a dominating set of size at most $\alpha|M|$, where M is an MDS.*

Definition 2 (Independent Sets). *An* independent set *is a subset of the nodes $I \subseteq V$ containing no neighbors. A* maximal independent set (MIS) *is an independent set for which adding any node destroys independence, i.e., $\mathcal{N}_I^+ = V$ and I is also a dominating set.*

Definition 3 (Forests and Forest Decompositions). *A* forest *is a graph containing no cycles. An* oriented forest *is a forest where edges have been oriented such that outdegrees are at most one. If $(v, w) \in E$ for two nodes in the forest, w is called* parent *of v and v is a* child *of w. An f-*forest decomposition *of a graph G is a decomposition of the edge set $E = E_1 \dot\cup E_2 \dot\cup \ldots \dot\cup E_f$ together with an appropriate orientation of the edges such that the subgraphs induced by each E_i, $i \in \{1, \ldots, f\}$, are oriented forests. Given a forest decomposition, we denote by $P(v) := \{w \in V \mid (v, w) \in E\}$ and $P(S) := \bigcup_{s \in S} P(s)$ the set of parents of the node $v \in V$ and the set $S \subseteq V$, respectively. The* arboricity *$a(G)$ of G is the minimal number of forests in a forest decomposition of G. In the forthcoming, we will write a instead of $a(G)$ whenever the argument is clear from the context.*

4.2 Algorithm

Our first algorithm is based on the following observations. Given an f-forest decomposition and an MDS M, the nodes can be partitioned into two sets. One set contains the nodes which are dominated by a parent, the other contains the remaining nodes, which thus are either themselves in M or have a child in M. Since dominating set nodes can cover only f parents, the latter are at most $(f + 1)|M|$ many nodes. If each such node elects all its parents into the dominating set, we have chosen at most $f(f + 1)|M|$ nodes.

For the first set, we can exploit the fact that each node has at most f parents in a more subtle manner. Covering the nodes in this set by parents only, we need to solve a special case of set cover where each element is part of at most f sets. Such instances can be approximated well by a simple sequential greedy algorithm: Pick any element that is not yet covered and add *all* sets containing it; repeat this until no element remains. Since in each step we add at least one new set from an optimum solution, we get an f-approximation. This strategy

can be parallelized by computing a maximal independent set in the graph where two nodes are adjacent exactly if they share a parent, as adding the parents of the nodes in an independent set in any order would be a feasible execution of the sequential greedy algorithm.

Putting these two observations together, first all parents of nodes from a maximal independent set in a helper graph are elected into the dominating set. In this helper graph, two nodes are adjacent if they share a parent. Afterwards, the remaining uncovered nodes have no parents, therefore it is uncritical to select them all. This approach is summarized in Algorithm 1.

Algorithm 1. Parent Dominating Set

input : f-forest decomposition of G
output: dominating set D
1 $H := (V, F)$, where $\{v, w\} \in F \Leftrightarrow P(v) \cap P(w) \neq \emptyset$ // neighbors in H share a parent in G
2 Compute a maximal independent set I on H
3 $D := P(I)$ // all parents of nodes in the independent set join D
4 $D := D \cup (V \setminus \mathcal{N}_D^+)$ // all still uncovered nodes may safely join D

4.3 Analysis

Lemma 1. *In Line 1 of Algorithm 1, at most $f(f+2)|M|$ many nodes enter D, where M denotes an MDS of G.*

Proof. Denote by $V_c \subseteq V$ the set of nodes that have a child in M or are themselves in M. We have that $|V_c| \leq (f+1)|M|$, since no node has more than f parents. Each such node adds at most f parents to D in Line 1 of the algorithm, i.e., in total at most $f(f+1)|M|$ many nodes join D because they are elected by children in $I \cap V_c$.

Now consider the set of nodes $V_p \subset V$ that have at least one parent in M, i.e., in particular the nodes in $I \cap V_p$ have at least one parent in M. By the definition of F and the fact that I is an independent set, no node in M can have two children in I. Thus, $|I \cap V_p| \leq |M|$. Since no node has more than f parents, we conclude that at most $f|M|$ many nodes join $|D|$ after being marked as candidate by a child in $I \cap V_p$.

Finally, observe that since M is a dominating set, we have that $V_c \cup V_p = V$ and thus

$$|D| \leq f|I \cap V_c| + f|I \cap V_p| \leq f(f+1)|M| + f|M| = f(f+2)|M| \ ,$$

concluding the proof. □

Theorem 1. *Algorithm 1 outputs a dominating set D containing at most $(f^2 + 3f + 1)|M|$ many nodes, where M is an optimum solution.*

Proof. By Lemma 1, at most $f(f+2)|M|$ nodes enter D in Line 1 of the algorithm. Since I is a MIS in H, all nodes that have a parent are adjacent to at least one node in D after Line 1. Hence, the nodes selected in Line 1 must either be covered by a child or themselves be in M. As no node has more than f parents, thus in Line 1 at most $(f+1)|M|$ many nodes join D. Altogether, at most $(f^2+3f+1)|M|$ many nodes may end up in D as claimed. □

Corollary 1. *In any graph G, a factor $\mathcal{O}(a(G)^2)$ approximation to the MDS problem can be computed distributedly in $\mathcal{O}(\log n)$ rounds with high probability.*[4] *In particular, on graphs of bounded arboricity a constant-factor approximation can be obtained in $\mathcal{O}(\log n)$ rounds with high probability. This can be accomplished with messages of size $\mathcal{O}(\log n)$.*

Proof. We run Algorithm 1 in a distributed fashion. To see that this is possible, observe that (*i*) nodes need only to know whether a neighbor is a parent or a child, (*ii*) that H can be constructed locally in 2 rounds and (*iii*) a synchronous round in H can be simulated by two rounds on G. Thus, we simply may pick distributed algorithms to compute a forest decomposition of G and a maximal independent set and plug them together to obtain a distributed variant of Algorithm 1.

For the forest decomposition, we employ the algorithm from [3], yielding a decomposition into $\mathcal{O}(a)$ forests in $\mathcal{O}(\log n)$ rounds. A maximal independent set can be computed in $\mathcal{O}(\log n)$ rounds with high probability by well-known algorithms [1,21], or a more recent similar technique [22]. In total the algorithm requires $\mathcal{O}(\log n)$ rounds with high probability and according to Theorem 1 the approximation guarantee is $\mathcal{O}(a)$.

Regarding message size, we need to check that we do not require large messages because we compute a MIS on H. Formulated abstractly, the algorithm from [22] breaks symmetry by making each node still eligible for the independent set choosing a random value in each round and permitting it to join the independent set if its value is a local minimum. This concept can for instance be realized by taking $\mathcal{O}(\log n)$ random bits as encoding of some number and comparing it to neighbors. The respective values will with high probability differ. This approach can be emulated using messages of size $\mathcal{O}(\log n)$ in G: Nodes send their random values to all parents in the forest decomposition, which then forward only the smallest values to all children.[5] □

4.4 Linear Time Central Algorithm

Employing well-known techniques, a central algorithm can compute a suitable forest decomposition with linear complexity.

[4] I.e., with probability at least $1 - 1/n^c$ for an arbitrary, but fixed constant $c > 0$.

[5] If (an upper bound on) n is not known, one can start with constantly many bits and double the number of used bits in each round where two nodes pick the same value. This will not slow down the algorithm significantly and bound message size by $\mathcal{O}(\log n)$ with high probability.

Lemma 2. *A $2a(G)$-forest decomposition of G can be computed in $\mathcal{O}(|E|+n) \subseteq \mathcal{O}(an)$ computational steps.*

Proof. For each node, we compute and store its degree ($\mathcal{O}(|E|)$ steps). Now we place the nodes into buckets according to their degree. We pick a node with smallest degree, we orient its edges, delete them, and update the assignment of the nodes to the buckets. This is repeated until no more nodes remain. Assuming appropriate data structures, the number of operations will be bounded by $\mathcal{O}(|E| + n) \subseteq \mathcal{O}(an)$, as each edge and node is accessed a constant number of times. Since a graph of arboricity a and n' nodes has less than an' edges, the smallest degree of any subgraph of G is at most $2a(G)$. Hence we obtain a forest decomposition into $2a(G)$ forests. □

Hence, for any graph of arboricity $a \in \mathcal{O}(1)$, a deterministic, central algorithm can compute an $\mathcal{O}(1)$-approximation to the MDS problem with linear complexity.

Corollary 2. *Deterministically, an $\mathcal{O}(a)$-approximation to an MDS can be computed in $\mathcal{O}(|E| + n) \subseteq \mathcal{O}(an)$ central steps.*

Proof. Using Lemma 2, we can compute a forest decomposition within the stated complexity bounds. In a central setting, Algorithm 1 can easily be implemented using $\mathcal{O}(|E| + n)$ steps. The approximation guarantee follows from Theorem 1. □

5 A Solution in the Port Numbering Model

Algorithm 1 might be unsatisfactory with regard to several aspects. Its running time is logarithmic in n even if the maximum degree Δ is small. This cannot be improved upon by any approach that utilizes a forest decomposition, as a lower bound of $\Omega(\log n / \log f)$ is known on the time to compute a forest decomposition into f forests [3]. The algorithm is not uniform, as it necessitates global knowledge of a bound on $a(G)$ or n.

Moreover, the algorithm requires randomization in order to compute a MIS quickly. Considering deterministic algorithms, one might pose the question how much initial symmetry breaking information needs to be provided to the nodes. While randomized algorithms may randomly generate unique identifiers of size $\mathcal{O}(\log n)$ in constant time with high probability, many deterministic algorithms assume them to be given as input. Milder assumptions are the ability to distinguish neighbors by means of a port numbering and/or an initial orientation of the edges.

In this section, we show that an uniform, deterministic algorithm exists that requires a port numbering only, yet achieves a running time of $\mathcal{O}(\log \Delta)$ and a good approximation ratio. The size of the computed dominating set is bounded linearly in the product of the arboricity $a(G)$ of the graph and the logarithm of the maximum degree Δ. Interestingly, we will observe later that if (an upper bound on) the arboricity is known to the algorithm, one can even drop the assumption of a port numbering.

5.1 The Port Numbering Model

In this section, we consider the so-called *port numbering model*. Again an undirected and simple graph $G = (V, E)$ is given. Communication is synchronous and computation is deterministic. Nodes refer to their neighbors by means of a port numbering, i.e., each node v uniquely maps the numbers $\{1, \ldots, \delta\}$ to its edges (where δ is the degree of v). Nodes can distinguish from which of their neighbors they received a specific message. However, in contrast to the previous setting where nodes had unique identifiers, different nodes now may refer to the same destination by different port numbers.

Note that this model is quite harsh. For instance, it is impossible to reliably discover cycles, compute a MIS, or determine the diameter of the graph.

5.2 Algorithm

The basic idea of Algorithm Greedy-by-Degree (Algorithm 2) is that it is always feasible to choose nodes of high *residual degree* (by which we mean the number of uncovered nodes in the inclusive neighborhood) simultaneously, i.e., all the nodes that cover up to a constant factor as many nodes as the one covering the most uncovered nodes. This permits to obtain strong approximation guarantees without the structural information provided by knowledge of $a(G)$ or a forest decomposition; the mere fact that the graph must be "locally sparse" enforces that if many nodes are elected into the set, also the dominating set must be large. A difficulty arising from this approach is that nodes are not aware of the current maximum residual degree in the graph. Hence, every node checks whether there is a node in its 2-hop neighborhood having a residual degree larger by a factor 2. If not, nodes may join the dominating set (even if their degree is not large from a global perspective), implying that the maximum residual degree drops by a factor of 2 in a constant number of rounds.

A second problem occurs once residual degrees become small. In fact, it may happen that a huge number of already covered nodes can each dominate the same small set of $a(G) - 1$ nodes. For this reason, it is mandatory to ensure that not more nodes may join the dominating set than actually need to be covered. To this end, nodes that still need to be covered elect one of their neighbors (if any) that are feasible according to the criterion of (locally) large residual degree. This scheme is described in Algorithm 2.

Note that nodes may never leave D once they entered it. Thus, nodes may terminate based on local knowledge only when executing the algorithm, as they can cease executing the algorithm as soon as $\delta_v = 0$, i.e., their complete neighborhood is covered by D. Moreover, it can easily be verified that one iteration of the loop can be executed by a local algorithm in the port numbering model using 6 rounds.

5.3 Analysis

In the sequel, when we talk of a *phase* of Algorithm 2, we refer to a complete execution of the while loop. We start by proving that not too many nodes with small residual degrees enter D.

Algorithm 2. Greedy-by-Degree.

```
   output: dominating set D
1  D := ∅
2  while V ≠ N_D^+ do
3      C := ∅ // candidate set
4      for v ∈ V in parallel do
5          δ_v := |N_v^+ \ N_D^+| // residual degree
6          Δ_v := max_{w∈N_v^+} {δ_w} // maximum residual degree within one hop
7          Δ_v := max_{w∈N_v^+} {Δ_w} // maximum residual degree within two hops
8          if ⌈log δ_v⌉ ≥ ⌈log Δ_v⌉ then
9              C := C ∪ {v}
10         end
11         if v ∈ N_C^+ \ N_D^+ then
12             w := any node from C ∩ N_v^+ (break symmetry by port numbers)
13             D := D ∪ {w} // uncovered nodes select a candidate joining D
14         end
15     end
16 end
```

Lemma 3. *Denote by M an MDS. During the execution of Algorithm 2, in total at most $16a|M|$ nodes join D in Line 2 of the algorithm after computing $\delta_v \leq 8a$ in Line 2 of the same phase.*

Proof. Fix a phase of the algorithm. Consider the set S consisting of all nodes $v \in V$ that become covered in this phase by some node $w \in \mathcal{N}_v^+$ that computes $\delta_w \leq 8a$ and joins D. As according to Line 2 nodes join D subject to the condition that residual degrees throughout their 2-hop neighborhoods are less than twice as large as their own, no node $m \in M$ can cover more than $16a$ many nodes in S. Hence, $|S| \leq 16a|M|$. Because of the rule that a node needs to be elected by a covered node in order to enter D, this is also a bound on the number of nodes joining D in a phase when they have residual degree at most $8a$. □

Next, we show that in each phase, at most a constant factor more nodes of large residual degree are chosen than are in an MDS.

Lemma 4. *If M is an MDS, in each phase of Algorithm 2 at most $16a|M|$ nodes v that compute $\delta_v > 8a$ in Line 2 join D in Line 2.*

Proof. Denote by D' the nodes $v \in V$ joining D in Line 2 of a phase in which they computed $\delta_v > 8a$ and by V' the set of nodes that had not been covered at the beginning of this phase. Define for $i \in \{0, \dots, \lceil \log n \rceil\}$ that

$$M_i := \{v \in M \mid \delta_v \in (2^{i-1}, 2^i]\}$$
$$V_i := \left\{ v \in V' \;\middle|\; \max_{w \in \mathcal{N}_v^+} \{\delta_w\} \in (2^{i-1}, 2^i] \right\}$$
$$D_i := \{v \in D' \mid \delta_v \in (2^{i-1}, 2^i]\} \ .$$

Note that $\bigcup_{i=\lceil \log 8a \rceil}^{\lceil \log n \rceil} D_i = D'$.

Consider any $j \in \{\lceil \log 8a \rceil, \ldots, \lceil \log n \rceil\}$. By definition, nodes in V_j may only be covered by nodes from M_i for $i \leq j$. Thus, $\sum_{i=0}^{j} 2^i |M_i| \geq |V_j|$.

Nodes $v \in D_j$ cover at least $2^{j-1}+1$ nodes from the set $\bigcup_{i \in \{j,\ldots,\lceil \log n \rceil\}} V_i$, as by definition they have no neighbors in V_i for $i < j$. On the other hand, Lines 2 to 2 of the algorithm impose that these nodes must not have any neighbors of residual degree larger than $2^{\lceil \log \delta_v \rceil} = 2^j$, i.e., these nodes cannot be in a set V_i for $i > j$. Hence, each node $v \in D_j$ has at least $2^{j-1}+1$ neighbors in V_j. This observation implies that the subgraph induced by $D_j \cup V_j$ has at least $2^{j-2}|D_j| \geq 2a|D_j|$ edges. On the other hand, by definition of the arboricity, this subgraph has less than $a(|D_j| + |V_j|)$ edges. It follows that

$$|D_j| \leq \frac{a|V_j|}{2^{j-2} - a} \leq 2^{3-j} a |V_j| \leq 2^{3-j} a \sum_{i=0}^{j} 2^i |M_i| \ .$$

We conclude that

$$\begin{aligned} \sum_{j=\lceil \log 8a \rceil}^{\lceil \log n \rceil} |D_j| &\leq \sum_{j=\lceil \log 8a \rceil}^{\lceil \log n \rceil} 2^{3-j} a \sum_{i=0}^{j} 2^i |M_i| \leq 8a \sum_{j=0}^{\lceil \log n \rceil} \sum_{i=0}^{j} 2^{i-j} |M_i| \\ &< 8a \sum_{i=0}^{\lceil \log n \rceil} \sum_{j=i}^{\infty} 2^{i-j} |M_i| = 16a \sum_{i=0}^{\lceil \log n \rceil} |M_i| = 16a|M| \ , \end{aligned}$$

as claimed. □

We now can bound the approximation quality of the algorithm.

Theorem 2. *Assume that G has maximum degree Δ. Then an execution of Algorithm 2 on G terminates within $6\lceil \log(\Delta+1) \rceil$ rounds and outputs a dominating set at most a factor $16a(G) \log \Delta$ larger than optimum. The worst-case approximation ratio of the algorithm is $\Theta(a(G) \log \Delta)$. Message size can be bounded by $\mathcal{O}(\log \log \Delta)$.*

Proof. We first examine the running time of the algorithm. Denote by $\Delta(i)$ the maximum residual degree after the i^{th} phase, i.e., $\Delta(0) = \Delta + 1$ (as a node also covers itself). As observed earlier, each phase of Algorithm 2 takes six rounds. Because all nodes v computing a δ_v satisfying $\lceil \log \delta_v \rceil = \lceil \log \Delta(i) \rceil$ join C in phase i and any node in $\mathcal{N}_C^+$ becomes covered, we have that $\lceil \log \Delta(i+1) \rceil \leq \lceil \log \Delta(i) \rceil - 1$ for all phases i. Since the algorithm terminates at the end of the subsequent phase once $\Delta(i) \leq 2$, in total at most $\lceil \log \Delta(0) \rceil = \lceil \log(\Delta+1) \rceil$ phases are required.

Having established the bound on the running time of the algorithm, its approximation ratio directly follows[6] by applying Lemmas 3 and 4. The bound on the message size follows from the observation that in each phase nodes need to exchange residual degrees rounded to powers of 2 and a constant number of binary values only.

[6] Note that in the last three phases the maximum degree is at most $8 \leq 8a$.

For the lower bound of $\Omega(a \log \Delta)$, we briefly sketch appropriate input graphs. We start with $a(G)$ being constant. For $k \in \mathbb{N}$, $k \geq 2$ an optimal solution consists of 4 nodes of degree 2^{k-1}. Half of their inclusive neighborhood is covered by a node having degree 2^k, another quarter by a node of degree 2^{k-1}, one eighth by a node of degree 2^{k-2}, and so on. In each phase, the algorithm will choose one of the latter nodes, namely the one of highest degree. Thus it will choose $k \in \Theta(\log \Delta)$ nodes, whereas an optimal solution contains $4 \in \mathcal{O}(1)$ nodes. This bound extends to arbitrarily large values of n by replicating this graph sufficiently often. The described graph has constant arboricity, as each node is covered at most constantly often. Similarly, we could decide to cover each node multiple times by nodes in the suboptimal solution computed by the algorithm (i.e., half of the neighborhood of a node in the optimal solution is covered by $\Theta(a)$ nodes, a quarter by another $\Theta(a)$ nodes, etc.). Hence, for any value of a and arbitrarily large n, we can construct a graph of n nodes where Algorithm 2 computes a solution by factor $\Omega(a \log \Delta)$ worse than the optimum. □

Like with the algorithms by Kuhn et al. [16,17], we can sacrifice accuracy in order to speed up the computation.

Corollary 3. *For any $\alpha \geq 2$, Algorithm 2 can be modified such that it has a running time of $\mathcal{O}(\log_\alpha \Delta)$ and approximation ratio $\mathcal{O}(a(G)\alpha \log_\alpha \Delta)$.*

Proof. We simply change the base of the logarithms in Line 2 of the algorithm, i.e., instead of rounding residual degrees to integer powers of two, we round to integer powers of α. Naturally, this linearly affects the approximation guarantees. In the proof of Lemma 4, we just replace the respective powers of 2 by α as well, yielding a bound of $\mathcal{O}(a(G) + \alpha \log_\alpha \Delta)$ on the approximation ratio by the same reasoning as in Theorem 2. □

If it was not for the computation of a MIS, we could speed up Algorithm 1 in almost the same manner (accepting a forest decomposition into a larger number of forests). However, the constructed helper graph is of bounded independence, but not arboricity or growth. For this graph class currently no distributed algorithm computing a MIS in time $o(\log n)$ is known.

We conclude this section with some final remarks. If nodes know $a(G)$ (or a reasonable upper bound), a port numbering is not required anymore. In this case, nodes will join D without the necessity of being elected by a neighbor, however only if the prerequisite $\delta_v > 8a$ is satisfied. To complete the dominating set, uncovered nodes may join D independently of δ_v once their neighborhood contains no more nodes of residual degree larger than $8a$. It is not hard to see that with this modification, essentially the same analysis as for Algorithm 2 applies, both with regard to time complexity and approximation ratio.

6 Conclusion

We presented efficient distributed minimum dominating set approximation algorithms for graphs of bounded arboricity. Compared to the best known solution for general graphs using reasonably sized messages, we can either (i) reduce the

running time by a factor of $\Theta(\log \Delta)$ (Algorithm 2) or (ii) change the running time from $\Theta(\log^2 \Delta)$ to $\Theta(\log n)$ and improve on the approximation quality by a factor of $\Theta(\log \Delta)$ (Algorithm 1). Moreover, our algorithms provide deterministic approximation guarantees. The Algorithms by Kuhn et al. [16] utilize randomized rounding, thus yielding probabilistic bounds. The faster of the two given algorithms is entirely deterministic. In contrast, Algorithm 1 necessitates the computation of a maximal independent set on a graph of bounded independence; deterministic solutions of running time linear in Δ are known, but a respective variant of the algorithm would be slow in comparison.

Of independent interest might be that on graphs of bounded arboricity, a central version of Algorithm 1 obtains a constant-factor approximation within a linear number of operations, without the need for randomization. In summary, minimum dominating sets are substantially easier to approximate on graphs of small arboricity, for distributed as well as for central algorithms.

Acknowledgements

We would like to thank Jukka Suomela for inspiring discussions and Topi Musto and the anonymous reviewers for many valuable comments that helped to improve this paper.

References

1. Alon, N., Babai, L., Itai, A.: A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. J. Algorithms 7(4), 567–583 (1986)
2. Baker, B.S.: Approximation Algorithms for NP-Complete Problems on Planar Graphs. J. ACM 41(1), 153–180 (1994)
3. Barenboim, L., Elkin, M.: Sublogarithmic Distributed MIS algorithm for Sparse Graphs using Nash-Williams Decomposition. Distributed Computing, 1–17 (2009)
4. Chlebk, M., Chlebkov, J.: Approximation Hardness of Dominating Set Problems in Bounded Degree Graphs. Information and Computation 206(11), 1264–1275 (2008)
5. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit Disk Graphs. Discrete Math. 86(1-3), 165–177 (1990)
6. Czygrinow, A., Hańćkowiak, M., Wawrzyniak, W.: Fast Distributed Approximations in Planar Graphs. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 78–92. Springer, Heidelberg (2008)
7. Czygrinow, A., Hańćkowiak, M.: Distributed Almost Exact Approximations for Minor-Closed Families. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 244–255. Springer, Heidelberg (2006)
8. Czygrinow, A., Hanckowiak, M.: Distributed Approximation Algorithms for Weighted Problems in Minor-Closed Families. In: Lin, G. (ed.) COCOON 2007. LNCS, vol. 4598, pp. 515–525. Springer, Heidelberg (2007)
9. Eppstein, D.: Diameter and Treewidth in Minor-Closed Graph Families. Algorithmica 27(3), 275–291 (2000)
10. Feige, U.: A Threshold of $ln n$ for Approximating Set Cover. J. ACM 45(4), 634–652 (1998)

11. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
12. Grohe, M.: Local Tree-Width, Excluded Minors, and Approximation Algorithms. Combinatorica 23(4), 613–632 (2003)
13. Hunt, H.B., Marathe, M.V., Radhakrishnan, V., Ravi, S.S., Rosenkrantz, D.J., Stearns, R.E.: NC-Approximation Schemes for NP- and PSPACE-Hard Problems for Geometric Graphs. Journal of Algorithms 26(2), 238–274 (1998)
14. Kuhn, F.: Personal Communication (2010)
15. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What Cannot Be Computed Locally! In: Proc. 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC (2004)
16. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The Price of Being Near-Sighted. In: Proc. 17th ACM-SIAM Symposium on Discrete Algorithms, SODA (2006)
17. Kuhn, F., Wattenhofer, R.: Constant-Time Distributed Dominating Set Approximation. Distrib. Comput. 17(4), 303–310 (2005)
18. Lenzen, C., Oswald, Y.A., Wattenhofer, R.: What can be Approximated Locally? In: 20th ACM Symposium on Parallelism in Algorithms and Architecture, SPAA (June 2008)
19. Lenzen, C., Wattenhofer, R.: Leveraging Linial's Locality Limit. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 394–407. Springer, Heidelberg (2008)
20. Linial, N.: Locality in Distributed Graph Algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
21. Luby, M.: A Simple Parallel Algorithm for the Maximal Independent Set Problem. SIAM J. Comput. 15(4), 1036–1055 (1986)
22. Métivier, Y., Robson, J.M., Saheb Djahromi, N.,, Z.: An Optimal Bit Complexity Randomised Distributed MIS Algorithm. In: Kutten, S., Žerovnik, J. (eds.) SIROCCO 2009. LNCS, vol. 5869, pp. 1–15. Springer, Heidelberg (2010)
23. Raz, R., Safra, S.: A Sub-Constant Error-Probability Low-Degree Test, and a Sub-Constant Error-Probability PCP Characterization of NP. In: Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC), pp. 475–484. ACM, New York (1997)
24. Schneider, J., Wattenhofer, R.: A Log-Star Distributed Maximal Independent Set Algorithm for Growth-Bounded Graphs. In: Proc. of the 27th Annual ACM Symposium on Principles of Distributed Computing, PODC (August 2008)
25. Takamizawa, K., Nishizeki, T., Saito, N.: Linear-Time Computability of Combinatorial Problems on Series-Parallel Graphs. J. ACM 29(3), 623–641 (1982)

Brief Announcement: Sharing Memory in a Self-stabilizing Manner*

Noga Alon[1], Hagit Attiya[2], Shlomi Dolev[3], Swan Dubois[4], Maria Gradinariu[4], and Sébastien Tixeuil[4]

[1] Sackler School of Mathematics and Blavatnik School of Computer Science, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv, 69978, Israel
nogaa@tau.ac.il

[2] Department of Computer Science, Technion, 32000, Israel
hagit@cs.technion.ac.il

[3] Contact author. Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il

[4] LIP6, Universite Pierre et Marie Curie, Paris 6/INRIA, 7606, France

Introduction. A core abstraction for many distributed algorithms simulates shared memory [3]; this abstraction allows to take algorithms designed for shared memory, and port them to asynchronous message-passing systems, even in the presence of failures. There has been significant work on creating such simulations, under various types of permanent failures, as well as on exploiting this abstraction in order to derive algorithms for message-passing systems (see [2].)

All these works, however, only consider permanent failures, neglecting to incorporate mechanisms for handling *transient* failures. Such failures may result from incorrect initialization of the system, or from temporary violations of the assumptions made by the system designer, for example the assumption that a corrupted message is always identified by an error detection code. The ability to automatically resume normal operation following transient failures, namely to be *self-stabilizing* [4], is an essential property that should be integrated into the design and implementation of systems.

This paper presents the first practically self-stabilizing simulation of shared memory that tolerates crashes. Specifically, we simulate a single-writer multi-reader (SWMR) *atomic* register in asynchronous message-passing systems where less than a majority of processors may crash. The simulation is based on reads and writes to a (majority) quorum in a system with a fully connected graph

* More details can be found in [1]. Noga Alon is supported in part by an ERC advanced grant, by a USA-Israeli BSF grant, by the Israel Science Foundation. Hagit Attiya is supported in part by the *Israel Science Foundation* (grant number 953/06). The work started while Shlomi Dolev was a visiting professor at LIP6 supported in part by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Microsoft, Deutsche Telekom, US Air-Force and Rita Altura Trust Chair in Computer Sciences.

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 525–527, 2010.

topology[1]. A key component of the simulation is a new bounded labeling scheme that needs no initialization, as well as a method for using it when communication links and processes are started at an arbitrary state.

Overview of our simulation. A simulation of a SWMR atomic register in a message-passing system, supports two procedures, read and write, for accessing the register. The ABD simulation [3] is based on a quorum approach: In a write operation, the writer makes sure that a quorum of processors (consisting of a majority of the processors, in its simplest variant) store its latest value. In a read operation, a reader contacts a quorum of processors, and obtains the latest values they store for the register; in order to ensure that other readers do not miss this value, the reader also makes sure that a quorum stores its return value.

A key ingredient of this scheme is the ability to distinguish between older and newer values of the register; this is achieved by attaching a *sequence number* to each register value. In its simplest form, the sequence number is an unbounded integer, which is increased whenever the writer generates a new value. This solution could be appropriate for a an *initialized* system, which starts in a consistent configuration, in which all sequence numbers are zero, and are only incremented by the writer or forwarded as is by readers. In this manner, a 64-bit sequence number will not wrap around for a number of writes that is practically infinite, certainly longer than the life-span of any reasonable system.

However, when there are transient failures in the system, as is the case in the context of self-stabilization, the simulation starts at an uninitialized state, where sequence numbers are not necessarily all zero. It is possible that, due to a transient failure, the sequence numbers might hold the maximal value when the simulation starts running, and thus, will wrap around very quickly.

Our solution is to partition the execution of the simulation into *epochs*, namely periods during which the sequence numbers are supposed not to wrap around. Whenever a "corrupted" sequence number is discovered, a new epoch is started, overriding all previous epochs; this repeats until no more corrupted sequence numbers are hidden in the system, and the system stabilizes. Ideally, in this steady state, after the system stabilizes, it will remain in the same epoch (at least until all sequence numbers wrap around, which is unlikely to happen).

This raises, naturally, the question of how to label epochs. The natural idea, of using integers, is bound to run into the same problems as for the sequence numbers. Instead, we capitalize on another idea from [3], of using a bounded labeling scheme for the epochs. A *bounded labeling scheme* [6,5] provides a function for generating labels (in a bounded domain), and guarantees that two labels can be compared to determine the largest among them.

Existing labeling schemes assume that initially, labels have specific initial values, and that new labels are introduced only by means of the label generation function. However, transient failures, of the kind the self-stabilizing simulation must withstand, can create incomparable labels, so it is impossible to tell which is the largest among them or to pick a new label that is bigger than all of them.

[1] Standard end-to-end schemes can be used to implement the quorum operation in the case of general communication graph.

To address this difficulty, we present a constructive bounded labeling scheme that allows to define a label larger than *any set* of labels, provided that its size is bounded. We assume links have bounded capacity, and hence the number of epochs initially hidden in the system is bounded.

The writer tracks the set of epochs it has seen recently; whenever the writer discovers that its current epoch label is not the largest, or is incomparable to some existing epoch, the writer generates a new epoch lable that is larger than all the labels it has. The number of bits required to represent an epoch label depends on m, the maximal size of the set, and it is in $O(m \log m)$. We ensure that the size of the set is proportional to the total capacity of the communication links, namely, $O(cn^2)$, where c is the bound on the capacity of each link, and hence, each epoch label requires $O((cn^2(\log n + \log c))$ bits.

It is possible to reduce this complexity, making c essentially constant, by employing a data-link protocol for communication among the processors.

We show that, after a bounded number of write operations, the results of reads and writes can be linearized in a manner that satisfies the semantics of a SWMR register. This holds until the sequence numbers wrap around, as can happen in a realistic version of the unbounded ABD simulation.

The labeling scheme. Let $k > 1$ be an integer, and let X be the set $\{1, 2, .., k^2{+}1\}$. $\mathcal{L}$ (the set of labels) is the set of all ordered pairs, (s, A), such that $s \in X$ and $A \subseteq X$ $|\mathcal{L}| = \binom{k^2+1}{k}k^2 + 1 = k^{(1+o(1))k}$. A label (s_i, A_i) is smaller ($\prec$) a label (s_j, A_j) if and only if $(s_j \in A_i)$ and $s_i \notin A_j$.

Given a subset S of at most k labels $(s_1, A_1), (s_2, A_2), \ldots$ in $\mathcal{L}$, we compute a new label (s_i, A_i) which is greater (with respect to $\prec$) than every label of S:

- s_i is an element of X that is not in the union $A_1 \cup A_2 \cup \ldots \cup A_k$, and
- A is a subset of size k of X containing all values $(s_1, s_2, \ldots, s_k)$.

It can be proved [1] that this element exists and that this yields a bounded labeling scheme, even with uninitialized values.

References

1. Alon, N., Attiya, H., Dolev, S., Dubois, S., Gradinariu, M., Tixeuil, S.: Practically Stabilizing Atomic Memory, arXiv 1007.1802 (2010)
2. Attiya, H.: Robust Simulation of Shared Memory: 20 Years After. EATCS Distributed Computing Column (2010)
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing Memory Robustly in Message-Passing Systems. J. ACM 42(1), 124–142 (1995)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
5. Dolev, D., Shavit, N.: Bounded Concurrent timestamping. SIAM J. on Computing 26(2), 418–455 (1997)
6. Israeli, A., Li, M.: Bonded timestamps. Distr. Comp. 6(4), 205–209 (1993)

Brief Announcement: Stabilizing Consensus with the Power of Two Choices

Benjamin Doerr[1], Leslie Ann Goldberg[2], Lorenz Minder[3], Thomas Sauerwald[4], and Christian Scheideler[5]

[1] Max-Planck Institute for Computer Science, Saarbrücken
[2] Department of Computer Science, University of Liverpool
[3] Computer Science Division, University of California, Berkeley
[4] Simon Fraser University, Burnaby, Canada
[5] Department of Computer Science, University of Paderborn

Consensus problems occur in many contexts and have therefore been extensively studied in the past. In the original consensus problem, every process initially proposes a value, and the goal is to decide on a single value from all those proposed. We are studying a slight variant of the consensus problem called the *stabilizing consensus problem* [2]. In this problem, we do not require that each process irrevocably commits to a final value but that eventually they arrive at a common, stable value without necessarily being aware of that. This should work irrespective of the states in which the processes are starting. In other words, we are searching for a *self-stabilizing* algorithm for the consensus problem. Coming up with such an algorithm is easy without adversarial involvement, but we allow some adversary to continuously change the states of some of the nodes at will. Despite these state changes, we would like the processes to arrive quickly at a common value that will be preserved for as many time steps as possible (in a sense that almost all of the processes will store this value during that period of time). Interestingly, we will demonstrate that there is a simple algorithm for this problem that essentially needs logarithmic time and work with high probability to arrive at such a stable value, even if the adversary can perform arbitrary state changes, as long as it can only do so for a limited number of processes at a time.

Our approach. We are focussing on synchronous message-passing systems with adversarial state changes. More precisely, we are given a fixed set of n processes that are numbered from 1 to n. The time proceeds in synchronized *rounds*. In each round, every process can send out one or more requests, receive replies to its requests, and perform some local computation based on these replies. We assume that every process can send a message to any other process (i.e., there are no connectivity constraints) and faithfully follows the stated protocol (given its current state, which might have been changed by the adversary).

Stabilizing consensus problem. We are studying a slight variant of the original consensus problem called the *stabilizing consensus problem* [2]. In this problem, we may start with an arbitrary set of initial output values. As in the usual convention, a *configuration* includes all of the processes' local states. A configuration C is called *output-stable* if in all possible executions starting from C, the

N.A. Lynch and A.A. Shvartsman (Eds.): DISC 2010, LNCS 6343, pp. 528–530, 2010.

output of each process does not change. If every process outputs x in an output-stable configuration C, we say the outputs stabilize to x in C. A *self-stabilizing consensus protocol* must satisfy the following three properties:

- **Stabilization:** the protocol eventually reaches an output-stable configuration.
- **Validity:** if a process outputs a value v, then v must have been output by some process in the previous round.
- **Agreement:** for every reachable output-stable configuration, all processes have the same output.

The adversary. We also assume that there is a T*-bounded adversary* that knows the entire configuration of the system in the current round. Based on that information, it may decide to introduce arbitrary state changes in up to T processes *at the end* of each round except that the protocol code cannot be manipulated. Of course, under a T-bounded adversary we cannot reach an output-stable configuration any more. Therefore, we will only require the system to reach a configuration C so that for any polynomial in n many time steps following C, all but at most $\mathcal{O}(T)$ processes agree on some stable value v (note that these $\mathcal{O}(T)$ processes can be different from round to round). We will call this an *almost output-stable configuration.*

Our contribution. We are focussing on stabilizing consensus problems based on an arbitrary (finite or countably infinite) set S of *legal* values with a total order. Classical examples are $S = \{0, 1\}$ and $S = \mathbb{N}$. All initial output values of the processes must be from S and also the adversary is restricted to choosing only values in S. (In case the adversary chooses a value outside of S in some process p, we may assume that p instantly recognizes that and then switches over to some default value in S.) We propose the following simple protocol called the *median rule*:

> In each round, every process i picks two processes j and k uniformly and independently at random among all processes (including itself) and requests their values. It then updates v_i to the *median* of v_i, v_j and v_k. Any request sent to process i will be answered with the value i had *at the beginning* of the current round.

For example, if $v_i = 10$, $v_j = 12$ and $v_k = 100$, then the new value of v_i is 12. The median rule works surprisingly well. We can prove that we reach an almost stable consensus in time $\mathcal{O}(\log m \log\log n + \log n)$ w.h.p.[1], where $m = |S|$. If no adversary is present, then the runtime decreases to $\mathcal{O}(\log m)$ w.h.p. Our results are listed in Table 1.

The work closest to our work is the paper by Angluin et al. [1] in the context of population protocols. Their consensus protocol can tolerate Byzantine behavior of $o(\sqrt{n})$ nodes. However, they only focus on two values whereas we allow any number of legal values, and in our model any node can experience state changes.

[1] We write w.h.p. to refer to an event that holds with probability at least $1 - n^{-c}$ for any constant $c > 1$.

	with adversary	without adversary
worst-case, $m = 2$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
worst-case, arb. m	$\mathcal{O}(\log m \log\log n + \log n)$	$\mathcal{O}(\log m + \log n)$
average-case, arb. m	$\mathcal{O}(\log m + \log\log n)$ if m is odd $\Theta(\log n)$ if m is even	$\mathcal{O}(\log m + \log\log n)$ if m is odd $\Theta(\log n)$ if m is even

Fig. 1. Our results on the time and work required by each process to reach an almost stable consensus (with adversary) or stable consensus (without adversary). $m = |S|$ is the number of legal values. By average-case we refer to the case where every initial value is chosen independently and uniformly at random among $m \leq n^{1/2-\epsilon}$ legal values for some constant $\epsilon > 0$. All results hold with probability at least $1 - n^{-c}$ for any constant $c > 1$.

Of course, $|S|$ may not be finite. In this case, our results still hold if we define m as the number of legal values between v_ℓ and v_r, where v_ℓ is the $(n/2 - c\sqrt{n \log n})$-smallest and v_r is the $(n/2 + c\sqrt{n \log n})$-smallest value of the initial values for some sufficiently large constant c. Therefore, as a by-product, the median rule computes a good approximation of the median, even under the presence of an adversary.

Our results also hold if instead of state changes at arbitrary processes we have a fixed number of $\sqrt{n}$ Byzantine processes. The bound on T is essentially tight as $T = \Omega(\sqrt{n \log n})$ would not allow the median rule to stabilize any more w.h.p. (even if the adversary can only put T processes to sleep instead of changing their states) because the adversary could keep two groups of processes with equal values in perfect balance for at least a polynomially long time.

Finally, if the T-bounded adversary is *static* in a sense that there is a *fixed* set of *bad* T processes that can experience adversarial state changes throughout the execution, then a simple extension of the median rule, called *careful median rule*, reaches a *stable* consensus for all non-bad processes for any polynomial number of time steps w.h.p.: each process runs the median rule as before but outputs the majority value of the last k rounds of the median rule as its value, for some constant $k \geq 3$. A stable consensus is not possible with the original median rule as with $T = \sqrt{n}$ there is a constant probability that some non-bad process contacts two bad processes and therefore changes its value to a value selected by the adversary. The details can be found in [3].

References

1. Angluin, D., Aspnes, J., Eisenstat, D.: A simple population protocol for fast robust approximate majority. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 20–32. Springer, Heidelberg (2007)
2. Angluin, D., Fischer, M., Jiang, H.: Stabilizing consensus in mobile networks. In: Gibbons, P.B., Abdelzaher, T., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, pp. 37–50. Springer, Heidelberg (2006)
3. Doerr, B., Goldberg, L.A., Minder, L., Sauerwald, T., Scheideler, C.: Stabilizing consensus with the power of two choices. Technical report, University of Paderborn (2010), http://wwwcs.upb.de/cs/scheideler

Author Index